UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Vitor Hugo Galhardo Moia

A Study on Approximate Matching for Similarity Search: Techniques, Limitations and Improvements for Digital Forensic Investigations

Um Estudo sobre Pareamento Aproximado para Busca por Similaridade: Técnicas, Limitações e Melhorias para Investigações Forenses Digitais

CAMPINAS

2020

Vitor Hugo Galhardo Moia

A Study on Approximate Matching for Similarity Search: Techniques, Limitations and Improvements for Digital Forensic Investigations

Um Estudo sobre Pareamento Aproximado para Busca por Similaridade: Técnicas, Limitações e Melhorias para Investigações Forenses Digitais

Thesis presented to the School of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Electrical Engineering in the area of Computer Engineering.

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Engenharia Elétrica, área de Engenharia de Computação.

Supervisor: Prof. Dr. Marco Aurélio Amaral Henriques

Este trabalho corresponde à versão final da tese defendida pelo aluno Vitor Hugo Galhardo Moia, e orientada pelo Prof. Dr. Marco Aurélio Amaral Henriques

CAMPINAS

2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

M727s

Moia, Vitor Hugo Galhardo, 1990-
A study on approximate matching for similarity search : techniques, limitations and improvements for digital forensic investigations / Vitor Hugo Galhardo Moia. – Campinas, SP : [s.n.], 2020.

Orientador: Marco Aurélio Amaral Henriques.
Tese (doutorado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Computação forense. 2. Crime por computador - Investigação. 3. Funções hash. 4. Documentação - Identificação. I. Henriques, Marco Auréio Amaral. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Um estudo sobre pareamento aproximado para busca por similaridade : técnicas, limitações e melhorias para investigações forenses digitais
Palavras-chave em inglês:
Computer forensics
Computer crimes - Investigation
Hash functions
Fingerprints - Identification
Área de concentração: Engenharia de Computação
Titulação: Doutor em Engenharia Elétrica
Banca examinadora:
Marco Aurélio Amaral Henriques [Orientador]
Ricardo Felipe Custódio
Routo Terada
Ricardo Dahab
Maurício Ferreira Magalhães
Data de defesa: 20-02-2020
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-0396-2873
- Currículo Lattes do autor: http://lattes.cnpq.br/0120943194989207

COMISSÃO JULGADORA - TESE DE DOUTORADO

Candidato: Vitor Hugo Galhardo Moia RA: 153843
Data da Defesa: 20 de fevereiro de 2020
Título da Tese: A Study on Approximate Matching for Similarity Search: Techniques, Limitations and Improvements for Digital Forensic Investigations.
Título da Tese em outro idioma: Um Estudo sobre Pareamento Aproximado para Busca por Similaridade: Técnicas, Limitações e Melhorias para Investigações Forenses Digitais.

Prof. Dr. Marco Aurélio Amaral Henriques (Presidente, FEEC/UNICAMP)
Prof. Dr. Ricardo Felipe Custódio (UFSC)
Prof. Dr. Routo Terada (USP)
Prof. Dr. Ricardo Dahab (IC/UNICAMP)
Prof. Dr. Maurício Ferreira Magalhães (FEEC/UNICAMP)

Ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

I dedicate this thesis to my parents Marcelo and Maria Lúcia, whose support was fundamental in this journey. This work was only possible because of them.

I also dedicate this work to my sister Bianca and my brother Vinicius who have never left my side. A special feeling of gratitude to my relatives, especially my goddaughter Lara, my aunt Maria Alice and my grandmother Maria Teresa, who always believed in me during these years.

Finally, I dedicate this work to God, for giving me this life-changing opportunity and the strength to overcome all the obstacles found in this journey.

Acknowledgements

First, I would like to thank my supervisor Professor Marco Aurélio Amaral Henriques. He has always guided and helped me during the program. I owe him many thanks for all his patience, support, and dedication.

Many thanks to the FEEC professors who have shared their knowledge with me during classes and the whole school staff for allowing me to conduct my research and providing me assistance.

I want to thank Professor Frank Breitinger from the University of New Haven, USA, for having me in his laboratory during a three-month internship. Our collaboration was fruitful, and part of the work is presented in this thesis. Thanks for all of your support.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. I am grateful for the support provided under the project FORTE - Timely and efficient digital forensics / Forense digital tempestiva e eficiente (grant number: 23038.007604/2014-69). Besides, I would like to thank Santander International Mobility Program for Graduate Students for the partial financial support during my three months internship at the University of New Haven, USA.

A special thanks to my friends of Andradas and also the ones I had the pleasure to meet during these four years studying at Unicamp, especially the ones from the Department of Computer Engineering and Industrial Automation (DCA) of FEEC/Unicamp. Finally, thanks to Daniela for all the understanding, your love and kindness.

Abstract

Digital forensics is a branch of Computer Science aiming at investigating and analyzing electronic devices in the search for crime evidence. With the rapid increase in data storage capacity, the use of automated procedures to handle the massive volume of data available nowadays is required, especially in forensic investigations, in which time is a scarce resource. One possible approach to make the process more efficient is the Known File Filter (KFF) technique, where a list of interest objects is used to reduce/separate data for analysis. Holding a database of hashes of such objects, the examiner performs lookups for matches against the target device under investigation. However, due to limitations over cryptographic hash functions (inability to detect similar objects), new methods have been designed based on Approximate Matching (AM). They appear as suitable candidates to perform this process because of their ability to identify similarity (bytewise level) in a very efficient way, by creating and comparing compact representations of objects (*a.k.a. digests*). In this work, we present the Approximate Matching functions. We show some of the most known AM tools and present the Similarity Digest Search Strategies (SDSS), capable of performing the similarity search (using AM) more efficiently, especially when dealing with large data sets. We perform a detailed analysis of current SDSS approaches and, given that current strategies only work for a few particular AM tools, we propose a new strategy based on a different tool that has good characteristics for forensic investigations. Furthermore, we address some limitations of current AM tools regarding the similarity detection process, where many matches pointed out as similar, are indeed false positives; the tools are usually misled by common blocks (pieces of data common in many different objects). By removing such blocks from AM digests, we obtain significant improvements in the detection of similar data. We also present a detailed theoretical analysis of the capabilities of the `sdhash` AM tool and provide some improvements to its comparison function, where our improved version has a more precise similarity measure (score). Lastly, new applications of AM are presented and analyzed: One for fast file identification based on data samples and another for efficient fingerprint identification. Through our results, we hope that practitioners in the forensics field and other related areas will benefit from our studies on AM when solving their problems.

Keywords: Digital Forensics, Known File Filtering, Approximate Matching, Similarity Search, Similarity Digest Search Strategies, Common blocks, Jaccard Similarity.

Resumo

A forense digital é apenas um dos ramos da Ciência da Computação que visa investigar e analisar dispositivos eletrônicos na busca por evidências de crimes. Com o rápido aumento da capacidade de armazenamento de dados, é necessário o uso de procedimentos automatizados para lidar com o grande volume de dados disponíveis atualmente, principalmente em investigações forenses, nas quais o tempo é um recurso escasso. Uma possível abordagem para tornar o processo mais eficiente é através da técnica KFF (Filtragem por arquivos conhecidos - *Known File Filtering*), onde uma lista de objetos de interesse é usada para reduzir/separar dados para análise. Com um banco de dados de hashes destes objetos, o examinador realiza buscas no dispositivo de interesse sob investigação por qualquer item que seja igual ao buscado. No entanto, devido a limitações nas funções criptográficas de hash (incapacidade de detectar objetos semelhantes), novos métodos foram projetados baseando-se em funções de Pareamento Aproximado (ou *Approximate Matching*) (AM). Estas funções aparecem como candidatos para realizar buscas uma vez que elas têm a capacidade de identificar similaridade (no nível de bytes) de uma maneira eficiente, criando e comparando representações compactas de objetos (conhecidos como *resumos*). Neste trabalho, apresentamos as funções de Pareamento Aproximado. Mostramos algumas das ferramentas de AM mais conhecidas e apresentamos as Estratégias de Busca por Similaridade baseadas em resumos, capazes de realizar a busca de similaridade (usando AM) de maneira mais eficiente, principalmente ao lidar com grandes conjuntos de dados. Realizamos também uma análise detalhada das estratégias atuais e, dado que as mesmas trabalham somente com algumas ferramentas específicas de AM, nós propomos uma nova abordagem baseada em uma ferramenta diferente que possui boas características para investigações forenses. Além disso, abordamos algumas limitações das ferramentas atuais de AM em relação ao processo de detecção de similaridade, onde muitas comparações apontadas como semelhantes, são de fato falsos positivos; as ferramentas geralmente são enganadas por blocos comuns (pedaços de dados em comum encontrados em muitos objetos diferentes). Ao remover estes blocos dos resumos de AM, obtemos melhorias significativas na detecção de objetos similares. Também apresentamos neste trabalho uma análise teórica detalhada das capacidades de detecção da ferramenta de AM `sdhash` e propomos melhorias em sua função de comparação, onde a versão aprimorada apresenta uma medida de similaridade (score) mais precisa. Por último, novas aplicações de AM são apresentadas e analisadas: uma de identificação rápida de arquivos por meio de amostragem de dados e outra de identificação eficiente de impressões digitais. Através de nossos resultados, esperamos que profissionais da área forense e de outras áreas relacionadas se beneficiem de nosso estudo sobre AM para resolver seus problemas.

Palavras-chaves: Forense Digital, Filtragem por Arquivos Conhecidos, Pareamento Aproximado, Busca de Similaridade, Estratégias de Busca de Similaridade baseadas em Resumos, Blocos comuns, Similaridade de Jaccard.

List of Figures

List of Tables

Contents

Appendix 202

1 Introduction

Digital forensics is a branch of forensics aiming at investigating digital devices in the search for crime evidence. According to Raghavan, S. (RAGHAVAN, 2013), digital forensics is a multi-staged process involving a sequence of activities, such as evidence identification, acquisition, examination, analysis, documentation and preservation. All procedures adopted during an investigation must be concerned with two issues: Integrity and authenticity. The first one ensures that the act of seizing and acquiring digital evidence does not modify it. In contrast, the second one allows one to confirm the integrity of the acquired information.

Applications of digital forensics include: Criminal investigations (child pornography, identity thief, e-crimes), civil litigation (e-discovery), and intelligence (terrorist attacks). Besides, this area of forensics have sub-branches related to the type of digital device under analysis. An example of some common branches are: Disk forensics, mobile forensics, network forensics, memory analysis and cloud forensics.

Among digital forensic branches, one problem that all areas are facing today is that, due to technology improvements, storage devices' capabilities have increased significantly in the past years. The growth is a result of the popularity of digital devices that became more accessible to people due to a decrease in costs. This trend imposes a severe challenge on forensic practitioners, who, even in ordinary investigations, have to handle terabytes of digital evidence (GARFINKEL, 2010; RAGHAVAN, 2013; QUICK; CHOO, 2014; LILLIS *et al.*, 2016). Moreover, according to Quick, D. and Choo, K. (QUICK; CHOO, 2014), there is a potential gap in the processing power and storage capabilities improvements, showing that we have more data than processing power as time passes. This observation is based on the predictions of Moore and Kryder's laws. The first one states that the number of transistors (and, indirectly, the computing power) doubles on an integrated circuit every 18-24 months, while the other says that the storage density of hard drives doubles every 12 months; in short, we have more space to store data than processing power for handling it. Consequently, the time and effort to undertake analysis on seized devices remain a challenge, forcing practitioners to explore solutions to handle the massive volume of data in a short time.

One possible approach to deal with the problem mentioned above more efficiently is the Known File Filter (KFF) method, which separates relevant from irrelevant information in prior analysis. In the triage phase, practitioners can use lists of interest objects to remove known good files from the analysis (objects of operating systems, known software, and other inoffensive ones in a white list) and/or separate bad ones (illegal or suspicious objects in a blacklist). Cryptographic hash functions (e.g., MD5, SHA-1, SHA-2, etc.) are a straightforward technique to perform KFF. Indeed, NIST (National Institute of Standards and Technology) (NIST, 2016) provides a hash database of *good* files that can be used by practitioners to do the filtering process, called NSRL (National Software Reference Library). An example of the adoption of KFF in practice can be seen in the well-establish forensic tool Autopsy (AUTOPSY, 2015), whose goal is to analyze Windows and UNIX disk and file systems. Among all the modules with distinctive functions of the tool, a particular one, called Hash Database Lookup Module, uses hash databases like the NSRL or others added by the practitioner, to determine if a file is known bad, known good, or unknown, based on its MD5 hash value (AUTOPSY, 2016).

However, cryptographic hash functions do not perform well in the scenario presented here since even small changes in an object will produce an entirely different hash, which will make the correlation of the original object and its modified version almost impossible. This way, attackers can bypass investigations by inserting random bytes in malicious objects. Also, since some objects get newer versions frequently (e.g., operating system updates), keeping a hash for every release is infeasible due to the database size required and the difficulties to keep it up to date.

Suitable candidates to mitigate hash limitations are the Approximate Matching (AM) functions. Using a small and compact data representation (digest), they can identify the similarity between objects in a way that similar objects will have similar digests; small changes in the input object will reflect in minor variations in its digest representation. When comparing two digests, AM produces a score related to the amount of content shared between them. Usually, if a minimal and pre-defined value (or threshold) is achieved, they can be considered as similar. The use of these functions in digital forensics is very beneficial, mainly when used as a data reduction technique. As shown by Breitinger, F. et al. (BREITINGER *et al.*, 2014b), this sort of function can increase the file identification rate significantly (from 1,82% using the traditional cryptographic hash to 23,76% with approximate matching). Furthermore, AM can be applied to digital forensic investigations in the examination phase, especially in a triage, dealing with a copy of the data acquired from seized devices (e.g., computers, mobiles, tablets, or any other electronic device capable of storing valuable data).

The downside of AM is that they are computationally more expensive than traditional hash functions. Comparing two digests is not as straightforward as the string comparison of traditional hashes. Specifically, AM requires a particular function to determine the similarity of two digests that is more complex and tool-dependent. Besides, current functions present other limitations regarding compression rates and/or precision. Each solution usually focuses on addressing satisfactorily one of these aspects to the detriment of others.

Another obstacle of the AM adoption for KFF is that the straightforward comparison solution usually adopted is the brute force (all-against-all). In this mode, every reference list digest is compared to every digest created for the target device under analysis. This way, the complexity of this search is quadratic, and the whole process becomes very time-consuming.

Given the limitations of the field, new solutions are required to handle the massive volume of data found in investigations nowadays. In this work, we present new solutions to deal with the current limitations of AM. We aim at showing how a digital forensic practitioner could use Approximate Matching functions to perform a similarity search, where large data sets are compared to identify similar objects. Our main goal is to answer the following leading research question:

Leading RQ. How can we perform a similarity search (from a digital forensic practitioner perspective) over large data sets in a (time/space) efficient manner?

To help us answer this general question, we elaborated and explored other and more specific research questions to cover many topics over the AM field.

RQ1. Can AM functions deal with huge data sets efficiently? How would digital forensic investigations benefit from the use of such functions?

RQ.2 How can we estimate the (theoretical) minimum similarity detected by AM functions?

RQ3. How can we improve current AM tools to perform better over large data sets and produce more reliable results?

RQ.4 Is there any other application for AM functions?

Topics outside the scope of this work: We emphasize that the following topics will not be addressed in this thesis.

- Adversary argument: According to Flowers, M. et al. (FLOWERS *et al.*, 1982), adversary argument encompasses the situation where we have two participants and neither of them expects to persuade or be persuaded; their intention is to remain adversaries and present arguments to make their side look good while making the opponent's look bad. In our scenario, the participants are the digital forensic practitioner and a malicious user who seeks to hide information, such as crime evidence. Both participants will try their best to accomplish their goal.
- Syntactic/Semantic similarity: All object similarity evaluation performed in this thesis does not take into consideration the internal structure of objects nor try to interpret them. The similarity is only assessed by looking at the object bytes.
- Stream data: No solution regarding stream data similarity is considered in this thesis.

Assumptions: This thesis considers the following assumption:

- Most of the solutions provided here consider that the devices under analysis, such as seized media, for instance, allow a complete or partial recovery of the objects stored on them, which means that the file system of the operating system is intact. Only in the last chapter of our work we deal with media having a corrupted file system, where no information about how to restore objects is available. This way, we only have access to shreds of objects.

The remainder of the text is structured as follow to deal with our research questions.

- Chapter II: Discuss the main problems we are facing in this work and presents Approximate Matching and other related terms, as well as the main tools of the field.
- Chapter III: Compares the similarity digest search strategies and show their benefits and limitations.
- Chapter IV: Proposes a new similarity digest search strategy and present some results when performing a similarity search.
- Chapter V: Presents, defines, and discusses the common blocks and proposes a new way to identify and remove these blocks from the similarity assessment. Results discuss how the common blocks affect digital forensics investigations.
- Chapter VI: Performs a theoretical analysis of the `sdhash` AM tool and shows some limitations. A new tool, called `J-sdhash`, is proposed and its benefits are discussed.
- Chapter VII: Proposes new applications for AM functions: Fast file identification (using sampling techniques) and Fingerprint identification (when dealing with larger data sets of templates).
- Chapter VIII: Gives the conclusions and some directions for future works.

2 Background and Related Work

The main concepts in the Approximate Matching field are presented in this chapter. Our study focus in the tools that operate in the bytewise level, where only the object bytes are analyzed when assessing similarity. We highlight that it is not in the scope of this chapter to provide a detailed and complete analysis of the field but to present the necessary background for the understanding of our contributions in this area. We begin by presenting the problem which led us to the AM field and possible solutions to it found in the literature.

2.1 Problem description

Finding similar data (from a digital forensics perspective) in documents, pictures, videos, or programs that share some pieces in common on large data sets is one of the problems faced by digital forensics practitioners nowadays. Due to technology improvements, users have digital devices with terabytes of data, given the easy access to storage media (external hard drives, pendrives, etc) with great capacities. The increase in media capacity makes forensic practitioners' job a challenging task, where terabytes of data have to be inspected in the search for crime evidence. Usually, white/black-listing techniques are employed, where one database of interest objects is contrasted to the media under investigation for the search of equal/similar data. All this process must happen in a time/space-efficient manner, which is a problem.

The identification of exact duplicates can be easily solved with cryptographic hashes (e.g, SHA-1, SHA-2 etc.), which are efficient; however, they can not deal with similar duplicates. For addressing such limitation, AM tools were developed.

2.2 What is similarity?

According to the Oxford dictionary, similarity is "*the state of fact of being similar*". The same dictionary defines the term similar as "*having resemblance in appearance, character, or quantity, without being identical*". Dekang, L. (LIN, 1998) states that:

1. The more commonality two objects share, the more similar they are.
2. The more differences two objects have, the less similar they are.
3. The maximum similarity between two objects is achieved when they are identical.

Furthermore, there are two categories in which similarity can be expressed: Resemblance and Containment. Broder, A.Z. (BRODER, 1997) defines resemblance when two objects resemble each other, while containment when one object is contained inside another. In digital forensic investigations, we may encounter both types of similarity. For instance, when looking for a picture, we may encounter other pictures with the same dimensions and similar content (resemblance) but also can find this picture inside a book with many other images and text (containment). Which similarity type we should focus on depends on the investigation goals.

2.3 Cryptographic hash functions

Cryptographic hash functions have many applications on several security areas, such as message authentication, digital signatures, one-way password, among others, including data integrity. In the forensics field, it plays a vital rule related to data integrity, where we can quickly verify whether two objects are identical or not. One of the applications of this latter example is on KFF (Known File Filtering), when we search for objects in a target device using a reference list, which is the focus of this work.

When looking for duplicate objects, the best option is to use cryptographic hash functions. This sort of technique is already well explored and for the purpose of detecting equal artifacts is the most appropriate option due to its interesting characteristics: Compact representations (a small fixed-size digest to represent an object of any size) and fast digest generation and comparison processes. Besides, modern processors have specific instructions to process hash functions more efficiently (INTEL, 2013).

According to Stallings, W. (STALLINGS, 2014), the security requirements for cryptographic hash functions are:

- Variable input size: The hash function H can be applied to data of any size;
- Fixed output size: A fixed-length output is produced by H;
- Efficiency: Computing $H(x)$ is relatively easy for any given input data x, leading to practical hardware and software implementations;
- Preimage resistance (one-way property): It is easy to generate a hash value h from a message y, but it must be computationally infeasible to generate the message y from the hash value h;
- Second preimage resistance (weak collision resistance): It must be computationally infeasible to find an alternative message y with the same hash value of a message x;
- Collision resistance (strong collision resistance): It must be computationally infeasible to find any two messages (x,y) such that their hash values are the same $(H(x) = H(y))$.

Another characteristic of a good hash function is that applying it to a broad set of inputs will produce evenly distributed and apparently random hash values. Furthermore, changing a single or several bits in the input data will produce, with high probability, a significant change in the hash code. Examples of hash functions are MD5, SHA-1, SHA-2, SHA-3, among others. Related to attacks on cryptographic hash functions, Stallings, W. says that collisions will exist in any good function, but it must be computationally infeasible to find them.

When using cryptographic hash functions for KFF with a black/white list, a severe limitation arrives: The identification of similar data. Such functions are very sensitive to even small changes in the input data, since the hash value of the original object will be completely different from the one of an object with the same content but differing in a single bit. Fixing this issue demands to store hashes for every new version of the object. However, as software, libraries and operating systems files are constantly updated, keeping a hash database for every single change is difficult to keep it up to date. Furthermore, attackers can bypass investigations by inserting random bytes in malicious objects. For this reason, we focus on approximate matching functions as a possible solution to mitigate hashes limitations.

2.4 Approximate matching

Approximate matching functions are defined by NIST (BREITINGER *et al.*, 2014b) as a *"Promising technology designed to identify similarities between two digital artifacts. It is used to find objects that resemble each other or find objects that are contained in another object"*. They can be classified according to their operational level, as follows:

- Bytewise[1] Matching relies on the byte sequence of the object. Since it does not try to interpret data either consider any structure within it, this level is more efficient than others and is format independent. Bytewise functions are also known as similarity hashing or fuzzy hashing.

- Syntactic: Relies on the internal structure of the object. For this reason, they are format dependent but does not interpret the content of the object to produce results. For example, the structure of a TCP network packet could be used to match packets from the same source and/or destination.

- Semantic: Relies on the contextual attributes of the object. It is also known as perceptual hashing or robust hashing, and it is closer to human perception. In this operational level, the object is interpreted and hence format dependent. Therefore, it is more expensive. The similarity of `JPG` and `PNG` images with the same content is an example: Their byte structures are different due to encoding, although the picture is the same.

[1] Although NIST differentiates the bytewise and syntactic level and separates them, one can argue that the first one can be classified as part of the second. The bytewise level considers an abstraction created by users to manipulate data (byte structure). To be more precise, one should refer to a bitwise level (free of any internal structure) and considers the bit as a unit. However, in this thesis, we will adopt the definition used by NIST and followed by most of the AM literature.

In this thesis, we focus on the bytewise level because of its interesting characteristics: Format independence and efficiency. In a triage process, forensic examiners must produce results as quickly as possible when dealing with a huge amount of data. This way, approximate matching appears as a suitable candidate for a first step into separating devices that potentially have evidence from those that do not. More details about the syntactic and semantic levels can be found in Dorneles, C. F. et al.'s (DORNELES *et al.*, 2011) work.

2.4.1 Main steps of any AM function

Every tool that implements AM concepts has at least two main functions: Digest generation and comparison. These two functions are presented next.

2.4.1.1 Digest generation process

To create a similarity digest, AM functions usually perform the following steps:

Extraction The object bytes are read and some information is extracted from them to compose the digest. We will refer to the information extracted in this step as a *feature*, independent of the AM tool. However, we highlight that the concept of a feature may change from tool to tool.

Selection/filtering After extracting the features from an object, some tools ended up with many features to deal with. Some features may not be as important as others (e.g., a sequence of all 0's in a byte sequence) or there are too many features that using all of them will create large digests. For this reason, some of the features are discarded.

Codification (digest generation) The last process is the codification of features into a digest, a byte sequence that can be easily compared to another digest to indicate the level of similarity between two objects. Some tools use as digest data structures such as bloom filters (BLOOM, 1970), ASCII characters, vectors, among others.

2.4.1.2 Digest comparison process

After creating the digest, we need to compare it to the digest of another object to assess how similar they are. For this reason, the second function present in any AM tool is the comparison function. We emphasize that this function depends on the data structure used to codify the features performed in the digest generation process. For instance, if we use bloom filters to store the features, our comparison function is the one that receives two filters and returns the number of bits in common between them. Another example is the use of ASCII characters to represent each feature of the object. We could just use the Edit distance (UKKONEN, 1983) to measure how distant a character is from its correspondent in the other digest and then repeat the measurement for the whole digest.

After comparing two digests, the comparison function returns a value indicating how similar two objects are, which can be in a fixed interval (0-100), for example.

2.4.2 Cryptographic hashes X Approximate matching

In comparison to traditional hash functions (e.g. MD5, SHA-1, SHA-2) where every bit change in the input is expected to cause a dramatic change in the digest and only binary answers are provided (two objects are equal or not), AM provides a confidence measure about the similarity shared between two objects. Some methods provide an answer in a fixed interval: 0-100, 0-128, etc.; others provide a value indicating dissimilarity.

We can also compare traditional hashes to approximate matching functions regarding the digest length. The output size of traditional hashes is fixed independent of the input, while approximate matching produces either a fixed or variable size output (proportional to the input), depending on the method. They also are more expensive than traditional hashes in both processes: Digest generation and comparison. Approximate matching methods need a special function designed to compare digests, requiring a more complex computation and processing due to their singularities and largest digests.

In short: Hashes are less computationally expensive and only say whether two objects are equal or not. AM requires more computational power in both digest and comparison functions but it is able to identify similarity in its two forms: Resemblance and Containment.

2.4.3 Applications

The range of applications for AM is vast. One can use it to identify new versions of documents and software, embedded objects (e.g. jpg file inside a word document), objects in network packets (without reconstructing the packet flow), locate variants of malware families, clustering, code reuse (intellectual property protection and/or bug detection), detection of deleted objects (fragments remaining on disk), deduplication on storage systems (e.g. cloud computing - save storage and bandwidth), cross-device deduplication, among others (ROUSSEV, 2011; HARICHANDRAN *et al.*, 2016; LI *et al.*, 2015). Besides, Bjelland et al. (BJELLAND *et al.*, 2014) present other scenarios in which AM can be used and show practical experiments where forensics can benefit from this technology. In one experiment, they look for emails using a small set given as leads to figure out other similar ones. The search uses an email database, and from the results, unexpected information about alternative conversations was revealed. The other scenario presented by Bjelland determined successfully that cryptographic software was downloaded in a machine by just analyzing recorded network traffic.

2.4.4 Approximate matching vs. Locality-sensitive hashing: Candidates to the Nearest Neighborhood Search problem

It is important to pay attention to an usual misleading of concepts between two different functions: Approximate matching (AM) and locality-sensitive hashing (LSH). The idea behind the LSH is to map similar objects into the same positions in a structure (e.g., table) with high probabilities. This structure, usually a table, will store all objects for which one wants to check for similarity. This function is usually applied in the data mining field, for clustering or resolving the Nearest Neighborhood Search problem (see next). Besides, LSH usually uses semantic information extracted from objects (or their attributes) to find similar objects in a given set. A broader view of LSH techniques can be found in Wang, J. et al.'s work (WANG *et al.*, 2014a). AM functions, on the other hand, are designed to produce a digest from the object and, by comparing two object digests, establish a confidence measure about their similarity.

The Nearest Neighborhood Search (NNS) is an optimization problem that aims to find the closest (or most similar) object in a given set to a given object (ANDONI, 2009). It is a broader concept that can be applied to many areas, such as pattern recognition, computer vision, databases, among others. Besides, there are many variants of NNS like *k-nearest neighbors* (*k* most similar objects), *Approximate nearest neighbor*, *Nearest neighbor distance ratio*, *Fixed-radius near neighbors*, and *All nearest neighbors*. LSH is one of the most common functions employed for solving the NNS problem.

We can define NNS for digital forensics investigations. Here, we can create a new definition of NNS based on the investigator perspective for this particular context such as "finding the file in a given data set (media under analysis) that is closest (or most similar) to a given file (of a reference list)"; for establishing closeness, we can use AM. Note that here we used the term file indicating that the similarity is related to the object bytes as well as its features are bytewise, and no semantic interpretations are performed. Given the definition below, we can perform NNS for digital forensics investigations as well and use AM to this end.

2.4.5 Some known AM tools

There are many tools that implement the concepts of AM to perform the similarity identification by using digests at the byte level. Among them, we can mention the ones that have better characteristics and are the target of constant research:

- `Nilsimsa` (DAMIANI *et al.*, 2004);
- `ssdeep` (KORNBLUM, 2006);
- `bbhash` (BREITINGER; BAIER, 2012a);
- `sdhash` (ROUSSEV, 2010);

- mrsh (ROUSSEV *et al.*, 2007) and mrsh-v2 (BREITINGER; BAIER, 2013);
- mvhash-b (BREITINGER *et al.*, 2013)
- TLSH (OLIVER *et al.*, 2013);
- saHash (BREITINGER *et al.*, 2014c);
- LZJD (RAFF; NICHOLAS, 2018).

Even though many functions were developed, none of them was able to address the main requirements for a proper AM function: Efficient digest generation and comparison, high detection capabilities and low space requirement. Nilsimsa mainly suffers from significantly high false positive rates (OLIVER *et al.*, 2013), while ssdeep works only for relatively small objects of similar sizes and cannot stand an active adversary against some attacks, such as anti-blacklisting and pre-computation of trigger sequences (BAIER; BREITINGER, 2011). bbhash is quite slow compared to other tools, and sdhash suffers from the same problem and also from another related to the low digest compression rate (digests are about 2.6% of the input object size). Although mvhash-b is very fast, it requires a specific configuration for each file type (HARICHANDRAN *et al.*, 2016) and seems to work only with objects of similar size. mrsh-v2 has detection capabilities worse than sdhash. TLSH, although being robust in the detection of small changes on objects, is less powerful than sdhash and mrsh-v2 with respect to containment similarity detection. At last, saHash works only for objects of similar sizes, restricting its application to only resemblance detection.

Next, we summarize some of the aforementioned tools that are explored along this work. We give special attention to sdhash since this tool is target of improvements in the following chapters of this thesis. Then, we present the similarity digest search strategies, which make use of AM tools to perform more efficient searches.

2.4.5.1 Block-based hashing

This is the most basic approximate matching function, where data is broken into fixed-size blocks (e.g., 512 bytes) and hashes are computed for each of them. The final signature is the concatenation of all hashes, as shown in Fig. 1. The similarity between two objects is measured by counting the number of common hashes. The dcfldd tool (HARBOUR, 2002), an extension of the disk dump tool dd, implements this scheme.

Although the Block-based hashing scheme is computationally efficient and straightforward to implement, it suffers from alignment issues. The insertion/deletion of a single bit at the beginning of the input will affect the content of all remaining blocks, and their hashes will be completely different. Besides, it cannot detect containment similarity.

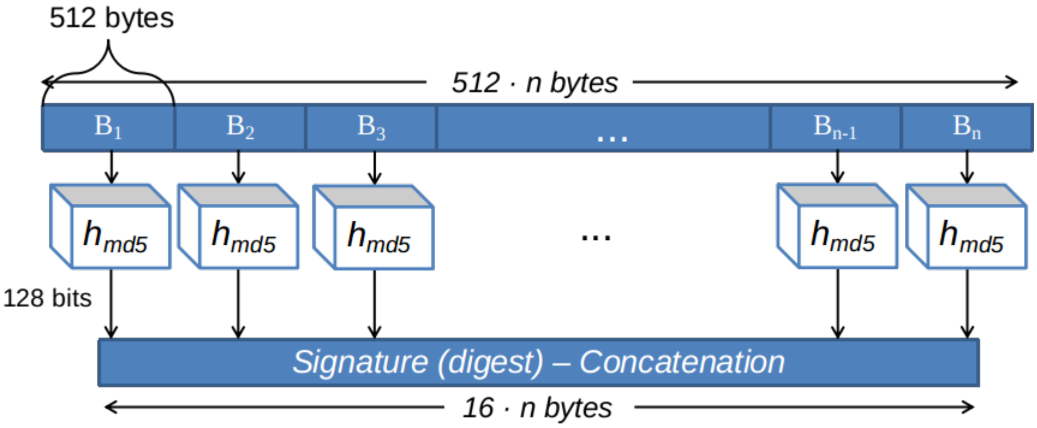


Figure 1 – Creating a digest with a Block-based hash approach.

2.4.5.2 Nilsimsa

Damiani, E. et. al. (DAMIANI *et al.*, 2004) proposes `Nilsimsa` as a method to detect spam messages. The technique consists in using a fixed-size sliding window (5 bytes) that goes byte-by-byte through the input and produces trigrams of possible combinations of the input characters. A subset of trigrams (eight from the ten possible ones for a 5 bytes sliding window) are mapped to a array of 256 positions (of integers), called accumulator, using a particular hash function; we highlight that some trigrams are removed because the last bytes of the of the sliding window repeats in the next window. Every time a position in the accumulator is selected, its value, initially set to zero, is incremented. After processing the entire input, all accumulator positions whose value is above a given threshold are set to one in a new array; the remaining ones are set to zero. This new array is the final digest (32 bytes). Fig. 2 shows the whole process.

`Nilsimsa` compares two digests by checking the number of identical bits in the same position (Hamming distance). The result is adjusted, and the range varies from 0 (dissimilar objects) to 128 (identical or very similar objects).

2.4.5.3 ssdeep

Adapted from Tridgell's spam email detector algorithm (TRIDGELL, 2002), Kornblum developed `ssdeep` (KORNBLUM, 2006). This tool is based on the Content Triggered Piecewise Hashing (CTPH) method, which is also meant to detect content similarity in the byte level, proposed by Tridgell. The main idea of `ssdeep` is to create variable-sized blocks using a rolling hash algorithm to determine the block boundaries (when it starts and stops). The rolling hashing function produces a random value based on a window that moves through the input object byte-by-byte. After the first value is generated, the next ones are created very quickly given the old hash value, the removed part of the window and the new added byte. The algorithm adopted by `ssdeep` was inspired in the Adler-32 checksum (KORNBLUM, 2006).

When generating a `ssdeep` digest, a sliding window of fixed-size (7 bytes) moves

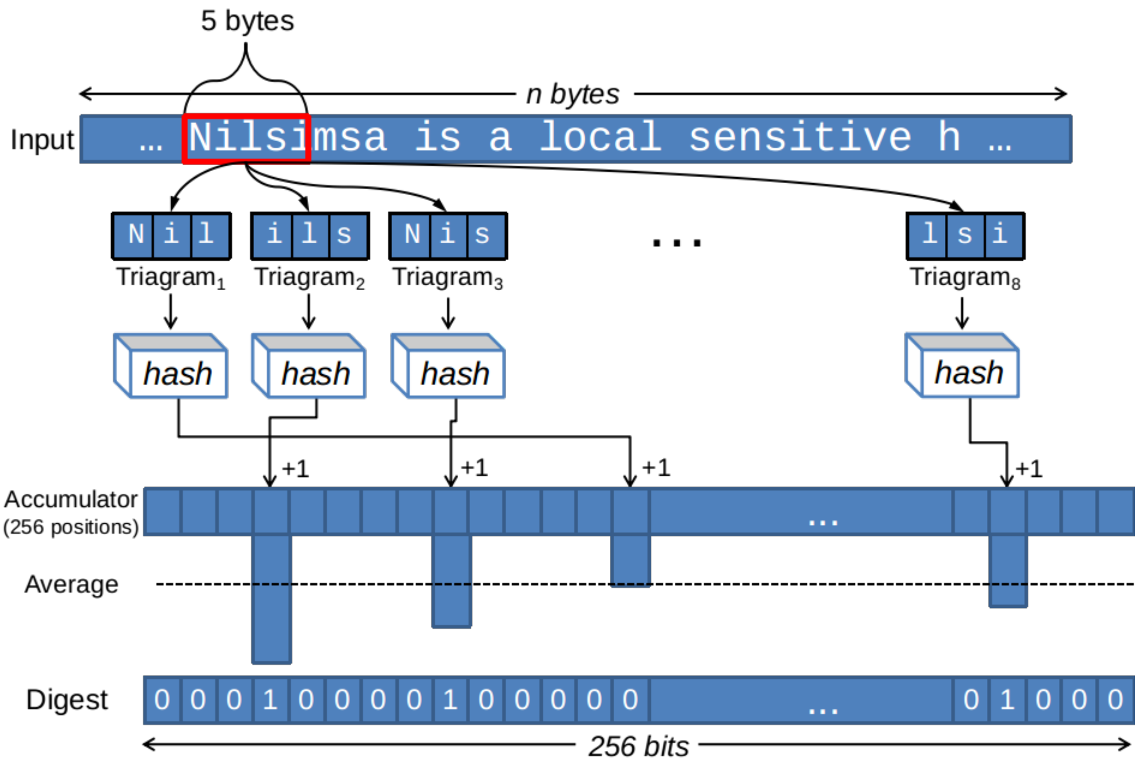


Figure 2 – Creating a digest with `Nilsimsa` (based on the work of (DAMIANI *et al.*, 2004)).

through the input, byte-by-byte, and whenever the rolling hash produces a specific output, based on the current bytes in the window, `ssdeep` identifies a trigger point, denoting the ending and beginning of a block. Then, all generated blocks are hashed using a cryptographic hash function (FNV (NOLL, 2012)) and the 6 least significant bits of each hash is encoded using a Base64 character. The final digest is the concatenation of all characters generated through the blocks. Figs. 3 and 4 depict the digest generation process of `ssdeep`. For a detailed explanation, including trigger values and required parameters, see (KORNBLUM, 2006).

To compare two digests, `ssdeep` uses the Edit Distance algorithm (UKKONEN, 1983). This function counts the minimum number of operations required to transform one string (digest) into another, using weighted operations, like insertion, deletion, substitution (single character), and transpositions (two adjacent characters). The result is a number in the range 0-100, where 0 means that the two objects are dissimilar and 100, a perfect match.

This scheme is not as sensitive to alignment issues as the Block-based hashing, and insertions/deletions are expected to have a minor impact on the similarity score. Although this is one of the most known schemes for approximate matching, it works only for relatively small objects of similar sizes, mostly due to one of the rolling hash parameters being the object size. One way found to increase its detection capabilities regarding the object size was using two digests per item instead of one. When creating the digest, `ssdeep` uses two different values as trigger point (derived from the object size, called block size and represented by *b*). The result is the creation of two digests, where the first corresponds to a trigger value *b* and the second 2*b*.

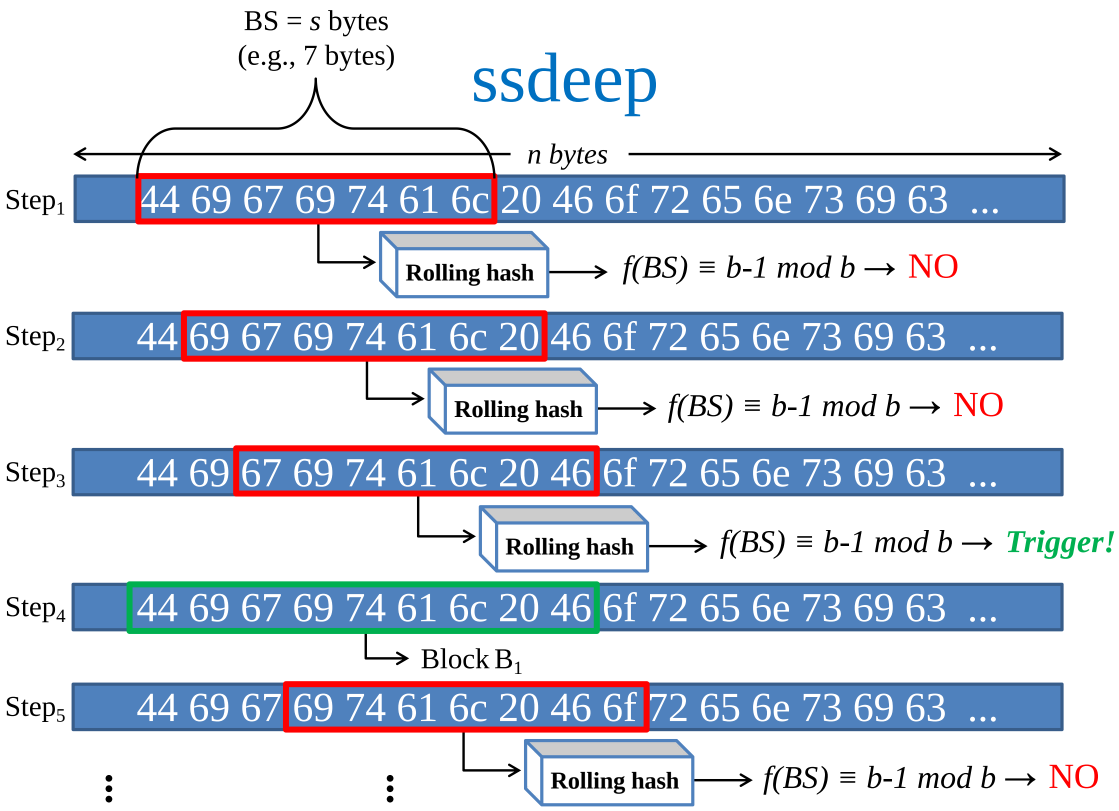

BS = s bytes
(e.g., 7 bytes)
ssdeep
n bytes
Step1
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 ...
Rolling hash
f(BS) ≡ b-1 mod b → NO
Step2
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 ...
Rolling hash
f(BS) ≡ b-1 mod b → NO
Step3
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 ...
Rolling hash
f(BS) ≡ b-1 mod b → Trigger!
Step4
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 ...
Block B1
Step5
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 ...
Rolling hash
f(BS) ≡ b-1 mod b → NO

Figure 3 – Identifying object blocks with the rolling hash function in ssdeep (based on the work of (KORNBLUM, 2006)).

This way, it is only possible to compare objects if their block sizes differ by a factor of two at most. Also, the first digest is always two times larger than the second, resulting in digest lengths of up to 64 and 32 bytes, respectively.

ssdeep has been object of researches that address its limitations, especially regarding performance (CHEN; WANG, 2008; BREITINGER; BAIER, 2012b). A security analysis was also done and concluded that ssdeep is vulnerable to an active attack (BAIER; BREITINGER, 2011).

2.4.5.4 sdhash

One of the most popular AM tools is sdhash, proposed by Roussev, V. (ROUSSEV, 2010) in 2010. The main purpose of this tool is to identify and pick features (byte sequences) from an object that are least likely to occur by chance in other objects (according to the feature entropy value) and use these features to represent the given object. Here, we present sdhash and its main working process, discussing the digest generation and comparison processes. Before going into specifics, we define the main terms used, as follows:

feature: Sequence of β bytes (default: 64) extracted from objects;

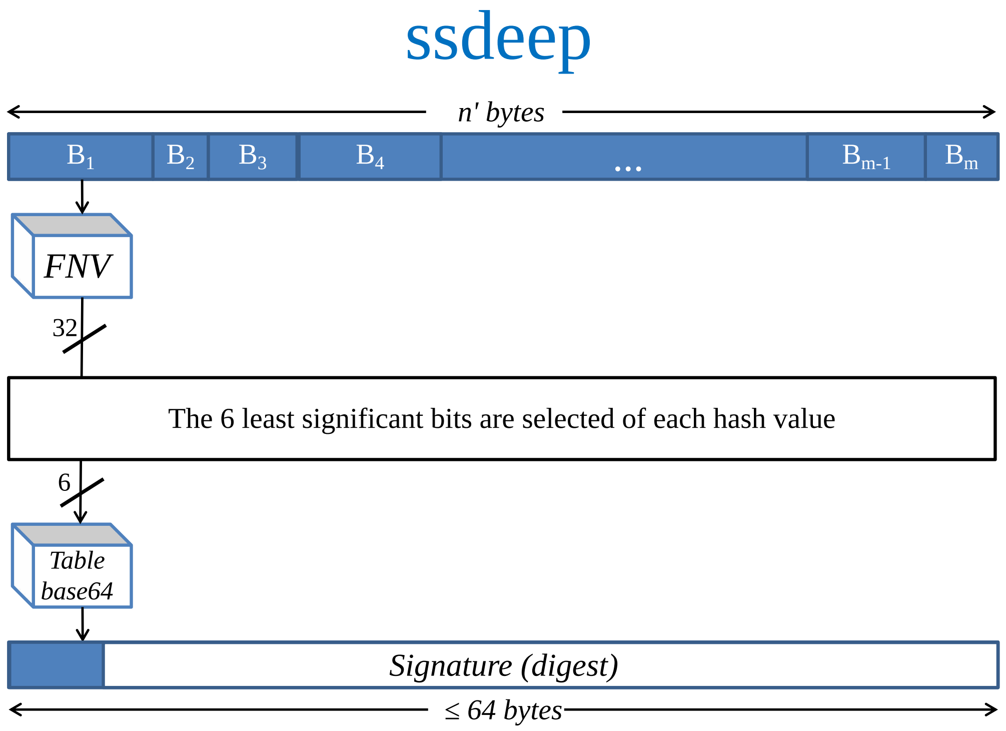


Figure 4 – Creating the object digest from the identified blocks with ssdeep (based on the work of (KORNBLUM, 2006)).

W **(window size):** The size of the sliding window (default: 64) used to select the feature with the smallest entropy in the current window;

m: Bloom Filter size, usually expressed in bits;

k: Number of hash functions used to set bits into the bloom filter (the order of the filter);

f_{max}: Maximum number of features inserted into a single bloom filter;

t: Threshold value (default: 16) used by sdhash to select which feature will be part of the digest;

R_{prec}: Precedence rank with the entropy value of each feature, proportional to the probability of encountering a feature;

R_{pop}: Popularity rank of each feature, representing the number of times a feature had the lowest entropy value in the selection feature process.

2.4.5.4.1 Digest generation process

The sdhash digest generation process can be divided into four steps depicted in Fig. 5, named: Feature extraction, filtering, selection, and encoding. In the following, we provide details of each step based on sdhash proposal (ROUSSEV, 2010).

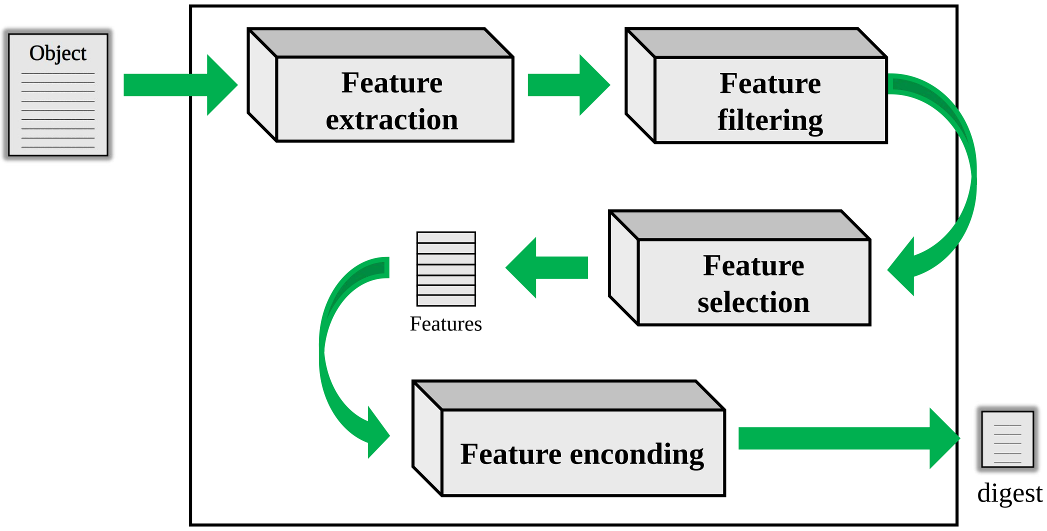


Figure 5 – sdhash digest generation process.

Feature extraction process The first step of the digest generation process is feature extraction. A set of features is extracted from a given object by a fixed-size sliding window (of the same size as a feature) that moves byte-by-byte through the whole object. After the extraction, all features have their Shannon entropy score (H) computed by Eq. 2.1; each entropy value has a precedence rank (R_{prec}) associated with it proportional to the probability that it will be encountered.

$$H = -\sum_{i=0}^{j} P(X_i) \cdot log_2(P(X_i)) \tag{2.1}$$

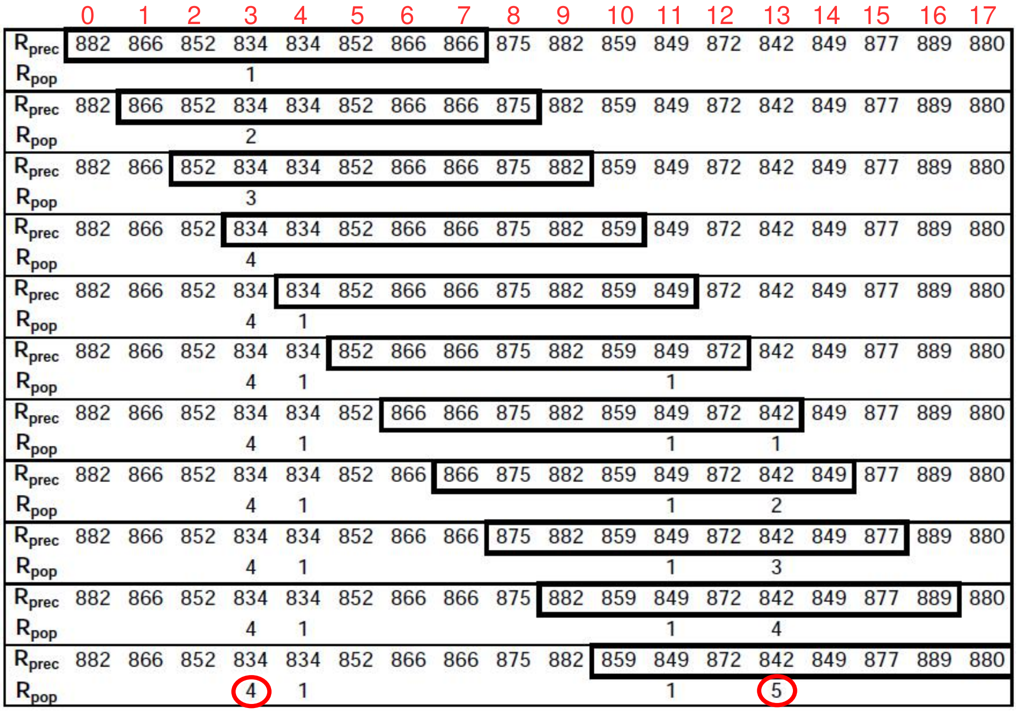
Here, $P(X_i)$ is the empirical probability of encountering byte X_i in the feature and j is the number of different bytes that compose the feature. H is scaled in the range 0 to 1000 (integer), according to Eq. 2.2.

$$H_{norm} = \lfloor 1000 \cdot H / log_2 W \rfloor \tag{2.2}$$

Note that the H calculus does not take into consideration all the 256 possible combinations of a byte, but only those related to the values present in a particular feature; this way, a byte that occurs only one time in a feature has $P(X_i)$ = 1/β (for $1 \leq j \leq \beta \leq 256$).

Feature filtering process Features considered weak are not taken into consideration. Roussev, V. (ROUSSEV, 2010) argues that this filtering helps in reducing the false positives rates and has minimal impact on the object coverage. All features with $H_{norm} \leq 100$ or $H_{norm} > 990$ are dropped out from consideration. The first condition ensures that many features with repeated characters blocks (e.g., sequences of the same byte - 0x00, 0xFF) be removed, while the second condition excludes features with near-maximum entropy value (found in tables spread across many objects, such as Huffman and quantization tables in JPEG headers).

Feature selection process After eliminating weak features, sdhash picks the ones that seem unique to represent the object. The sliding window goes through all features (moving feature-by-feature) and increments the corresponding R_{pop} of the leftmost feature in the current context with the smallest entropy value. In the end, all features having $R_{pop} \geq t$ are selected to be part of the object digest. Fig. 6 illustrates a simplified feature selection process. The input object has 18 features, $W = 8$, and $t = 4$. In the end, only two features (f_3 and f_{13}) are chosen to represent the object.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
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						
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		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		2				
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		3				
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		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							1		5				

Figure 6 – Example of the sdhash feature selection process, using $W = 8$ (adapted from (ROUSSEV, 2010)).

Feature encoding process The last step of the digest generation process encodes all selected features into a small representation of the object, the digest. sdhash uses a sequence of bloom filters to store all object features. Fig. 7 illustrates the whole process. Each selected feature is mapped into a filter. More specifically, a feature is hashed (using SHA-1) and the result is split into k parts; the $log_2(m)$ least significant bits of each piece are selected to set bits within the bloom filter. A maximum of f_{max} features is inserted into a single bloom filter, and when the filter reaches its capacity, a new one is created. The final sdhash digest is the concatenation of all bloom filters produced. The default parameters values of sdhash are: $k = 5$, $m = 2048$ bits, and $f_{max} = 160$.

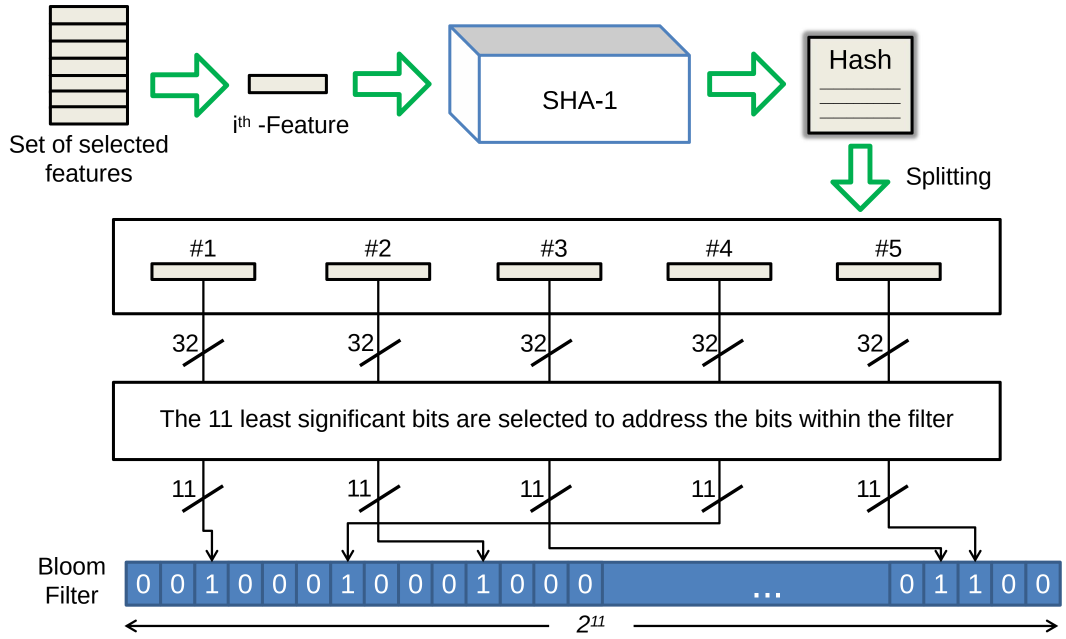


Figure 7 – Encoding feature process of `sdhash`

2.4.5.4.2 Digest comparison process

To assess the similarity of two objects with `sdhash`, we need to create and compare the objects' digests. The first step was introduced in the previous subsection, while the latter will be explained here based on (ROUSSEV, 2010).

The `sdhash` digest is composed by a small header followed by one or more bloom filters (all concatenated). The header keeps the information about the file represented by the digest, such as filename, size, number and type of hash functions used to map features into the digest, number of bloom filters the file has, etc (ROUSSEV; QUATES, 2013). Basically, when assessing the similarity of two objects F_x and F_y, `sdhash` compares the set of bloom filters of both. One filter of F_x is compared to all filters of F_y and the maximum similarity value (number of common bits) is selected; after comparing all filters of F_x to all filters of F_y, an average is produced representing the similarity score. The score ranges from 0 (dissimilar, non-match) to 100 (very similar, not necessarily identical). A value of -1 produced as similarity score is a rare occurrence and it is classified as an unknown result (data with large regions with low entropy data) (ROUSSEV; QUATES, 2013). In the next paragraphs, we explain in detail how digests are compared by `sdhash`.

First, we start by explaining the comparison of two bloom filters and how to come up with a similarity measure. Consider two filters bf_i and bf_j with m bits and k hash functions each. The comparison of these filters takes into consideration the bits in common (set to one) to measure the overlap between the sets they represent (in this case, the features in common). Before we present the formula for such comparison, we need to be aware that bloom filters are probabilistic structures and are prone to false positives. A single bit set in common between two filters could be due to the same element or any other one particular to each set. By using classical bloom filter analysis (BLOOM, 1970), we can estimate (using Eq. 2.3) the number of expected bits in common in bf_i and bf_j ($E_{i,j}$). Consider that each filter contains g_i and g_j elements ($g_i \leq g_j$) and $G(bf_i, bf_j)$ is a functions that returns the number of elements in common between bf_i and bf_j.

$$E_{i,j} = m \cdot (1 - p^{k \cdot g_i} - p^{k \cdot g_j} + p^{k \cdot (g_i + g_j - G(bf_i, bf_j))}) \quad (bits) \tag{2.3}$$

Here, p (defined in Eq. 2.4) is the probability that, after setting a single bit in the filter, a randomly taken bit is still 0 (zero).

$$p = 1 - 1/m \tag{2.4}$$

Based on eq. 2.3, we can estimate the maximum and the minimum number of possible overlapping bits due to chance, using Eq. 2.5 and 2.6. Variables e_i and e_j represent the number of bits set to one in bf_i and bf_j, respectively.

$$e_{max} = min(e_i, e_j) \quad (bits), \tag{2.5}$$

$$e_{min} = m \cdot (1 - p^{k \cdot g_i} - p^{k \cdot g_j} + p^{k \cdot (g_i + g_j)}) \quad (bits) \tag{2.6}$$

Based on e_{max} and e_{min}, Roussev defines a *cutoff* point C (Eq. 2.7). Only if a certain number of bits ($> C$) is found, a similarity score is produced; otherwise, `sdhash` assumes that the common bits of the filters under comparison were due to chance and no similarity is attributed (score = 0).

$$C = \alpha \cdot (e_{max} - e_{min}) + e_{min} \tag{2.7}$$

where α is a variable defined empirically (default: 0.3) (ROUSSEV, 2010).

Given all this background, we can finally compute the similarity score of filters bf_i and bf_j, using Eq. 2.8.

$$SF_{score}(bf_i, bf_j) = \begin{cases} -1, & \text{if } g_i < N_{min} \\ 0, & \text{if } e_{i,j} \leq C \\ \left[100 . \frac{(e_{i,j} - C)}{(e_{max} - C)}\right] & \text{otherwise} \end{cases} \tag{2.8}$$

Here, N_{min} is the minimal number of elements in a bloom filter to compute a meaningful score (default: 16). The -1 value is just a flag to indicate this particular bloom filter comparison does not produce a meaningful result and should not be considered while computing the average value. Parameter $e_{i,j}$ is the number of bits set to one in $bf_i \cap bf_j$.

After comparing all filters, `sdhash` summarizes the results according to Eq. 2.9, where the similarity score is computed for two digests, SD_x and SD_y, produced from objects F_x

and F_y, where each one consists of u and v bloom filters each ($u \leq v$) (ROUSSEV, 2010).

$$SD_{score}(SD_x, SD_y) = \frac{1}{u}\sum_{i=1}^{u} max_{1 \leq j \leq v} SF_{score}(bf_{x,i}\ ,\ bf_{y,j}) \tag{2.9}$$

2.4.5.4.3 Research on sdhash

Extensive research has been conducted over sdhash exposing some vulnerabilities when one attempts to perform changes on objects to avoid blacklists, and also some improvements in mitigating these issues (CHANG *et al.*, 2015; OLIVER *et al.*, 2014; BREITINGER *et al.*, 2012). Furthermore, Roussev, V. (ROUSSEV, 2011) showed that sdhash outperforms ssdeep with respect to its detection capabilities. Besides, sdhash can detect both resemblance and containment similarity.

2.4.5.5 mrsh and mrsh-V2

Multi-Resolution Similarity Hashing-v2 (mrsh-v2) (BREITINGER; BAIER, 2013) is an extension of mrsh (ROUSSEV *et al.*, 2007) regarding the detection capabilities and performance, that combines parts of the digest generation process of the most known AM tools, such as ssdeep and sdhash. The mrsh-v2 uses a rolling hash algorithm over a 7-byte sliding window that moves through the object byte-by-byte. Whenever the hash function hits a specific value (based on a predefined block size parameter b), it identifies a trigger point, denoting the ending and beginning of a block. Fig. 8 depicts the first part of the process, based on *ssdeep* (with the difference in the chosen rolling hash algorithm and block size value). On the second part, based on the sdhash tool, a hash function (FNV-1a) is used to hash each block defined previously; the result is split into k sub-hashes, where the $k \cdot log_2(m)$ bits of each part are used to address bits in a bloom filter (k is the number of sub-hashes and m the bloom filter size), as shown in Fig 9. The final digest is a sequence of bloom filters, with a length of about 0.5% of the input size.

The comparison digest function is the same as the one used by sdhash (Sec.2.4.5.4.2). Although this method is faster than sdhash, its precision and recall rates are worse.

2.4.5.6 TLSH

Proposed by Oliver, J. et al. (OLIVER *et al.*, 2013), TLSH is based on the locality-sensitive hashing (LSH) scheme, producing a fixed-size digest used to find similarities among objects. Given an object as input, TLSH starts processing it using a 5-byte sliding window that moves byte-by-byte, extracts six trigrams in each step (a combination of the window characters), and populates a 128-bit array of counter buckets using a mapping function (Pearson hash (PEARSON, 1990)); next, three quartile points are calculated based on the array. Fig. 10 illustrates this process. The final digest is composed of a header and a body, with 35 bytes of size. The first part (header) has 3 bytes corresponding to the quartile points, the object size,

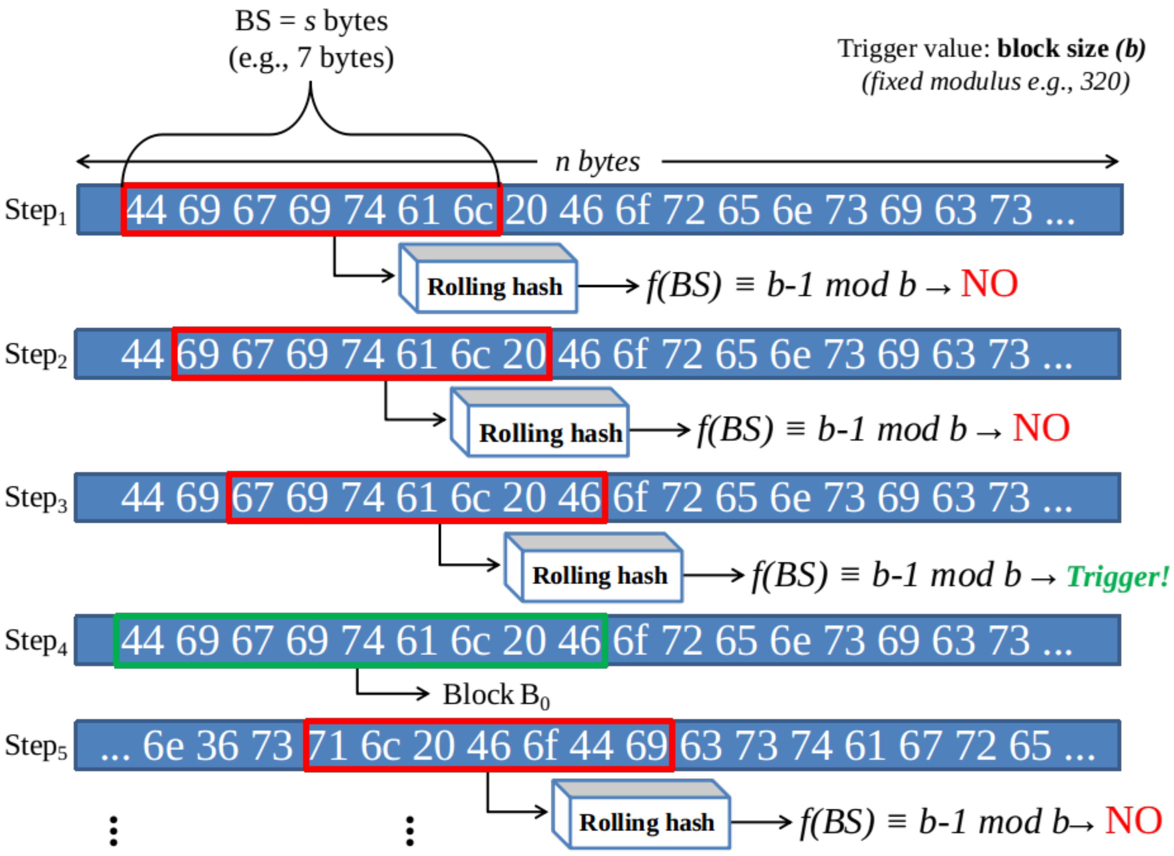

BS = s bytes
(e.g., 7 bytes)
Trigger value: block size (b)
(fixed modulus e.g., 320)
n bytes
Step1
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 73 ...
Rolling hash
f(BS) ≡ b-1 mod b → NO
Step2
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 73 ...
Rolling hash
f(BS) ≡ b-1 mod b → NO
Step3
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 73 ...
Rolling hash
f(BS) ≡ b-1 mod b → Trigger!
Step4
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 73 ...
Block B0
Step5
... 6e 36 73 71 6c 20 46 6f 44 69 63 73 74 61 67 72 65 ...
Rolling hash
f(BS) ≡ b-1 mod b→ NO

Figure 8 – Extracting object blocks with `mrsh-v2` (based on the work of (BREITINGER; BAIER, 2013))

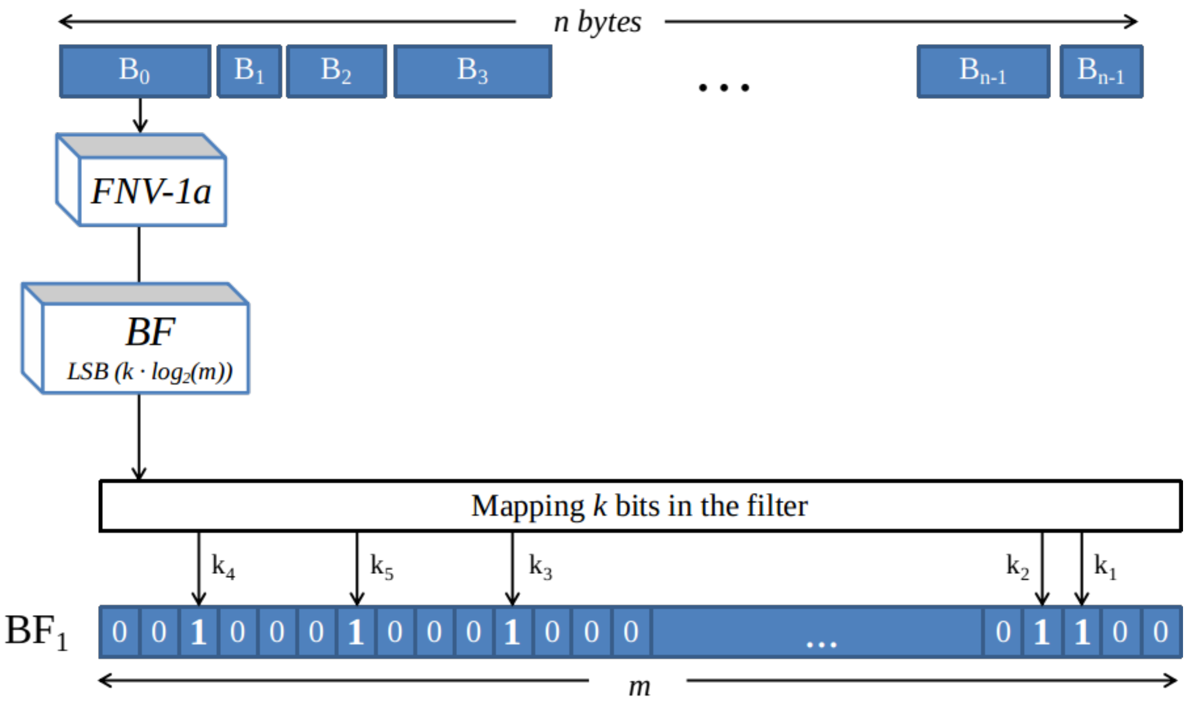


Figure 9 – Hashing and inserting object blocks into the digest with `mrsh-v2` (based on the work of (BREITINGER; BAIER, 2013))

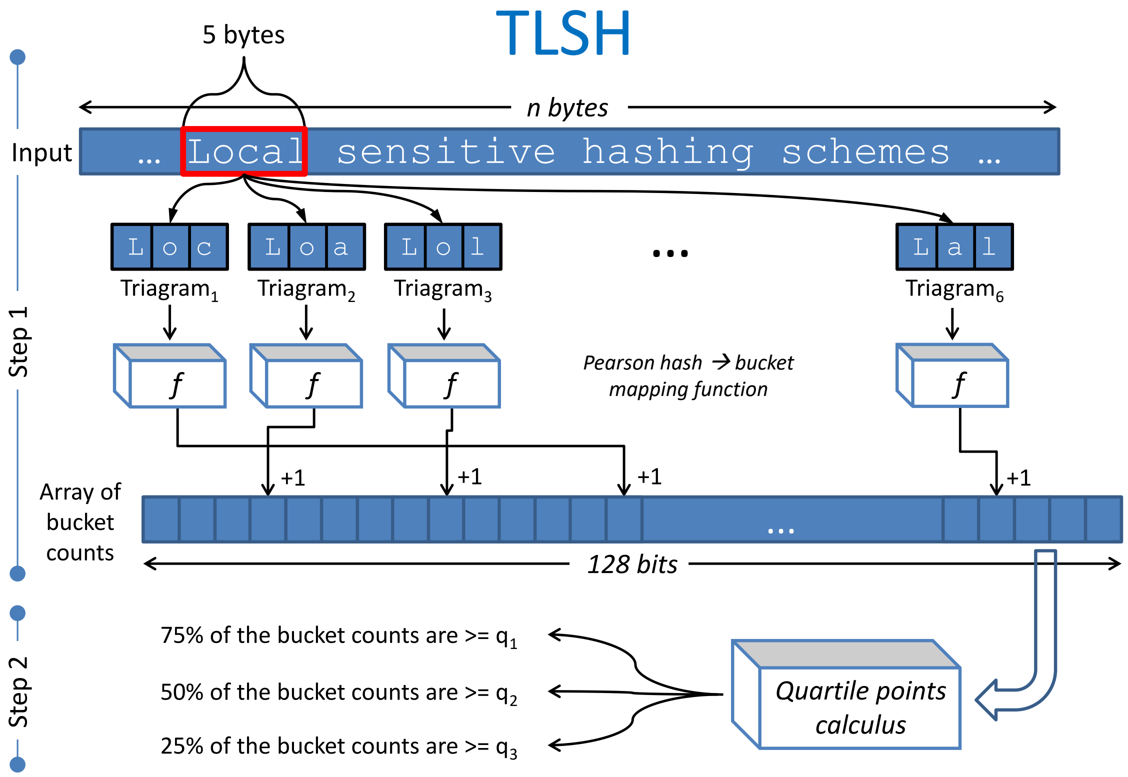

TLSH
5 bytes
n bytes
Input
... Local sensitive hashing schemes ...
L o c
L o a
L o l
L a l
Triagram$_1$
Triagram$_2$
Triagram$_3$
Triagram$_6$
f
Pearson hash → bucket mapping function
+1
Array of bucket counts
...
128 bits
Step 1
Step 2
75% of the bucket counts are >= q$_1$
50% of the bucket counts are >= q$_2$
25% of the bucket counts are >= q$_3$
Quartile points calculus

Figure 10 – Digest generation process of TLSH. Part I: extracting trigrams and populating the bucket array (based on the work of (OLIVER *et al.*, 2013))

and a checksum (one byte per parameter), according to Fig. 11. The second part (body), encompassing 32 bytes as depicted in Fig. 12, is constructed by comparing each position in the counter buckets array to the quartile points, and the result is a bit pair defined according to the quartile the value ranges on.

The comparison function of TLSH is divided into two, each comparing one part of the digest. The first one, *Distance Header*, produces an output based on the object size and quartile points. The second one, *Distance Body*, calculates an approximation of the hamming distance between the digests. The sum of both functions is the final result, scoring 0 (zero) for identical (or nearly identical) objects or more for different ones.

Further research showed that TLSH is more robust to random changes and adversarial manipulations than ssdeep and sdhash (OLIVER *et al.*, 2014). However, this scheme focuses on resemblance detection and does not seem to work well for containment detection. This statement is corroborated by Lee, A. and Atkison, T. (LEE; ATKISON, 2017), showing in their work a small detection capability of TLSH concerning containment similarity.

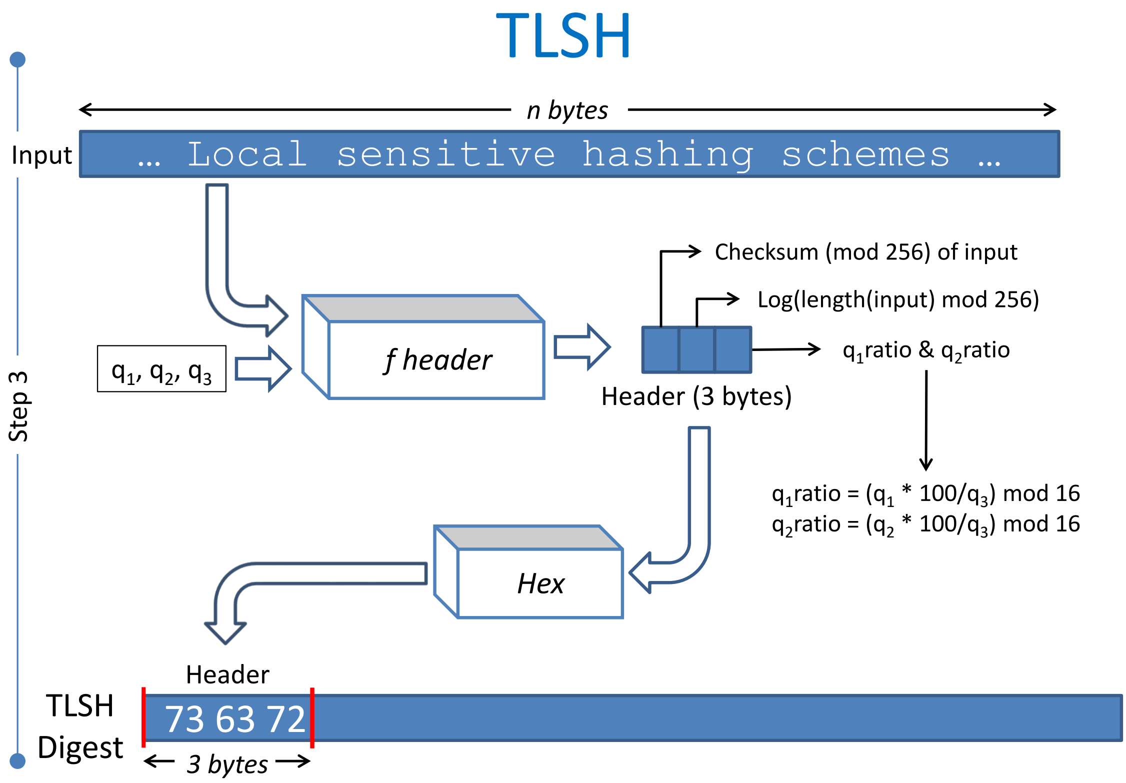

TLSH
n bytes
Input
… Local sensitive hashing schemes …
Checksum (mod 256) of input
Log(length(input) mod 256)
$q_1$ratio & $q_2$ratio
q_1, q_2, q_3
f header
Header (3 bytes)
Step 3
$q_1$ratio = (q_1 * 100/q_3) mod 16
$q_2$ratio = (q_2 * 100/q_3) mod 16
Hex
Header
TLSH
Digest
73 63 72
3 bytes

Figure 11 – Digest generation process of TLSH. Part 2: Creating the digest header (based on the work of (OLIVER *et al.*, 2013)).

2.5 Strategies for similarity digest search

The major bottleneck in digital forensic investigations when performing KFF based on AM is the similarity digest search. An examiner, who usually has a reference list containing objects of interest, needs to compare each object from this set to each one got from the target system under analysis. The goal here is to find similar objects, which can be efficiently done by using one of the AM tools described in the previous section. As stated by Harichandran, V. S. et al. (HARICHANDRAN *et al.*, 2016), this challenge is related to the Nearest Neighbor Search problem, with the difference that we need to identify the similar objects by comparing their digests only.

It is important to mention that this problem is different from finding exact matches, which can be solved efficiently with ordinary databases (WINTER *et al.*, 2013). The similarity search involves finding similar objects sharing a certain degree of commonality (higher than a threshold) only by the comparison of their digests. The objects found this way are separated for a further and deeper analysis (blacklist) or eliminated from the investigation (white list).

Most AM tools perform the similarity search by the naive brute force method: each object from the target system is compared to all objects from the reference list (all-against-all comparison). However, the brute force could be too time-consuming when dealing with large

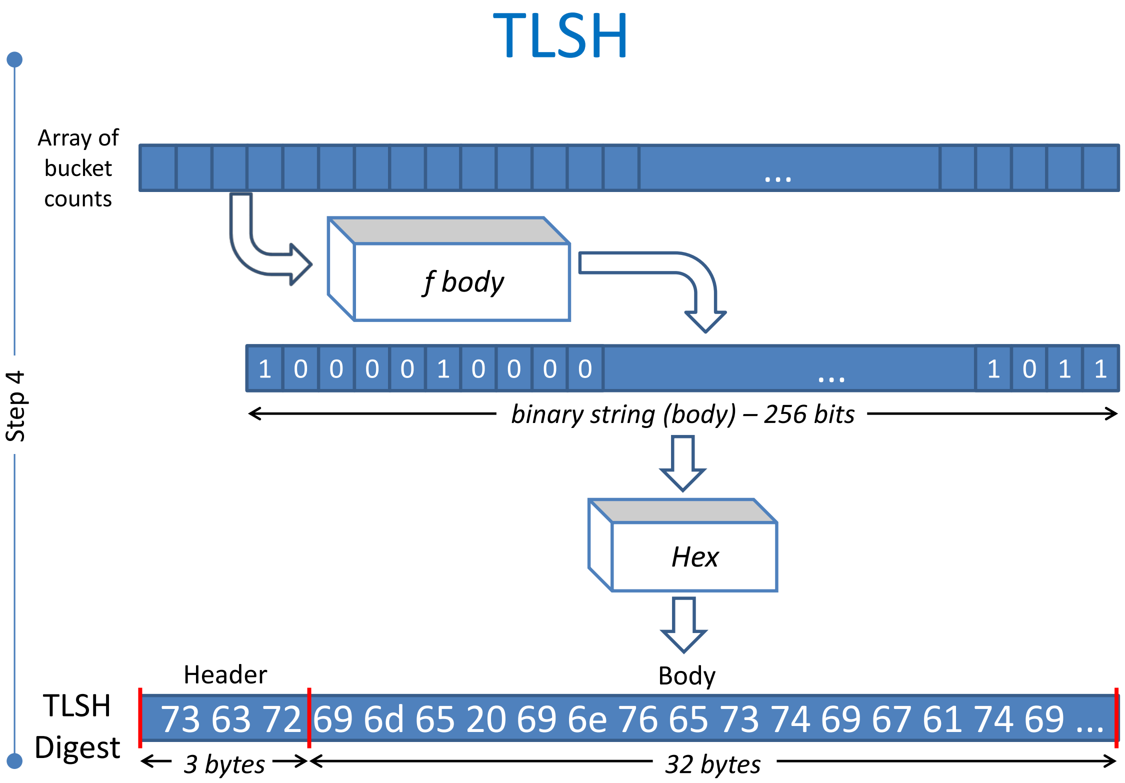


Figure 12 – Digest generation process of TLSH. Part 3: Creating the digest body (based on the work of (OLIVER *et al.*, 2013)).

data sets. The complexity of a search is $O(r \cdot n)$, where r is the number of digests in the reference list and n the number in the target system. On the other hand, traditional approaches aiming at finding exact matches usually create indexes for the objects (using traditional hash functions as SHA-1, SHA-2) of the reference list and store them in sorted lists, balanced trees or hash tables. The complexity of a single query in such cases is $O(log(r) \cdot n)$ for sorted lists and trees and $O(n)$ for hash tables, which is lower than the $O(r \cdot n)$ from brute force similarity search.

To cope with this problem, researchers have proposed techniques aiming to reduce the time involved in the similarity digest search, which encompasses at least the two phases described as follows (CHAWATHE, 2012):

- Preparation phase: The reference list objects (black list, for instance) have their digest created using the chosen AM tool, and then these digests are organized somehow in a structure to improve the lookup procedure;

- Operational phase: The target system has digests created for its objects and, for each one, a comparison is done using the material compiled in the preparation phase. In this phase, the digital forensic examiner verifies the presence or not of the objects of interest in the target system.

In the following subsections, we will present the approaches proposed so far (from the best of our knowledge), which seek to deal with the growing amount of data in forensic investigations by reducing somehow the required time for the search process. For the rest of this section, we will refer to r and n as the number of digests in the reference list and the target system, respectively.

2.5.1 The naive brute force approach

The naive method for pursuing a similarity digest search process is by brute force (all-against-all comparison). Every object in the target system is compared to all objects in the reference list. This approach can be performed with all AM tools presented in section 2.4.5. In the preparation phase, the forensic examiner needs to create digests for the reference list objects using the chosen tool and store them in a file, database or any other structure. In the operational phase, the examiner needs to create a digest for each object of the target system and compare it to all reference list digests, using the comparison function in the AM tool. The best match is the one sharing the highest similarity value with the queried object, and if it is above a predefined threshold, the corresponding object is separated.

The major drawback of this approach is the time complexity, which is $O(r \cdot n)$ or $O(r)$ for a single query. For larger data sets, the search can take days or weeks using common hardware (WINTER *et al.*, 2013).

2.5.2 Distributed P2P search

This strategy aims at performing the search in a distributed way through a peer-to-peer approach. Each node in the network is responsible for managing part of the data in the reference list. Basically, under a request for a search of a given digest, it is calculated in which nodes similar objects may reside based on the distance to the nodes reference digests. The queried digest is then sent to the nodes sharing the higher similarity in order to be compared with the reference data stored on them. The nodes return whether there is or not a similar digest. This method assumes that similar digests will always be distributed to the same nodes.

Although this approach seems interesting due to the distributed processing, which could decrease the time taken in investigations, it presents some drawbacks, as extra storage requirement, a high number of machines to work with, and network delays. The first problem comes when one node leaves the network. Since each node manages one reference point, another one must come up and take this reference point to maintain data availability. However, this is not the ideal solution if it takes longer for a new node to enter the network. A solution would involve storing extra data (redundancy) on each node, in a way that, even though some nodes are gone, data can still be recovered, which would increase even more the storage requirement of the system.

Another problem of the distributed P2P approach is the high number of machines needed to maintain both availability and scalability. Communication delays could also degrade the quality of an investigation process. For this strategy, we will present two approaches: DHTnil and iCTPH. Both approaches do not address the aforementioned problems.

2.5.2.1 DHTnil: Distributed Hash Tables with Nilsimsa

`DHTnil` is an efficient lookup strategy for finding similar digests, based on the `Nilsimsa` AM tool and DHT (Distributed Hash Tables). Its main goal is to identify spam e-mails. According to Jianzhong, Z. et al. (ZHANG *et al.*, 2008), `DHTnil` stores digests in different nodes in a way that digests of similar objects are stored on one of a few nodes. It divides the `Nilsimsa` digest space into some subspaces (with no overlap) managed by the DHT nodes. Space is divided based on a point set, where every point is a core of a subspace (reference point) attributed to a node. Chord was chosen as DHT as well as the Voronoi diagram to divide the multi-dimensioned space, using Euclidean distance to verify similarity. As each subspace is managed by a DHT node, the similarity digest search involves only comparing the digest to the ones stored in a few chosen DHT nodes.

In the preparation phase, `DHTnil` requires that forensic examiners create digests for the reference list objects and generate the reference points. These points are few digests selected from the reference list to represent the subspace. The selection could be randomly or carefully chosen. Next, the remaining digests are stored on the corresponding DHT nodes where the distance from the queried digest and the reference point is smaller.

In the operation phase, a digest is created for the queried object, and the subspace it belongs to is evaluated. The DHT node selected and its neighbors are searched for similarity, where the digest is compared to all other ones stored in the nodes. The number of matches is then returned (ZHANG *et al.*, 2008).

The main drawbacks of `DHTnil` are the already mentioned ones related to distributed P2P systems and also one due to the AM tool chosen (`Nilsimsa`), which suffers from significantly high false positive rates (OLIVER *et al.*, 2013). This approach also performs unnecessary lookups per digest and hence possesses a high time complexity ($O(r)$), equal to the brute force method, although in practice the time is expected to be smaller. This high complexity is due to the number of items in each node being proportional to the number in the set (considering a uniform distribution of the items).

2.5.2.2 iCTPH: Distributed Hash Tables with ssdeep

A similar approach to `DHTnil` is `iCTPH`, which also uses Chord to store and lookup digests, but instead of `Nilsimsa`, it uses the `ssdeep` tool. The iDistance technique (JAGADISH *et al.*, 2005) is used to map similar digests into near clusters. The vector space is divided into clusters, identified by reference points, and the digests are mapped into a cluster according to their distance from the reference point, using the Edit distance algorithm (minimum number of operations required to transform one string into another) (JIANZHONG *et al.*, 2010).

Jianzhong, Z. et al. (JIANZHONG *et al.*, 2010) explains that, in the preparation phase, one must compute the digest of all objects in the reference list, choose a set of points as a reference, and then decide which cluster each digest belongs to. In the operational phase, each queried object has its digest calculated. iCTPH generates a query interval for each cluster related to the queried item and performs a comparison with all digests of the clusters corresponding to that interval. The number of similar digests found is returned.

iCTPH has the same drawbacks of DHTnil: poor AM tool (ssdeep) (ROUSSEV, 2011) and some unnecessary lookups per digest, resulting in a time complexity equal to the brute force: $O(r)$ (single query). Just like DHTnil, this approach also inherits the limitations of a P2P system.

2.5.3 Indexing strategy

Winter, C. et al. (WINTER *et al.*, 2013) present a different approach for similarity digest search, called Fast Forensic Similarity Search (F2S2). The authors use an indexing strategy based on ssdeep (but not restricted to it) to avoid the overwhelming amount of time required by the naive brute force method. It builds an index structure over the n-grams (n consecutive bytes) contained in a digest. All digests with the same n-gram queried are returned in a lookup procedure. They are suitable candidates for being similar to the queried item, and the comparison is restricted to these candidates only. Here we are interested in finding exact matches on the n-grams level. The results presented in the paper points out an impressive speedup compared to brute force method.

The index structure chosen was a particular kind of hash table, containing two parts: A central array (index table) and variable size buckets that can store multiple entries each (n-grams). The n-grams of a digest b with l bytes are: $b_1...b_n$, $b_2...b_{n+1}$, ..., $b_{l-n+1}...b_l$. They serve as lookup key and provide a link to all digests containing the same values. N-grams are composed by two parts. The first one is used as the entry in an address function, responsible for mapping keys (n-grams) to positions in the index table, using the k leading bits of the n-gram (since a digest of ssdeep is base64 encoded, it is necessary to decode it before selecting the bits). The other part, called e-key, is used to identify the n-gram and it is part of the bucket entry, as well as the ID, a link to the corresponding digest (WINTER *et al.*, 2013).

In the preparation phase of F2S2, digests are created for all reference list objects using ssdeep and an ID is assigned to each one. Then, an index table is created, and the digests are inserted on it. However, they are not added directly. A sliding window goes byte-by-byte mapping each n-gram and ID to a position in the index table, inserting it in a new bucket or adding it to an existing one (as long as they share the same n-gram but have different IDs), as shown in Fig. 13.

In the operational phase, the first step is loading the index structure into the main memory. Then, digests are created for each item in the target system, and their n-grams are extracted. A lookup procedure using these n-grams selects all digests (candidates) in the index containing the same n-gram queried. Finally, the `ssdeep` comparison function is executed for the queried digest and each candidate to confirm the similarity. We highlight that since `ssdeep` digests have two signatures for each object (one using block size b and another $2b$), the lookup procedure is done for both.

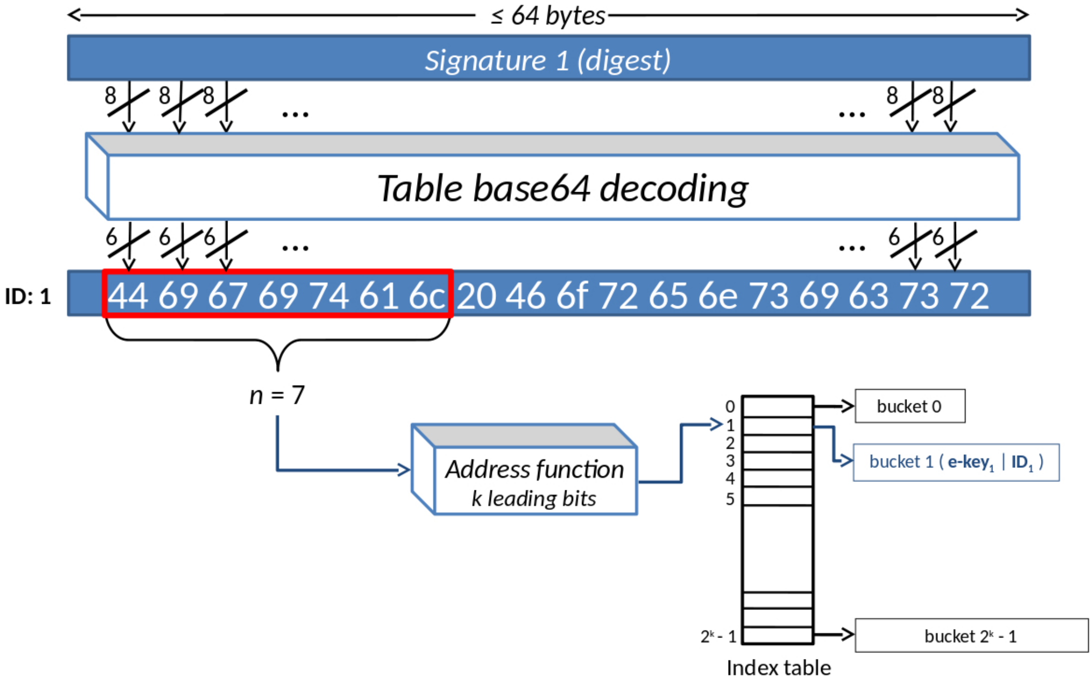


Figure 13 – Inserting a n-gram in the index table in Winter's method (based on the work of (WINTER *et al.*, 2013)).

The main drawback of `F2S2` is the chosen AM tool; `ssdeep` is less accurate than others, especially when comparing objects of different sizes (ROUSSEV, 2011). The proposed strategy does not work with more precise tools, such as `sdhash`, since it does not support digests represented by Bloom filters, which cannot be ordered/indexed. Also, since the number of candidates sharing the same n-grams as query digest is proportional to the number of entries in the index, the time complexity of `F2S2` is the same as brute force, $O(r)$, even though experiments have shown a speedup factor above 2000 compared to brute force (WINTER *et al.*, 2013).

2.5.4 Bloom filter-based search

2.5.4.1 The MRSH-NET strategy

Breitinger, F. et al. (BREITINGER *et al.*, 2014a) present a new similarity digest search strategy based on bloom filters, which reduces the lookup complexity of a single query from $O(r)$ to $O(1)$. The `MRSH-NET` approach is intended to work with `sdhash` and `mrsh-v2` AM tools and uses a single, huge bloom filter to represent all the objects of a reference list. However, due to the characteristics of bloom filters, the method is restricted to membership queries only: Does this set contain any similar object to the queried one? If so, an affirmative answer is returned, but it does not point the similar object(s).

`MRSH-NET` uses `sdhash/mrsh-v2` to extract features from the reference list objects and insert them into a single bloom filter instead of having one or multiple filters per object. This procedure aims to avoid the expensive brute force approach and hence speedup the similarity digest search process. To decide whether or not an object is inserted in the filter, the match decision is based on a sufficiently large number of succeeding features that needs to be found in the filter so the queried object is considered part of the set.

In the preparation phase, one has to extract the features from all objects in the reference list, create a single, huge bloom filter, and then insert the features into the filter.

The operational phase involves loading the structure into the main memory, extracting the features of the queried object, and checking in the bloom filter for their presence. If the object has more than a predefined threshold of succeeding features found in the filter, the object is said to be part of the set and can be separated; otherwise, it is discarded, and the lookup procedure moves to another object.

One of the main drawbacks of `MRSH-NET` is only answering membership queries and not pointing out the similar objects. Also, the filter has to fit into the main memory due to efficiency reasons, another possible problem when the reference list set increases. Removing elements from the structure and inserting too many new objects are both problematic since the false positive rates can increase.

2.5.4.2 Bloom filter-based tree structure

As a form of mitigating `MRSH-NET` main limitation (answering only whether an object is present in a set or not), Breitinger, F. et al. (BREITINGER *et al.*, 2014c; LILLIS *et al.*, 2017) propose a new similarity digest search strategy based on the well-known divide and conquer paradigm. In the new strategy, defined as `HBFT`, the authors build a bloom filter-based tree data structure to store digests and efficiently locate similar objects. The time complexity of a single lookup is $O(log_x(r))$, where x is the degree of the tree. Even though the complexity is higher than the previous method (`MRSH-NET` - $(O(1))$), `HBFT` can return the actual matching object(s).

The basic idea of the HBFT approach is to recursively divide a given set S of similarity digests into X subsets. First, each object has its features extracted (e.g., by the sdhash tool) and inserted into the root node of the tree, a huge bloom filter. Then, S is divided into X subsets containing n/X elements, a child node of the root node is created for each subset and the objects inserted in each corresponding new filter. This procedure is applied recursively. Finally, an FI (File Identifier) is created in the leaf (a link to a database containing the digest of the related bloom filter) as well as an FIC (File Identifier Counter), initially set to zero and incremented in a lookup procedure when FI is reached. An example of the construction of a Bloom filter-based tree is illustrated in Fig. 14.

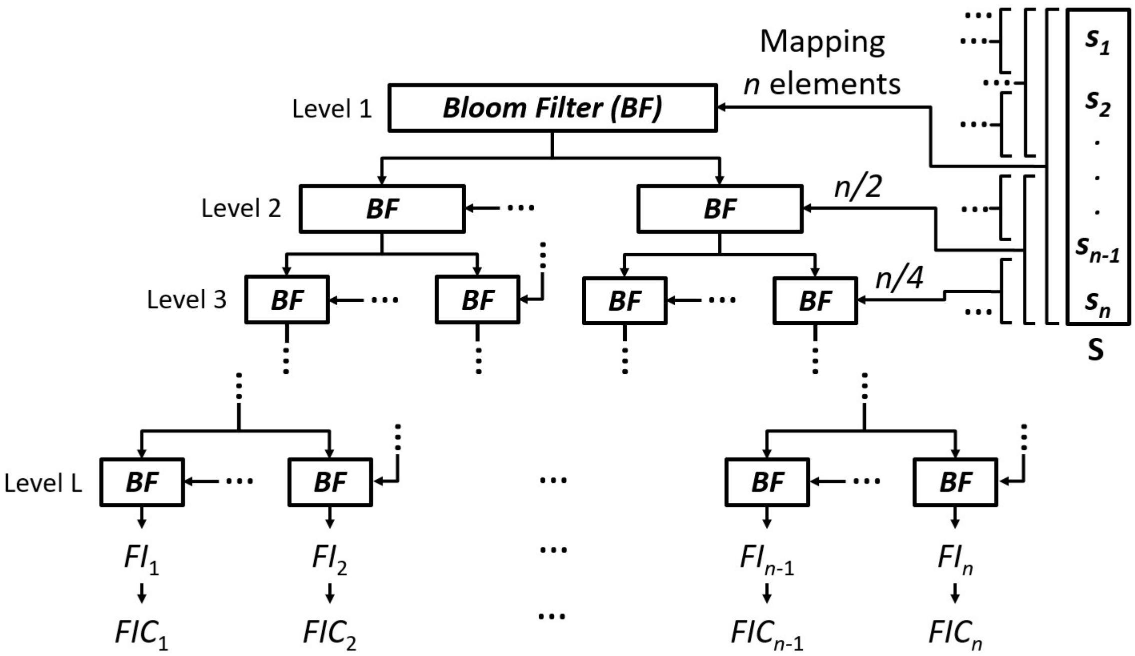


Figure 14 – Bloom filter-based tree construction (Binary tree). Adapted from (BREITINGER *et al.*, 2014c).

One of the main advantages of the scheme is the lookup operation. It is not necessary to compare a digest of a target system against all reference list digests but only to a subset of nodes in the tree structure. Also, as most comparisons will yield a non-match for blacklisting cases, the search starts and ends in the root node. The tree is only traced down to a leaf if a match is found in the root, which means that the queried object (feature) is present in the reference list, and now we only have to determine which object it belongs to, by tracing down the tree and locating the corresponding FI. The match decision on whether an object is inserted in the tree data structure or not is based on a threshold, representing the number of following features required to be found in the tree. Every time we identify a leaf containing the features queried, we increase FIC. In the end, the highest FIC is compared to the threshold, and if its value is equal or higher, we can say that the object is present in the set and take the corresponding FI to reach it. Once we have found the candidate similar to the queried object, we might perform the conventional comparison using the approximate matching tool chosen.

The preparation phase of HBFT consists in extracting the features of all objects in the reference list, create the bloom filter-based tree structure and insert all features on it, including the corresponding metadata (FI and FIC).

The first step of the operational phase is to load the bloom filter tree into the main memory. Then, for each queried object, we need to extract its features and check the filter for their presence. In the end, the object with the highest FIC is returned, and the similarity may be confirmed by using the conventional comparison function of the chosen AM tool (sdhash/mrsh-v2).

The main problem of HBFT is the huge amount of memory required for its operation since several large bloom filters are created to store all data from the reference list.

2.5.5 Cuckoo filter: A new alternative for bloom filters

The MRSH-CF is a new strategy for the similarity digest search problem, presented as an alternative for the MRSH-NET approach, where Cuckoo filters are used to mitigate some of the limitations of bloom filters. The same sdhash and mrsh-v2 can be used to work with this strategy (GUPTA; BREITINGER, 2015).

Cuckoo filter (FAN *et al.*, 2014), a modification of Pagh et al. (PAGH; RODLER, 2004) cuckoo hashing, is a minimized hash table for performing membership queries; cuckoo hashing is used to resolve collisions when inserting elements into the structure (different items inserted into the same hash table position). Data is transformed in a fingerprint before being inserted into the structure, where it is stored into buckets. The filter has an array of buckets of size b, referring to the number of fingerprints that can be stored on it. Also, there is a load factor (α) describing the usage percentage of the filter to decide if the filter needs to be resized.

A Cuckoo filter is composed by a hash table and three hash functions. Each key (entry value) is hashed by two of the hash functions (H_1 and H_2), responsible for assigning the key to buckets in the table. The bucket corresponding to the first hash is tried and checked for an empty space. The key is placed into this bucket in case it is empty. Otherwise, the key is stored in another bucket, corresponding to the second hash value. In case there is already a key stored on the second bucket, the stored key is moved to its second bucket option, and the process repeats until all keys are allocated. If a cycle happens (the same bucket is visited twice), it means that the table is not big enough and needs to be resized, or the hash function needs to be replaced. The third hash function (H_3) is used to store the key in the structure in a compressed form, hashing it and using only f bits as the tag size (FAN *et al.*, 2014).

According to Fan, B. et al. (FAN *et al.*, 2014), the main benefits achieved with this structure compared to bloom filters, is the support for adding and removing items dynamically, better lookup performance, and less space requirement for some applications (related to some false positive rates). Although insertion operations are more complex due to possible keys real-locations, its time complexity is the same one as for bloom filters ($O(1)$). Deletion and lookup complexity are also $O(1)$.

To allow similarity identification, MRSH-CF stores only object features in the filter instead of the whole object. Based on MRSH-NET, this new Cuckoo filter-based strategy considers an object match when a specific number of features are found in the structure.

The preparation phase of this strategy requires the extraction of the features from the reference list objects using the chosen AM tool. Then, the Cuckoo filter structure is created, and all features are inserted on it.

In the operational phase, the first step is to load the structure into the main memory. The queried object has its features extracted and checked within the filter. If a number equal or higher than a predefined threshold of features is found, the queried object is said to be part of the set; otherwise, it is put away, and the lookup procedure continues with the next object.

The main limitations of MRSH-CF are the same ones of MRSH-NET: membership queries and high memory consumption. The Cuckoo filter strategy only gives binary answers: the object belongs or not to the set. It does not point out which is the similar object, which could be enough for some problems, but for KFF it represents a limitation.

2.5.6 Other strategies

There are other similarity digest search strategies not addressed in this thesis for a particular reason or because we are not aware of. A particular one is proposed by Chawathe, S. S. (CHAWATHE, 2009; CHAWATHE, 2012) which it is based on a Locality Sensitive Hashing (LSH) method. However, due to the lack of data presented in the papers, we choose not to include it in our analysis.

2.6 Conclusions

In this chapter, we discussed one of the main problems of digital forensic investigations: The efficient identification of similar objects. To perform such task, we pointed out the AM functions as an alternative and presented some background, where one can efficiently perform black/white-listing using them. We also showed some tools proposed in the field and presented a brief description of their main characteristics. Furthermore, for dealing with large data sets, we indicated that AM can be used in the form of the similarity digest search strategies, which can reduce significantly the time taken in forensic investigations. In the next chapters, we present some limitations of current AM solutions and some contributions of this thesis with respect to this field.

3 Comparative analysis of the Similarity Digest Search Strategies

To cope with the excessive amount of time required by digital forensic investigations when dealing with large data sets, new strategies have been proposed in the literature to perform queries more efficiently and avoid the expensive brute force approach. In this chapter, we present a detailed comparison of the Similarity Digest Search Strategies (SDSS), first introduced by Zhang, J. et al. (ZHANG *et al.*, 2008). Our comparison involves several aspects, as time complexity, memory requirement, search precision, among others. We pointed out their strengths and weakness, and show that even though some strategies outperform others in some aspects, they fail in others. There is no currently suitable approach that satisfies the most relevant requirements.

The second part of this chapter presents an evaluation of the operational costs of some SDSS and show how they would behave in practical scenarios. We conclude by showing the significant improvements of the strategies in comparison to the corresponding brute force approach, indicating the impact of more clever ways when dealing with large data sets. For the rest of this chapter, we consider a scenario where digital forensic practitioners have a data set consisting of many objects (black or white list), referred to as the reference list. This set will be contrasted to the devices under investigation (referred to as the target system) to identify similar content, which can be excluded from/separated for analysis, depending on the investigation goal. We aim at comparing the SDSS when performing such a process.

From the best of our knowledge, this is the first work that compares current SDSS present in the literature, showing their strengths and weakness from many aspects and also how they would perform in practical scenarios.

3.1 Part I: Characteristics and theoretical evaluation of the SDSS

In this section, we present a comparison of all strategies for similarity digest search discussed previously in related work. We first present the strategies and some of their characteristics and then present a theoretical evaluation assessing the time and space required by the approaches, along with other topics such as false positive rates and detection capability.

We highlight that for the `MRSH-NET`, `HBFT`, and `MRSH-CF` strategies, we have considered `sdhash` as the AM tool since it presents better detection capabilities, although they could also work with `mrsh-v2` tool.

3.1.1 Characteristics

One of our analysis of the SDSS involves comparing the approaches through their main characteristics. Table 1 presents our results followed by a discussion of the evaluated aspects and their importance for the strategies' performance.

Table 1 – Similarity digest search strategies - Characteristics

Strategy	**Tools**	**Main technology**	**Input**	**Output threshold (t)**	**Match decision**	**Insert/ remove elements**	**Database use**
Brute force (sdhash)	sdhash	Bloom filters	sdhash digest	Digest $\geq t$	Bloom filter comparison	✓ / ✓	X
Brute force (ssdeep)	ssdeep	Rolling Hash	ssdeep digest	Digest $\geq t$	Edit distance	✓ / ✓	X
Brute force (TLSH)	TLSH	LSH	TLSH digest	Digest $\geq t$	Header/ body distance	✓ / ✓	X
DHTnil	Nilsimsa	DHT + Voronoi Diagram	Bit vector	Number of matches $\geq t$	Adapted euclidean distance	✓ / ✓	X
iCTPH	ssdeep	DHT + iDistance	ssdeep digest	Number of matches $\geq t$	Edit distance	✓ / ✓	X
F2S2	ssdeep	Indexing (n-grams) + hash table	ssdeep digest	Candidates sharing the same n-gram queried	Edit distance	✓* / ✓	✓
MRSH-NET	sdhash, mrsh-v2	Single, huge Bloom filter	Object features	YES / NO (Consecutive features found in the filter $\geq t$)	Bloom filter matches	X / X	✓
HBFT	sdhash, mrsh-v2	Bloom filter tree structure	Object features	Candidate with highest number of features found in the filter $\geq t$	Bloom filter matches	X / X	✓
MRSH-CF	sdhash, mrsh-v2	Cuckoo filter	Object features	YES / NO (Consecutive features found in the filter $\geq t$)	Cuckoo filter matches	X / ✓	✓

* A data set increase (beyond its real capacity) is allowed at the cost of performance.

3.1.1.1 Supported AM tools and technology

Some strategies can be used with any AM tool while others are restricted to a specific one. In the latter case, we may have to use a tool with low accuracy depending on the strategy. In one hand, this would decrease the time to perform a similarity digest search, but on the other hand, it would increase the process of manual inspection of the results due to the high number of false positive matches. In the worst case, some tools could miss relevant data in the search. Another point for consideration: Depending on the chosen technology to perform the search, we need more computational power than we have. The strategies iCPTH and DHTnil, for example, require several machines working together to perform the search efficiently, which may not be the case for some forensic investigations, and therefore, these methods would not be appropriate.

3.1.1.2 Strategy input/output

Each approach aiming at reducing the time for the similarity digest search requires a different input format. Some strategies require only the digest created by the supported similarity tool while others, a pre-computed set of values generated by a sub-process of the tool. The naive brute force method requires as input only the digest created by the chosen tool, while other strategies receive as input the features extracted from objects, like MRSH-NET, HBFT, or MRSH-CF, which take only the result of the intermediary step of the digest generation process of sdhash/mrsh-v2 tools as input.

The outcome of some strategies is a list of possible candidates to similarity found above some threshold value (e.g. brute force and F2S2) or the most similar object (HBFT); a next step requires the inspection from the practitioner to confirm the similarity and/or eliminate candidates, either by using the corresponding AM tool or by manually looking at the matches. Other tools (e.g. DHTnil and iCPTH) return the number of similar matches found or a binary answer indicating the presence or not of the queried object in the set (MRSH-NET and MRSH-CF); although these last cases are too restricted, they could be sufficient in a blacklisting case to separate the corresponding media for further and deeper analysis.

3.1.1.3 Strategies' match decision

The decision of whether two objects are similar (match) or not usually incorporates the characteristics of the AM tools supported by the strategy and take advantage of their structure. F2S2 creates n-grams of ssdeep digests based on the assumption that the tool encodes each *feature* extracted from the object in one byte in the digest. This way, similar objects will have similar features and hence similar bytes in the digests; the strategy tries to identify similarity by indexing and later comparing small parts (*n* consecutive bytes) of the digests. MRSH-NET, HBFT, and MRSH-CF take the features extracted from sdhash/mrsh-v2 and insert them into a single, huge bloom filter. The match decision in such cases is the number of following features found in the structures. The choice of the AM tool and match decision can bring some undesired side effects to the SDSS, such as false positives. F2S2 does not create false positives because its match decision method does not decide whether two digests are similar or not (WINTER *et al.*, 2013); it relies on ssdeep' approach to perform comparisons. However, MRSH-NET, HBFT, and MRSH-CF approaches do, according to the explanation presented next.

According to Breitinger, F. et al. (BREITINGER *et al.*, 2014c), the false positive probability for a match between two different objects in HBFT (which also applies to MRSH-NET and MRSH-CF) is calculated by $p_f = p^r$, where p is the false positive probability of a single feature and r the number of following features required to be found in the filter for two objects being considered as similar. While r can be adjusted according to the desired false positive rate, p is defined by: $p \approx (1 - e^{-kz/m})^k$, where k is the number of independent hash functions (filter order), z the number of features inserted into the bloom filter and m the filter size (BREITINGER *et al.*, 2014c). This way, the strategy decides when a match is found by the number of features present in the filter, and based on it, it can create more or less false positives depending on the parameters used by the strategy.

On the other hand, we have strategies in which a match decision does not depend on the strategy itself, but on the tool under consideration. The chosen comparison method is the tool's comparison function, and it is responsible for establishing when a match is found or not. The most simple strategy (brute force) is an example of such a case. When using the `sdhash` AM tool, the method is the comparison of bloom filters. Using `ssdeep`, it changes to the edit distance, while for `TLSH`, it uses a distance header/body (Hamming distance approximation) function. Changing the tool does not interfere in the strategy itself, but only in the comparison function, which is tool-related.

3.1.1.4 Insert/remove elements and use of an own database

With respect to the insertion of new elements, some strategies allow the increase of the database size dynamically, while others need to be constructed considering a predefined maximum number of objects that the strategy will store. Brute force, `DHTnil`, `iCPTH`, and `F2S2` are examples that allow the data set to increase dynamically without requiring the re-creation of the database. `F2S2`, which uses a chaining hash table, is a particular case in which the insertion beyond its real capacity is allowed at the cost of performance degradation (linear cost as the table fills). Other strategies, such as `MRSH-NET` and `HBFT` rely on technologies (Bloom filters) that require the knowledge about the maximum number of elements beforehand to adjust some parameters, like false positives rates, for instance. Although it is possible to insert as many elements into a bloom filter as we want, its false positive rates will increase and degrades the search quality, compromising the results of the strategy. In such cases, a new data structure will have to be created and adjusted to the new number of elements. `MRSH-CF` is another case that needs the maximum number of objects before the structure creation since the hash table and buckets have a fixed and predefined size. However, this structure is more robust than bloom filters as it can store multiple elements in each bucket without altering the false positives rates significantly.

Removal operations are also possible for most strategies, except for the ones based on bloom filters, in which, once we insert several elements on it, we can not distinguish the bits set in the filter by one particular object. This way, removing elements is not possible.

Another point for consideration is that some strategies have their own technology to store the similarity data, as `F2S2` (hash table), `MRSH-NET` and `HBFT` (Bloom filters), and `MRSH-CF` (Cuckoo filter). Others, like the brute force, `DHTnil`, and `iCPTH` require an external database to store the digests. These strategies may use whatever storage technology the practitioner wants: Ordinary databases, files, xml, and others. Although the latter methods scale better to data set increasing, they may have their efficiency degraded depending on the chosen technology, causing some extra delays.

3.1.2 Evaluation

We also evaluate the approaches concerning time and space requirements, false positives rates and detection capability. We present in this section a discussion of these aspects and summarizes ours results in Tab. 2. The formulas used in each case are available in Appendix A.

Table 2 – Similarity digest search strategies - Performance assessment of different properties

Strategy	**Memory requirements for data sets of:**				**Single lookup complexity**	**False positives**	**Resemblance/ containment detection**
	1 GiB	**10 GiB**	**100 GiB**	**1 TiB**			
Brute force (`sdhash`)	25.60 MiB (2.50%)	256.00 MiB (2.50%)	2.50 GiB (2.50%)	25.60 GiB (2.50%)	$O(n)$	*No*	✓ / ✓
Brute force (`ssdeep`)	0.19 MiB (0.02%)	1.87 MiB (0.02%)	18.75 MiB (0.02%)	192.00 MiB (0.02%)	$O(n)$	*No*	✓ / *X*
Brute force (`TLSH`)	0.07 MiB (0.01%)	0.68 MiB (0.01%)	6.84 MiB (0.01%)	70.00 MiB (0.01%)	$O(n)$	*No*	✓ / *X*
`DHTnil`	32.49 MiB (3.17%)	33.05 MiB (0.32%)	38.68 MiB (0.04%)	96.43 MiB (0.01%)	$O(n)$	*No*	✓ / *X*
`iCTPH`	96.62 MiB (9.44%)	98.30 MiB (0.96%)	115.18 MiB (0.11%)	288.43 MiB (0.03%)	$O(n)$	*No*	✓ / *X*
`F2S2`	1.71 MiB (0.17%)	17.07 MiB (0.17%)	170.70 MiB (0.17%)	1.71 GiB (0.17%)	$O(n)$	*No*	✓ / *X*
`MRSH-NET`	16.00 MiB (1.56%)	128.00 MiB (1.25%)	1.00 GiB (1.00%)	16.00 GiB (1.56%)	$O(1)$	**Yes**	✓ / ✓
`HBFT`	176.00 MiB (17.19%)	1.79 GiB (17.90%)	17.64 GiB (17.64%)	336.00 GiB (32.81%)	$O(log(n))$	**Yes**	✓ / ✓
`MRSH-CF`	14.00 MiB (1.37%)	140.00 MiB (1.37%)	1.37 GiB (1.37%)	14.00 GiB (1.37%)	$O(1)$	**Yes**	✓ / ✓

3.1.2.1 Memory requirements

We evaluated all SDSS related to the amount of memory required for different data set sizes, varying from 1 GiB to 1 TiB. We emphasize that the memory evaluated here is not the storage one, but the working memory (e.g., RAM) of devices, which allows fast similarity queries using SDSS structures. Our results are shown in Table 2, describing the amount required (MiB or GiB) and the compression rate for each strategy. The details of our calculations are presented in Appendix A.

We highlight that some strategies, as `MRSH-NET` and `HBFT`, have their structure size adjusted for practical reasons, since they are based on bloom filters; the size has to be a power of two (2^c, for $c \in \mathbb{N}$). For this reason, when calculating the filter size and getting a result of $2^{31.27}$ bits, for instance, we need to adjust the size for 2^{32}. Although this modification can almost double the size of the filter in some cases, it is necessary for practical implementations. Other strategies are less effected by this issue, as `MRSH-CF`, which also needs some adjustments in the tag size. After defining the size for the object representation in the filter according to the false positive rate and number of entries in each bucket, we may obtain a decimal number. In such case, we choose the next integer in order to not increase the false positive rates. The result is an increase in the structure size, but not as significant as it happens with the bloom filter approaches. The brute force, `DHTnil`, `iCTPH`, and `F2S2` strategies do not suffer from this issue.

The first point to mention about our analysis is the memory growth rate of some strategies, as shown in Fig. 15. While some of them have a linear behavior (brute force, `F2S2`, `MRSH-NET`, `HBFT` and, `MRSH-CF`), others do not (`DHTnil` and `iCTPH`). The latter group presents some specific costs which do not scale linearly since they have minimum setting costs necessary for operation, as the need for storing the Chord finger table and reference points list in each node (which in our case are kept fixed for all reference list sizes). These two values are counted in the final memory requirement and, as the digests of both approaches have a short length, this setting cost stands out for small data sets.

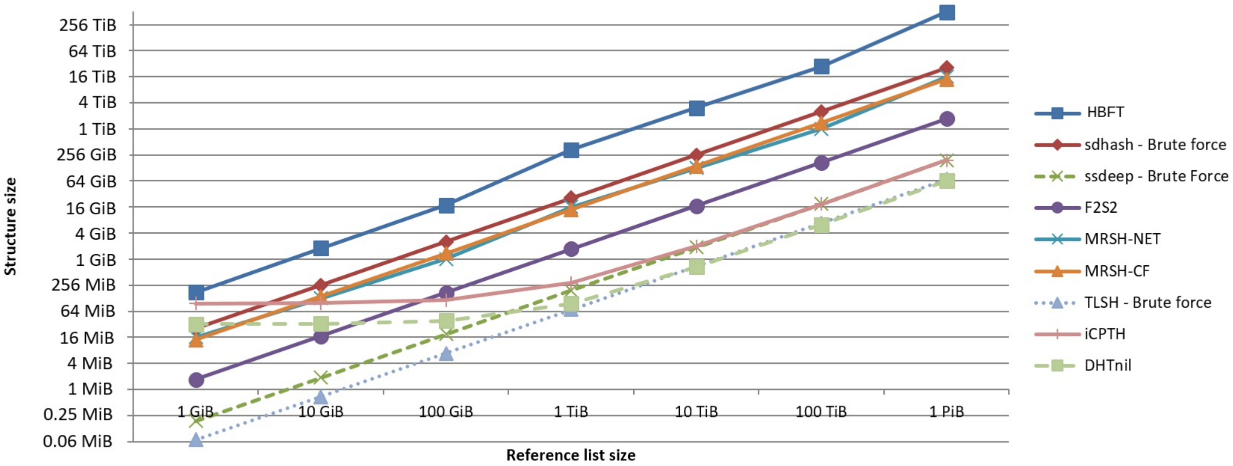


Figure 15 – Memory requirements: Strategies' growth behavior according to data set variation.

According to these results, we see a significant disparity from one approach to another, especially when increasing the reference list size. This fact is noticed comparing the brute force (`TLSH`) and `HBFT` approaches. For a 1 TiB data set, the difference is ≈ 4915 times. The main reason for this difference is related to the AM tool under use, a fact that can be corroborated comparing all other approaches with the ones using `sdhash`, for instance. Comparing the brute force approaches using `TLSH` (digest of 35 bytes) and `ssdeep` (up to 96 bytes) with the one using `sdhash` (which has large digest size, varying according the object size - ≈ 2.6%), we can see another great disparity, being `sdhash` 374.50 and 136.53 times more expensive than `TLSH` and `ssdeep`, respectively. The same applies to `F2S2` and the two methods using bloom/cuckoo filters (`MRSH-NET`, `HBFT`, and `MRSH-CF`), where the former beats the others since it is based on `ssdeep` and the others on `sdhash`.

Due to efficiency reasons, the structures should fit into main memory, a major problem for some strategies as the reference list grows. We can see that the `HBFT` approach stands out due to its high memory consumption in comparison to others. It has a bad compression rate, consuming about 336 GiB of memory for 1 TiB data set size (corresponding to 32.81% of the whole set size). Given a blacklisting case and the increasing size of data nowadays (images and videos are becoming larger due to high-quality standards), 1 TiB is a reasonable size to consider. Therefore, this approach becomes impractical for examiners to handle. On the other hand, for the same amount of data, `F2S2` consumes only 1.71 GiB (0.17%), which is easier to deal with. Other strategies like a brute force for `TLSH` and `ssdeep`, `DHTnil`, and `iCTPH` consume even less, about 70, 192.00, 96.43, and 288.43 MiB, respectively.

We emphasize that the structure size must be taken into consideration when choosing a strategy, since it is a major requirement in practical scenarios. A good choice would be the one that fits the hardware specifications of the processing machine since loading the entire structure into main memory is the desirable form to have a more efficient search.

3.1.2.2 Lookup complexity

Another import requirement for SDSS is the lookup complexity, which gives us an idea on how the strategies would scale in response to the reference list data set increase, presented in the form of *Big-O* notation. Table 2 shows the asymptotic upper bound for performing a single lookup. Our results indicate `MRSH-NET` and `MRSH-CF` as the best options for performing efficient queries, with time complexity of $O(1)$, although they are limited to only membership queries. Besides, experiments corroborate this statement since they indicate that `MRSH-NET` (best case) is about 12 times faster than brute force `sdhash` (BREITINGER *et al.*, 2014a). The `HBFT` strategy comes up as our third option, with $O(log(n))$ complexity. All other approaches presented complexity equal to the naive brute force method ($O(n)$).

Although the lookup complexity is an important and necessary measurement for evaluating SDSS, in some cases having the time spent in the process is a more accurate form of comparison. Most strategies are much faster than brute force for normal operating conditions and yet have the same time complexity, as `F2S2`, for instance. Winter, C. et al. (WINTER *et al.*, 2013) show that calculating the complexity of this approach requires two steps: Finding candidates (digests sharing the same n-gram as the queried item) and similarity calculus. The first task can be accomplished with $O(log(n))$ for a fixed index table or $O(1)$ for dynamic resizing of the index table, while the second one presents complexity similar to brute force: $O(n)$. The reason for this high complexity is because the effort required is proportional to the size of the reference list. Then, when summing the complexity of the two steps, we get a complexity of $O(n) + O(log(n)) \approx O(n)$ (single lookup) for both dynamic resizing and fixed index table. However, in practice, the benefits achieved by `F2S2` will depend mainly on the efficiency and effectiveness of the candidates' selection. As we will not compare the queried objects with all reference list digests but with a restricted set of those sharing the same n-grams with it, we expect a much faster process. According to Winter, C. et al. experiments (WINTER *et al.*, 2013), F2S2 achieved speedup above 2000 times faster than brute force ssdeep.

Other strategies may also be faster than the brute force approach, as DHTnil and iCPTH. In both methods, instead of computing r operations for a single lookup procedure just like the brute force approach, we compute only $p+l$ operations, where p denotes the number of reference points chosen and l the sum of the number of digests presented in each selected node. The sum $p+l$ is expected to be much smaller than r, resulting in a significant reduction in the search time in practice. However, the time complexity for this approach is the same as the brute force one ($O(r)$), since $p+l$ is proportional to the data set size and increases with it.

It is important to analyze the strategies regarding their time complexity and also their running time for a more accurate comparison, since some schemes may perform better in practice. In section 3.2, we provide an evaluation considering the running time for some strategies in order to complement our analysis. We derive equations to estimate their running time for both phases (preparation and operational), showing how the strategies scale in practice and in which conditions one is better than others.

3.1.2.3 False positives

As discussed in Sec. 3.1.1.3, some approaches create false positives in the similarity digest search, as MRSH-NET, HBFT, and MRSH-CF. Others do not have the match decision incorporated in the strategy (brute force) and rely on the decision given by the tool.

DHTnil, iCTPH, and F2S2 are a middle term class since they have the match decision associated with the strategy. However, they do not create new false positives because the approach does not decide which objects are similar. The process is delegated to a function derived from the AM tool where its goal is to separate a small set of possible candidates only. The tool's comparison function is later used to compare the set with the queried item to decide the similarity, and hence reduce the number of objects to be manually inspected by forensics practitioners.

3.1.2.4 Resemblance/containment similarity detection

Detecting both resemblance and containment is a desirable property in any AM tool since a practitioner can find objects that resemble each other as objects contained in others. However, most AM techniques are designed to detect only resemblance, the most basic operation mode. sdhash is the only one that can efficiently identify both modes. This statement is corroborated by Lee, A. and Atkison, T. (LEE; ATKISON, 2017) showing that sdhash performed the best regarding this aspect in comparison to most tools. Since the strategy is mostly tied to the tool it uses, it becomes limited to the sort of detection performed by it.

Most strategies analyzed in this work cannot detect containment. Only those based on sdhash may have this ability (MRSH-NET, HBFT, and MRSH-CF). There is no current analysis on literature (from our knowledge) about the effectiveness of the strategies and the minimum amount of data shared between two objects for having a containment/resemblance detection. However, we believe that as these strategies encode the features extracted by sdhash into their structures, any element sharing the same (or a few) features will be a possible candidate for both detection modes. Since the strategies require a minimum number of consecutive features found in the filter to consider the queried object a match, any object fragment (small piece of data) already stored in the set is expected to have its features present in the filter and will be considered a match. For this reason, we believe containment can be detected by sdhash related strategies. Future work is necessary to corroborate this statement.

3.1.2.5 Approximate matching tool's precision

The search precision is more tied to the tool than to the strategy itself. In some cases, the strategy only reduces the number of comparisons the examiner should do. While some approaches are based on tools like sdhash, which have interesting characteristics and can detect resemblance and containments for a variety of object sizes without compromising its results, others rely on limited tools. An example is the classic ssdeep, limited to only comparing objects of similar sizes and not suitable for dealing with large objects. This is corroborated by Roussev, V. (ROUSSEV, 2011) and Breitinger, F. et al. (BREITINGER *et al.*, 2013), showing that sdhash outperforms ssdeep in accuracy and scalability. Furthermore, Breitinger. F. and Roussev, V. (BREITINGER; ROUSSEV, 2014) present an evaluation of ssdeep, mrsh-v2, and sdhash using real data (extracted from the t5 corpus database (ROUSSEV, 2011)). They analyze the precision and recall rates of these tools and point out that sdhash has the best overall performance. The authors also state that even though the precision rates of ssdeep and sdhash are high, the recall of all tools are relatively low.

With respect to Nilsimsa tool, Oliver, J. et al. (OLIVER *et al.*, 2013) state that even though this technique has powerful capabilities for resemblance detection, it suffers from significantly higher false positive rates compare to TLSH. Harichandran, V. S. et al. (HARICHANDRAN *et al.*, 2016) mentions that TLSH is less powerful than sdhash for cross correlation.

Considering the tools' precision aspect, the strategies using sdhash are a better choice than the ones using ssdeep, Nilsimsa, or TLSH since the final result will be more accurate and scalable. Besides, it supports both detection modes efficiently.

3.1.3 Discussion

All similarity digest search strategies presented so far either show a high cost associated with memory requirements have an approximate matching function not as good as the best ones available nowadays, or have high costs related to the lookup procedures. In the first case, we have the HBFT strategy which incorporates sdhash as AM function, having desirable detection capabilities (both resemblance and containment). However, the tree structure is too memory-consuming and becomes infeasible to work with large data sets. On the other hand, `F2S2` presents better scalability regarding memory consumption, but it uses `ssdeep` as a similarity function, which leads to major limitations regarding similarity detection that can compromise and/or restrict analysis.

`MRSH-NET` and `MRSH-CF` are strategies that use a good similarity tool (`sdhash`), have low lookup complexity, and require less memory consumption compared to `HBFT`. However, they are restricted to membership queries only, which limits their application. The two P2P strategies (`DHTnil` and `iCPTH`) have the smallest memory consumption on average, but they use weak AM tools (`Nilsimsa` and `ssdeep`), have high operating costs associated with the several machines necessary to perform the search, and they may suffer from network communication delays. These strategies suffer from the following dilemma: The fewer machines used, the more it becomes similar to brute force; the more machines, the higher the costs and delays.

The brute force methods are very time-consuming independent of the chosen similarity tool due to the high number of comparisons when dealing with large data sets. Any other strategy can perform better than brute force. They are not suitable candidates for conducting investigations since the amount of data for each case has been increasing very fast (QUICK; CHOO, 2014). Besides, they are strongly dependent on the tool in terms of precision, time and memory requirement.

In this first part, our findings showed that none of the SDSS presented so far addressed at least the most desirable aspects: low memory requirement, high detection capabilities (for both resemblance and containment), and efficient lookup procedure. In the second part of our study, we aim at showing how some of the strategies may behave in practical scenarios to complement our analysis.

3.2 Part II: Operational costs evaluation

In the second part of this analysis, we evaluate the operational costs of SDSS. We show how some strategies carry out the search when dealing with large data sets and develop equations to estimate their operational costs, allowing a more precise time comparison. We performed this theoretical analysis because most strategies do not have their source codes or a compiled version available (at least by the time we were executing experiments) to calculate their costs and compare them. Besides, some strategies were removed from comparison since they require a complex environment for performing the experiments (e.g., DHTnil and iCPTH, which are P2P-based approaches) and do not have interesting characteristics given our previous analysis. Here, we consider that the preparation phase was previously performed with all digests computed for each object (from the reference list) and inserted in each strategy's structure.

3.2.1 Steps in a similarity digest search procedure

We evaluate the necessary steps in the operational phase of each strategy and present equations that we developed to estimate the time needed for each of them.

3.2.1.1 Brute force

The procedure involving the brute force can be summarized in the following steps:

1. For each item in the target system, perform:
   a) Digest generation;
   b) For each item in the reference list, perform:
      i. Digests comparison.
   c) Return pair of digest with a higher similarity score above a predefined threshold *t*.

The above steps correspond to the operation phase of a simple brute force approach. We can estimate the time to perform such a task using Eq. 3.1.

$$T_{op} = i \cdot (T_{digCalc} + (r \cdot T_{compFunc})) \tag{3.1}$$

Here, i is the number of objects in the target system, $T_{digCalc}$ is the time for calculating a single digest using the chosen AM tool (`ssdeep`, `sdhash`, etc.) or hash function (SHA-1, SHA-2, etc.), r the number of objects in the reference list, and $T_{compFunc}$ is the time to compare two entries using the same tool.

Brute force can be used with any similarity tool. In our work, we chose `ssdeep` (KORNBLUM, 2006) and `sdhash` (ROUSSEV, 2010) to perform brute force and hence compare them with other strategies regarding time performance. We also chose SHA-1 hash function as a benchmark since most AM functions aim at achieving times close to it. We emphasize that SHA-1 vulnerabilities regarding collisions are not an issue here, as this hash function is not being used with a security role. Even simpler (and also compromised) functions as MD-5 would be useful here, as well.

3.2.1.2 F2S2

Upon an investigation process where the forensic examiner wants to perform KFF in a target system, the following steps must be done for `F2S2`:

1. Load the *F2S2* index structure into main memory;
2. For each item in the target system, perform:
   a) `ssdeep` digest generation;

b) n-gram extraction;

c) n-gram lookup (this step returns all digests that share the same n-grams as the ones belonging to the queried item; *L* represents the amount returned)

d) `ssdeep` comparison of the queried item with the *L* returned digests.

e) Return pair of digest with higher similarity score above a predefined threshold *t*.

To estimate the time to perform the operational phase of `F2S2`, we can use Eq. 3.2.

$$T_{op} = T_{indexLoad} + i \cdot (T_{genSsdeep} + T_{ngramExtr} + T_{ngramLookup} + (L \cdot T_{compSsdeep})), \tag{3.2}$$

where $T_{indexLoad}$ represents the time to load the index into memory (step 1), i the number of objects in the target system, $T_{genSsdeep}$ the time to compute a single `ssdeep` digest (step 2.a), and $T_{ngramExtr}$ the time to extract all n-grams from the corresponding object (step 2.b). $T_{ngramLookup}$ refers to the time to perform a lookup procedure in the index (step 2.c), calculated by Eq. 3.3. L represents the number of candidates sharing the same n-grams as the queried item, returned by the lookup process, and $T_{compSsdeep}$ denotes the time to calculate the Edit Distance for two digests (`ssdeep` comparison function)(step 2.d).

To calculate the time to perform the lookup procedure, we can use Eq. 3.3.

$$T_{ngramLookup} = g \cdot (T_{hashI} + (T_{compStrE} \cdot b)). \tag{3.3}$$

Here g is the number of n-grams extracted from the digest ($g = l_{dig} - n + 1$, where l_{dig} is the digest size (bytes) and n the n-gram sequence size), T_{hashI} the time to hash a string (n-gram index), $T_{compStrE}$ the time to compare two strings (n-gram e-key), and b the average number of different n-grams in each bucket (section 3.2.2.2 shows how to compute it).

3.2.1.3 MRSH-NET

The operational process using `MRSH-NET` involves the following steps:

1. Load the bloom filter structure into the main memory;

2. For each object x in the target system, perform:

   a) feature extraction (with `sdhash`), resulting in $|F_x|$ features;

   b) $|F_x|$ lookups.

   c) Return *True* (there is a match) in case a minimum number of consecutive features are found or *False* otherwise.

To estimate the time to perform this task, we first need to calculate the average number of features of the target system ($\bar{z}$) per object. To this end, we can use Eq. 3.4, derived from Breitinger, F. et al (BREITINGER *et al.*, 2014c) statement: *"`sdhash` maps 160 features into a bloom filter for every approximately 10 KiB of input file".*

$$z = (\mu \cdot 2^{20} \cdot 160)/(10 \cdot 2^{10}) = 2^{14} \cdot \mu, \tag{3.4}$$

where μ is the size (MiB) of all objects in the set and both factors, 2^{20} and 2^{10}, are used to change from MiB and KiB to bytes, respectively. By dividing z by the number of objects in the target system, we get its average ($\bar{z}$). Then, we can use Eq. 3.5 to estimate the required time of `MRSH-NET`, as follows:

$$T_{op} = T_{bfLoad} + i \cdot (T_{featureExtr} + (\bar{z} \cdot T_{bfLookup})). \tag{3.5}$$

where T_{bfLoad} is the time to load the bloom filter data structure into memory (step 1), i the number of objects in the target system and $T_{featureExtr}$ the time to extract the features from an object (step 2.a). The $T_{bfLookup}$ is the time to perform a lookup procedure in the filter (step 2.b), calculated by eq. 3.6.

$$T_{bfLookup} = T_{hashF} + (k \cdot T_{compStrF}) \tag{3.6}$$

T_{hashF} denotes the time to hash each feature of β bytes and k the number of sub-hashes. `MRSH-NET` inserts a feature in the bloom filter by first hashing it and breaking the hash into k parts (sub-hashes). The resulting pieces are used to set the bloom filter. $T_{compStrF}$ is the time to compare two strings of F bytes each (F = feature size divided by k).

3.2.1.4 HBFT

We can sum up the required `HBFT` operations by the following steps:

1. Load the bloom filter-based tree structure into main memory;
2. For each item x in the target system, perform:
   a) feature extraction (with `sdhash`), resulting in $|F_x|$ features;
   b) $|F_x|$ lookups in the tree.
   c) If any object *FIC* (File Identifier Counter) is higher than a minimum predefined number, then:
      - compare the corresponding object digest with the queried one (using `sdhash`).
   d) Return pair of digest with a higher similarity score above a predefined threshold t.

To estimate the operational phase time of `HBFT`, we can use Eq. 3.7.

$$T_{op} = T_{bfTreeLoad} + i \cdot (T_{featureExtr} + (\bar{z} \cdot (T_{bfLookup} \cdot h)) + T_{compFunc}). \tag{3.7}$$

where $T_{bfTreeLoad}$ corresponds to the time to load the structure into memory (step 1), i is the number of objects in the target system, $T_{featureExtr}$ the time to extract the features from a single object (2.a) and $T_{bfLookup}$ the time to lookup each feature in the tree (Eq. 3.6) (step 2.b). The h parameter denotes the number of steps required to reach an object in the tree (see Sec. 3.2.2.3), while $T_{compFunc}$ is the time to compare two digests using the chosen tool (step 2.c).

3.2.2 Methodology and some peculiarities

The SDSS costs were estimated using the formulas developed in the previous section. We have designed pieces of code simulating the operations required by the strategies and then measured the necessary time for performing them. However, some parameters required deeper analysis to understand their behavior and to estimate their values. To this end, we created a database of real data objects to perform the measurements. In this section, we discuss the details about this database, the peculiarities of some parameters, and finally, our results.

In this work, the analysis is focused on one of the most important parameters in forensics investigations: The data set size (of the reference list and target system). Other variations in the strategies parameters to find their best-operating conditions can be addressed in future works.

3.2.2.1 Test database details

To better understand the behavior of some operations and then estimate their values more precisely, we created a database from real data. This set encompasses over one million objects extracted from two Linux operating systems (Elementary OS client - Ubuntu 16.04-based - and Ubuntu 16.04 server), Microsoft Windows 10 Home, and also from personal data, which includes photos, documents, videos, applications, etc. Our database has 1,256,356 objects, corresponding to about 233.32 GiB of data. There were several typical applications installed in each operating system, including Latex, LibreOffice, Microsoft Office 2013, Foxit Reader, NetBeans IDE, Internet Explorer, Google Chrome, Firefox, and default operating system applications. There is a significant diversity about the object types encompassing our database: `.pdf`, `.jpg`, `.png`, `.bmp`, `.txt`, `.doc`, `.docx`, `.odf`, `.mkv`, `.avi`, `.py`, `.mp3`, `.wma`, `html`, `.jar`, `.rar`, `.c`, `.bin`, among others. Since this work focus on the best performance of the AM tools in their best conditions and our experience shows that `ssdeep` do not produce reliable results with large objects, we limited the object size in our database to 200 MiB. Future works will address scenarios using larger objects.

Given that the files have private information, we will not release them for the community. We believe that one can easily obtain the same sort of data and get approximate results (given the approximate number and size of our database) using a similar amount of the same types of files.

3.2.2.2 The F2S2 case analysis: Using different database sizes

To estimate some parameters of the F2S2 strategy, we implemented a simplified version of this approach, aiming to observe how data is spread over the hash table. Then, we estimated the values of variables b (number of different n-grams per bucket) and L (number of objects sharing the same n-gram as the queried item), necessary to the calculus of F2S2 operational phase. To this end, we generated ssdeep hashes for every object in our database (encompassing all four data sets) and inserted them in the index structure. The chosen hash table size was 2^{24} (four 6-bit base64 characters from the n-gram).

After inserting all n-grams obtained from our database objects in the index structure, a total of 75,549,716 n-grams were counted. However, 2,218,267 of 16,777,216 (13.22%) buckets were still empty, which means that lots of buckets contained multiple entries. Fig 16 shows the distribution of the number of different n-grams per bucket (b), while Fig. 17 is an amplified version of it, presenting only the b value for the 300 first buckets. Estimating b can be done using Eq. 3.8, as follows:

$$b = d_{ngrams}/n_{buckets}, \tag{3.8}$$

where d_{ngrams} is the number of different n-grams inserted in the hash table and $n_{buckets}$ the hash table size (number of buckets). The statistic information about the distribution of b in the table follows: standard deviation = 5.22, median = 2, and mode = 2. To find the average number of different n-grams in each bucket, we applied Eq. 3.8 for the data inserted in the F2S2 structure, obtaining a result of 2.27. Although we may find a few buckets with lots of n-grams, a significant portion of the table is empty and will lead to empty buckets, as we can see by Fig. 17. There is a peak in one bucket with a significant number of n-grams occurrences and others, but as expressed by the mode and median got from the analysis, most buckets have two or fewer elements. This fact led us to choose the average number of different n-grams as the b value (Eq. 3.8) since this seems a better choice when comparing to the other statistic components got from the analysis.

Concerning the number of objects sharing the same n-gram L, we have used Eq. 3.9 to calculate this value.

$$L = n_{obj}/d_{ngrams}. \tag{3.9}$$

Here, the new variable n_{obj} denotes the number of objects in our database. We cannot use the average as we did in the previous case because digests may share multiple n-grams and we are only interested in finding the number of different digests sharing at least one n-gram with the queried object, not the number of similar n-grams. When we apply Eq. 3.9 to the data of the F2S2 hash table, we get a result of 0.033, which express our expectation for the average number of different objects sharing the same n-gram.

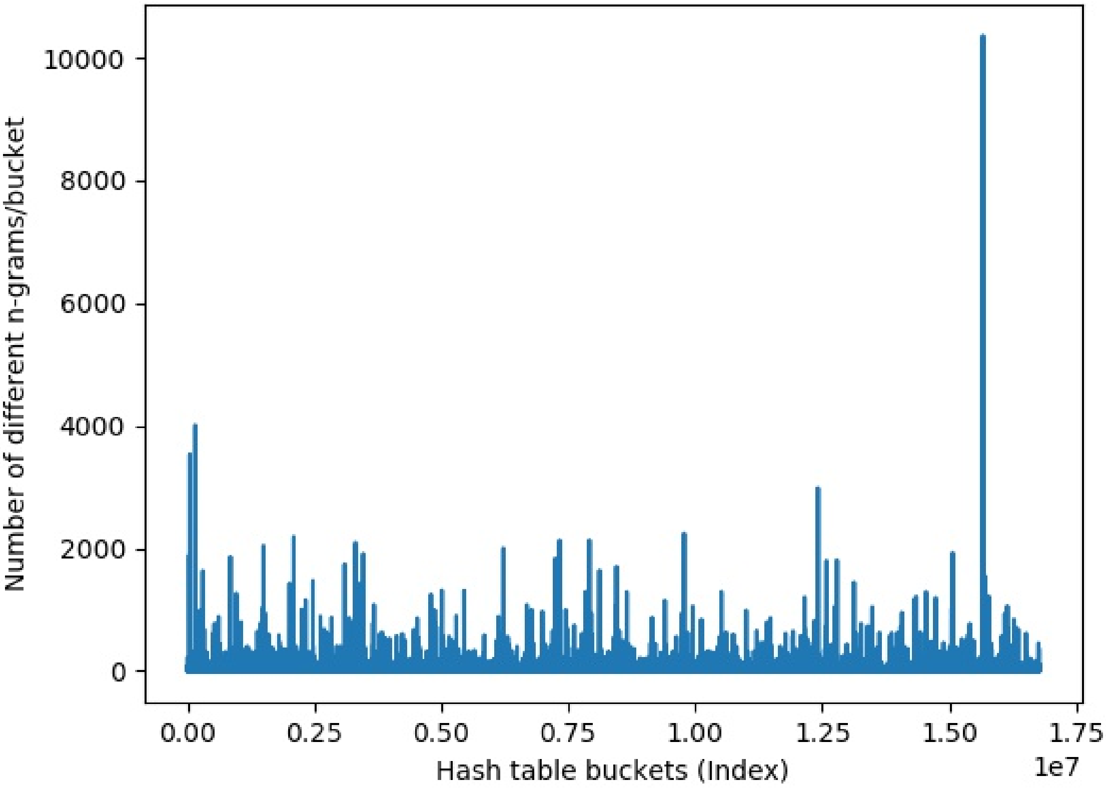


Figure 16 – Number of different n-grams per bucket

The values calculated so far are concerning to our data set only. Other sets of similar or different sizes may present different values for b and L since these parameters depend exclusively on the data being handled, which could have a different n-gram distribution. To extend our results to other sets, one needs to compute the b and L values for them. To this end, equations 3.8 and 3.9 can be applied. However, if by on hand we can define our hash table size ($n_{buckets}$) and count the number of objects in our database (n_{obj}) quickly, on the other hand, it is hard to know the number of different n-grams (d_{ngrams}) without inserting all ssdeep digests in the hash table and counting them.

To estimate the value of different n-grams (d_{ngrams}) for a data set with different types and sizes, required to the calculus of the parameters b and L, we need to find an expression that gives us such value based on the total number of the n-grams in this set. First, we consider the different data sources that form the database (Linux client, Linux server, Windows 10, and personal data) to simulate different systems and get a more general idea on how the n-grams are spread across them. Then, we insert each set separately in the F2S2 index structure and count the number of different n-grams. We also use the data got from our first analysis where the entire database (all former sets together) was inserted in the structure since it can represent a different and larger set.

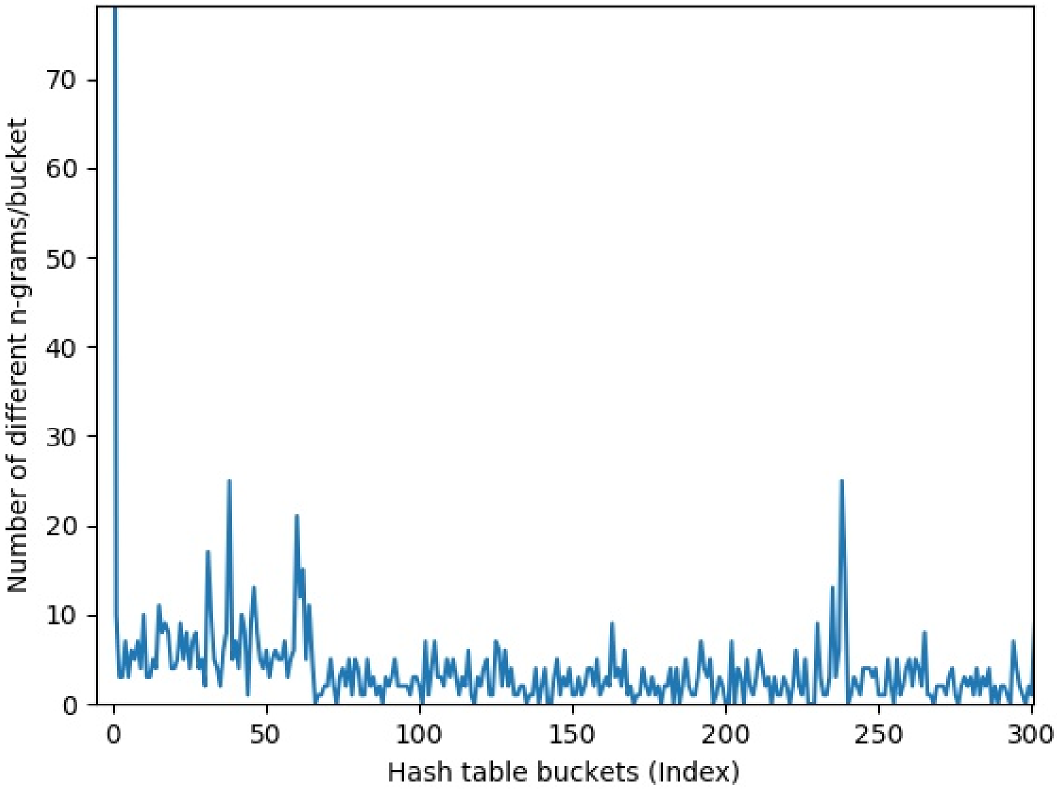


Figure 17 – Number of different n-grams per bucket (zoomed)

To find the expression that gives us the value of d_{ngrams}, we use regression analysis techniques. This way, we can determine the relationship between the values of different n-grams (unknown) and the total number of n-grams in the database (known). We chose the least squares method to this end. After applying this technique to our results, summarized in Tab. 3, we came up with the following expression:

$$d_{ngrams} = 2237231.04 + (0.4816 \cdot n_{ngrams}), \tag{3.10}$$

where n_{ngrams} is the number of n-grams of the database. We can obtain this number by multiplying the number of objects for the average number of n-grams in a single digest.

Table 3 – Number of different n-grams in each data source

Source	**Total number of n-grams**	**Number of different n-grams**	**% of different n-grams**
Linux client	22,937,595	12,828,701	55.93
Linux server	28,071,532	18,405,835	65.57
Microsoft Windows	11,376,395	7,357,119	64.67
Personal data	13,164,194	7,316,768	55,58
Σ	75,549,716	38,053,476	50,37

The expression above allows us to estimate the number of different n-grams in any database size and hence calculate both *b* and *L* values.

3.2.2.3 HBFT case analysis: Average steps in a search

Estimating the time needed for the *HBFT* strategy demands the knowledge of the number of times we will need to go through the tree structure to identify a given element. In a lookup procedure, there will be cases where the search stops in the root when the given element is not present in the structure, but in others, we may have to go through the entire structure to find it. Knowing the number of steps performed by element (h) will allow us to estimate the operational cost of the *HBFT* strategy (Eq. 3.7). To this end, one can develop an algorithm to count the average number of steps considering two scenarios: when the input is present in the tree and when it is not. In the first case, we go through the tree structure checking each node (bloom filter) for the presence of the given object until we find it, stopping the procedure in a leaf. For the second one, the search ends in the root tree, but there is also the possibility of occurring false positives, leading the process to go through the tree until all nodes of any level return a negative result. In the end, a weighted average between the number of steps found in each scenario and the number of elements must be returned, where the percentages of the two events occurrences are used as weight.

3.2.2.4 Parameters definitions and measurements

Using the proposed formulas we can estimate the costs for performing the searches with different data set sizes. The values adopted in our experiments are presented in Table 4. We performed the tests using the following machine: Elementary OS 0.4.1 Loki 64-bit (built on Ubuntu 16.04.2 LTS), i7-5500U CPU @2.40 GHz processor, 8 GB of memory, 1 TB SATA 3Gb/s hard disk drive (5,400 rpm), and NVIDIA GeForce 920M. We measured the time for each operation using the clock library from the C language, except for `ssdeep` and `sdhash` times (generation and comparison functions), which were measured using the time command (sys + user times) available on Linux distributions, since we used the compiled version of both tools. The times to compute the `ssdeep`, `sdhash`, and SHA-1 (both hashes generation and comparison) were calculated over the average object size, presented in the table. We repeated all experiments 20 times and took the average, taking care of clearing the cache each time to prevent previous results influencing new ones.

Before calculating the operational costs, we need to determine the strategies structure size, by using the formulas and parameters presented in sec. 3.1.1, adapting only the data set and average object sizes. We measured the time to load an object of 1 GiB from disk to memory (buffer of s_{buf} bytes) and adjusted this value according to the strategies structure size to simulate the loading process.

Table 4 – Similarity digest search strategies experiments - Parameters

Parameter	Description	Value	STD
s_{obj}	Average object size	195 (KiB)	-
$s_{strategy}$	Strategies' structure size	see section 3.2.2.4	-
s_{buf}	Buffer size	4096 (bytes)	-
T_{read}	Time to read 1 GiB from disk to memory	429.7565 (ms)	8.1452 (ms)
T_{genSH}	SHA-1 digest generation	0.6581 (ms)	0.0949 (ms)
T_{compSH}	SHA-1 comparison function	0.0102 (ms)	0.0037 (ms)
s_{ngram}	n-gram size	7 (bytes)	-
s_{index}	index size	4 (bytes)	-
s_{ekey}	e-key size	3 (bytes)	-
T_{genSS}	`ssdeep` digest generation	5.6000 (ms)	2.3324 (ms)
T_{compSS}	`ssdeep` comparison function	1.0000 (ms)	1.7320 (ms)
$T_{indexLoad}$	Time to load the index structure	$s_{strategy} \cdot T_{read}$ (ms)	-
$T_{ngramExtr}$	Time to extract n-grams from `ssdeep` digest	0.0207 (ms)	0.0062 (ms)
b	N-grams in each bucket	see section 3.2.2.2	-
L	Candidates sharing the same n-gram	see section 3.2.2.2	-
T_{hashI}	Time to hash s_{index} bytes	0.0813 (ms)	0.0115
$T_{compStrE}$	Time to compare two strings (s_{ekey} bytes)	0.0010 (ms)	0.0001 (ms)
T_{genSD}	`sdhash` digest generation	19.2000 (ms)	5.3066 (ms)
T_{compSD}	`sdhash` comparison function	19.4000 (ms)	4.7791 (ms)
T_{bfLoad}	Time to load the bloom filter structure	$s_{strategy} \cdot T_{read}$ (ms)	-
$T_{featureExtr}$	Time to extract features from an object	12.2975 (ms)	4.4667 (ms)
β	Feature size	64 (bytes)	-
T_{hashF}	Time to hash a feature of β bytes	0.0929 (ms)	0.0295 (ms)
$T_{compStrF}$	Time to compare two strings (β bytes)	0.0010 (ms)	0.0001 (ms)
k	Number of hash functions for the bloom filter	5	-
$T_{bfTreeLoad}$	Time to load the bloom filter tree structure	$s_{strategy} \cdot T_{read}$ (ms)	-
h	Average steps in a lookup procedure	see section 3.2.2.3	-
x	Degree of the tree	2	-

3.2.3 Evaluation

We measured the time for the three presented strategies (F2S2, MRSH-NET, and HBFT) and also for brute force using SHA-1, ssdeep, and sdhash. In our experiments, we first considered the database presented in section 3.2.2.1 as our reference list and then measured the time for performing over it using different target system sizes, varying them from 1 GiB to 10 TiB. It is important to mention that our database encompasses objects from different fonts gathered in a single set. Using the equations from Section 3.2.1 along with the values of section 3.2.2.4, we can estimate the strategies operational costs, shown in Fig. 18 (a).

According to our results, the strategies presented a linear growth as the target system size increased. As expected, the brute force approach had the worst results, with sdhash as the most expensive one, followed by ssdeep and SHA-1. The latter one had similar (but worst) results to HBFT. The best strategy regarding the operational time for our experiments was F2S2, presenting, on average, a speedup of 1,933,723.39, 99,676.83, and 996.82 times compared to the brute force approaches sdhash, ssdeep, and SHA-1, respectively, and 778.89 and 25.24 times better than HBFT and MRSH-NET strategies, respectively.

Fig. 18 (b) shows how the strategies perform over a reference list size variation

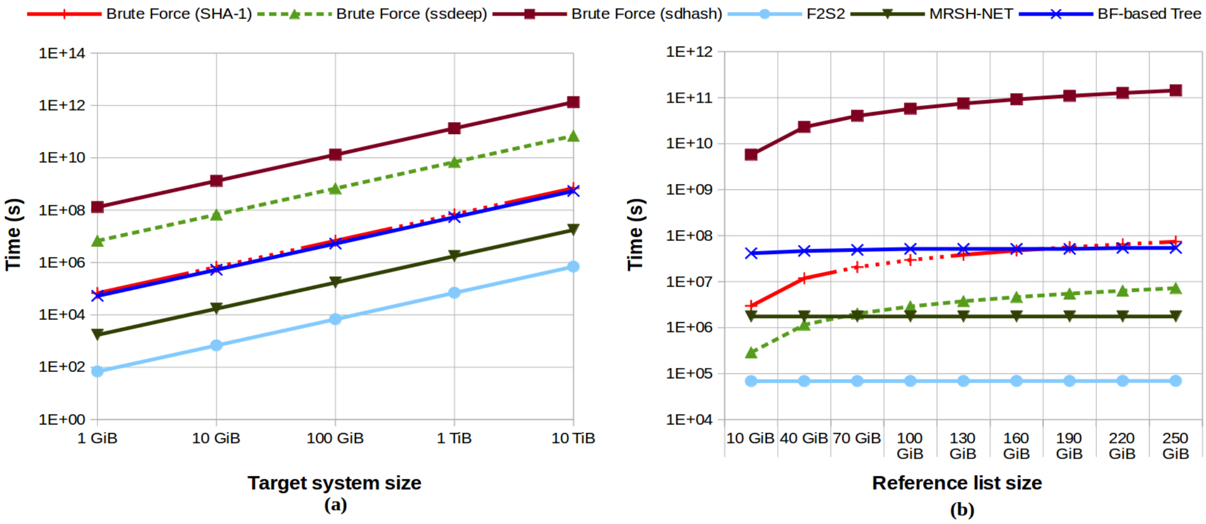


Figure 18 – Strategies operational costs:
(a) Target system size variation, considering a fixed size ref. list (233.32 GiB)
(b) Ref. list size variation, considering a fixed size target system (1 TiB)

(10 GiB to 250 GiB) and using a fixed target system size database (1 TiB). We limited our experiments to this particular range since it is the one covered by our data. Estimating the behavior of b and L parameters in the F2S2 equation for data set sizes far beyond ours could lead to wrong results once the data distribution in such cases could be different. For larger sets, additional studies are required to estimate these variables and hence the strategies operational costs.

Again, our results pointed out that sdhash brute force is the most time-consuming approach and, on average, F2S2 is the best one. However, despite the linear behavior of brute force, other strategies presented a different time variation as the reference list size increased. When increasing the reference list size, MRSH-NET showed constant time as expected due to the lookup procedure time being independent of the structure size. The time on HBFT was not constant but increased slowly due to the tree structure getting larger as the reference list expanded (number of elements growth), elevating the average number of steps in a lookup procedure and hence the time to go through the tree. For F2S2 strategy, we had a very small increase in time due to the number of similar n-grams increase. For data sets ranging from 10 GiB to 250 GiB, the F2S2 time increase was only 1.24% of the initial value.

3.2.4 Discussion

Our findings showed the prohibitive costs of the brute force method when performing over large data sets. For instance, an investigation of a 1 TiB target system using a reference list of 233.32 GiB and sdhash in the form of brute force could take more than 4 thousand years to be accomplished (around 6,917,958,887,350 comparisons between the data sets are necessary). The same search using the strategies MRSH-NET and the HBFT could take only 70 and 2145 days, respectively. Using the `ssdeep` (brute force), the same process would require around 754 years, while `F2S2` would only demand no more than 20 hours. It is important to mention that this enormous amount of time is due to the hardware used to perform the experiments. In a working station with powerful hardware and using parallelism techniques, these times are expected to decrease, but the relative differences found among the strategies are expected to remain.

Another important fact to mention is that even though `F2S2` presents higher lookup complexity than `MRSH-NET` and `HBFT` (see part I of this section), it performed better in our experiments when comparing large data sets. As expected, `MRSH-NET` was better than the `HBFT`.

Our results indicate that these strategies are a better choice than the simple brute force, and can reduce significantly the time consumed in KFF investigations. We also show how they scale when performing over different data set sizes.

3.3 Conclusions

In this chapter, we presented the Similarity Digest Search Strategies (SDSS) as an efficient alternative to the brute force method, which is too time-consuming and even prohibitive for dealing with large data sets. Our contributions to the field include a comparison of current similarity digest search strategies to point out strengths and weaknesses. We also provided a detailed analysis of the operational costs of some strategies, showing how they scale with different data set sizes and which performed best. Our results demonstrated that even though some strategies outperform others in some aspects, they fail in others. There is no currently suitable approach satisfying the most relevant requirements. From the best of our knowledge, this is the first work of this kind.

We conclude that new strategies are required to address the overwhelming amount of data forensics examiners have to deal with. For this reason, we present below a list of requirements for a new similarity digest search strategy, aiming to fulfill the most desirable points discussed so far and also other complementary ones.

A desirable strategy should:

- Have low memory consumption;
- Have an efficient lookup procedure (low time complexity);
- Support the most accurate approximate matching tools;
- Allow both detection modes: resemblance and containment;
- Return the actual object(s) similar to the queried one (in contrast to membership queries);
- Have few or no false positives in the process;

- Rely on its own storage structure;
- Insert/remove elements dynamically.

As shown by our analysis, none of the current approaches in the literature can address well all these aspects, since they mostly focus on improving only a particular aspect in detriment of others. Finding a balance of these requirements would already be a significant improvement to the field, allowing more efficient investigations in an era of an overwhelming amount of data.

Future studies regarding SDSS encompass extending our analysis to other similarity digest search strategies, along with the estimation of the preparation phase costs and variations of the strategies parameters to find the best operating conditions of each approach. Future work also includes analyzing scenarios using larger objects, since we limited the object size in our database to 200 MiB. We also plan to find new methods to estimate the parameters b and L of `F2S2` to scale with any data set size.

Given that some gaps remain on the similarity digest search field, we present in the next section, a new strategy that addresses most of the proposed requirements.

4 A new Similarity Digest Search Strategy: FSDS

4.1 Introduction

The similarity digest search strategies presented earlier work with different AM tools: `Nilsimsa`, `ssdeep`, `mrsh-v2`, and `sdhash`. However, newer AM tools normally can not work with existed strategies due to their unique characteristics; all strategies were developed taking into consideration the working process of a particular tool. This way, the default option for other tools when performing searches is the brute force approach. `TLSH` is an example, even though it presents attractive features for forensics investigations as robustness to detect small changes in objects, small digests, and low latency to create and compare digests. With no better strategy than brute force, `TLSH` use becomes restricted. In this chapter, we developed a search strategy to work with `TLSH` and thus reduce the required time for investigations. We start by a description of the proposed strategy, followed by an assessment of such approach comparing it with brute force to assess its effectiveness, memory requirements, lookup efficiency, among other parameters. The comparison of our strategy to others present in the literature will be done in future works.

4.2 Fast Similarity Digest Search strategy (FSDS) proposal

In this section, we propose a strategy to perform similarity digests search efficiently, called `FSDS`. Our strategy was inspired on the `F2S2` approach and uses a hash table to store digests based on their distance from a reference point. When performing a search with the goal of finding similar objects, one needs to compute the `TLSH` digest for the given object and the distance of such digest to the reference point. The result points to a position in the hash table where similar objects will reside. Using `TLSH` comparison function with all digests in the given position and with others stored around it, we can figure out whether there is or not similar items stored in the structure. In the following sections, we describe the details of our proposal along with our design choices. The implementation of our approach can be found in our GitHub page <https://github.com/regras/fsds>. Since the proposed strategy is based on the `TLSH` AM tool, details about how this AM tool works can be found in 2.

4.2.1 Structure

`FSDS` uses a particular kind of hash table to store digests, consisting of a central array (*main table*) and buckets. Each position in the main table, referred to as a bucket, can keep multiple entries composed by one field only, the *ID*, which is a link to the corresponding digest.

4.2.2 Distance function

The distance function maps digests to positions in the main table according to their distance to a reference point. We choose the `TLSH` comparison function itself to perform such a task since it estimates the hamming distance between two digests. This way, similar objects will produce similar digests with small differences between each other; they will be mapped to the same or near position in the main table. For instance, let's suppose we have digests A, B, C, and R, where the later is our reference point. Digests A and B are similar (with distance $d_{AB} \approx 0$), while A and C, and B and C, are very different ($d_{AC} \approx d_{BC} >> 0$). When we compare B and R, we get a distance of d_{BR}, which is expected to be close to d_{AR} since A and B are similar. On the other hand, comparing C to R is expected to result in a value larger than d_{AR} and d_{BR}. It is important to mention that we work with the absolute value (modulus) of the distance.

However, the downside of our distance function is that it may store different digests in the same bucket since this function is like a modulo operation. For instance, the character A is four positions distant from character E, but at the same time, I is also four positions distant from E. Taking E as the reference point, our function will store A and I in the same bucket since they present the same distance to E, even though the two characters are different (8 position distant). There is also a problem with the header calculus in our distance function (OLIVER *et al.*, 2013) that may store distant digests in the same bucket.

Given the limitations of our distance function, we need to take an extra step when performing queries. We first need to calculate the distance from the input digest to the reference point. Then, all digests stored in the main table at the given position and also the ones around it are selected. The extra step is related to performing a normal digest comparison (using `TLSH`) of the queried digest with all selected candidates to confirm similarity. We emphasize that our strategy is meant only to reduce the number of total comparisons one should make, limiting the search to a small digest subset instead of the whole set.

4.2.3 Reference point

The reference point is used in conjunction with the distance function to map digests in the main table. Our choice is to use a random `TLSH` digest selected from our data set for such task. Section 4.3.1 discusses the impact of our decision based on some experiments.

4.2.4 Main table size

The maximum size of the main table is limited by the maximum difference between two digests since our distance function maps digests into buckets based on their differences.

For this reason, we simulated a case where we have two digests that would return the maximum distance when computing TLSH comparison function (Header and Body distance) found in (OLIVER *et al.*, 2013); our results yielded a value of 2089. This way, the main table should have 2089 buckets. Although it can have less than this number, the consequence of using small sizes is a concentration of different digests in the same buckets, increasing the number of comparisons. One alternative to increase the table size beyond its capacity (2089 buckets) is to increase *TLSH* digest size.

4.2.5 Preparation phase: Inserting digests in the structure

The preparation phase of FSDS strategy, represented in Fig. 19 and detailed in Algorithm 4.1, consists in creating the main table and fill it up with our data set objects, got from the digital forensic practitioner reference list (black or white list), for instance. First, we need to compute the TLSH digest for the objects from our set and assign an ID to each one. Then, we store the ID of each digest in the main table position related to the distance from the corresponding digest and the reference point, according to our distance function. Eq. 4.1 shows the time taken by this process.

$$T_{FSDSpp} = T_{creation} + n \cdot (T_{calc} + T_{comp} + T_{insertion}) \tag{4.1}$$

Here $T_{creation}$ is the time to create the main table, n the number of digests in the reference list, T_{calc} and T_{comp} the time to compute and compare a TLSH digest, respectively, and $T_{insertion}$ the time to insert the digest ID in the table.

Algorithm 4.1 FSDS Preparation Phase

```
input : Main table size ($size$) and reference list objects ($refList$).
output : FSDS created structure ($mainTable$).
procedure $PreparationPhase(refList : list < string >)$
    foreach $object$ in $refList$ do
        $digest \leftarrow GenerateDigestTLSH(object)$
        $id \leftarrow CreateId()$
        $position \leftarrow DistanceFunction(digest, referencePoint)$
        $StoreDigestInMainTable(id, position)$
    end for
    return 0
end procedure
procedure $Main(refList : list < string >, size : int)$
    $mainTable \leftarrow CreateMainTable(size)$
    $PreparationPhase(refList)$
    return $mainTable$
end procedure
```

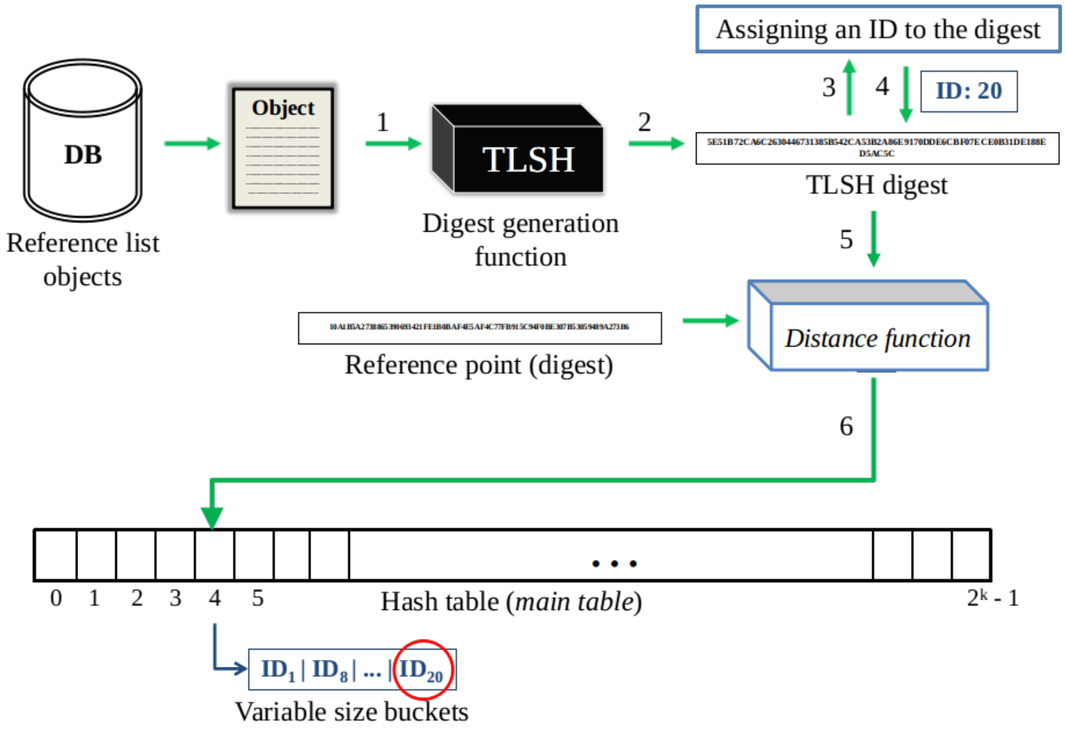

Assigning an ID to the digest
3
4
ID: 20
DB
Object
1
TLSH
2
5E51B72CA6C2630446731385B542CA53B2A86E9170DDE6CBF07ECE0B31DE188ED5AC5C
TLSH digest
Reference list objects
Digest generation function
5
10A1B5A2738865390693421FE1B0BAF4E5AF4C77FB915C94F0BE307B53059489A273B6
Reference point (digest)
Distance function
6
. . .
0 1 2 3 4 5
Hash table (main table)
2ᵏ - 1
ID₁ | ID₈ | ... | ID₂₀
Variable size buckets

Figure 19 – Preparation phase of FSDS.

4.2.6 Operational phase: Performing similarity digest queries with FSDS

After the preparation phase, FSDS will be available for handling queries. The input object has its TLSH digest created and then processed by the distance function to point out to the main table position where possible similar candidates will be placed. All digests in that position along with their neighbors (those stored in r near buckets from each direction) will be selected. Then, all candidates are compared with the queried digest to determine which ones are indeed similar. The whole process is depicted in Fig. 20 and also in Algorithm 4.2. Eq. 4.7 estimates the time for FSDS operational phase.

$$T_{FSDSop} = T_{load} + q \cdot (T_{calc} + ((L+1) \cdot T_{comp})) \tag{4.2}$$

where T_{load} is the time to load the structure into main memory and q the number of target system digests. T_{calc} and T_{comp} are the time to compute and compare $L+1$ TLSH digest, respectively, including the reference point calculus. The parameter L denotes the number of similar candidates returned in a query, estimated by the average number of digests per bucket (avg) multiplied by the number of near buckets visited ($L = avg \cdot (2 \cdot r)$).

4.2.7 Number of visited buckets

A starting point to determine the number of visited buckets near the position indicated by the distance function is the TLSH threshold value. For instance, if we set the threshold for t, every comparison performed by TLSH which result is less than this value will be accepted as similar. Then, all digests in the $r = t$ buckets below and above a given position will

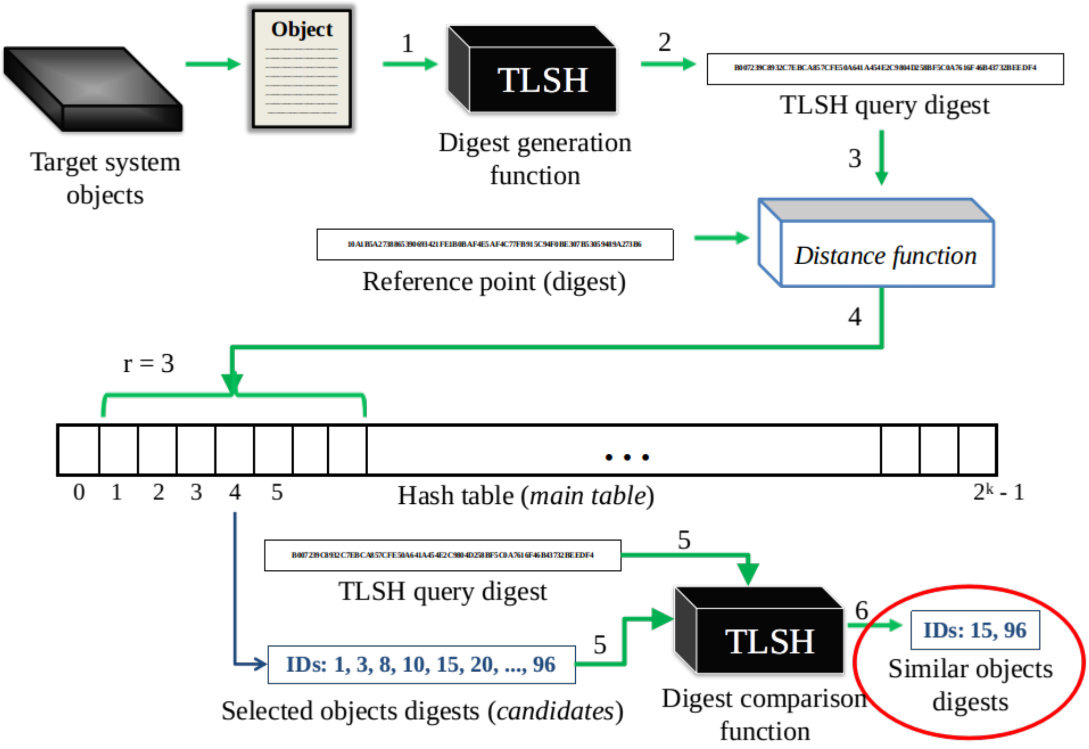

Object
Target system objects
1
TLSH
Digest generation function
2
B007239C8932C7EBCA857CFE50A641A454E2C9804D258BF5C0A7616F46B43732BEEDF4
TLSH query digest
3
10A1B5A2738865390693421FE1B0BAF4E5AF4C77FB915C94F0BE307B53059489A273B6
Reference point (digest)
Distance function
4
r = 3
0 1 2 3 4 5
. . .
Hash table (main table)
2^k - 1
B007239C8932C7EBCA857CFE50A641A454E2C9804D258BF5C0A7616F46B43732BEEDF4
TLSH query digest
5
5
IDs: 1, 3, 8, 10, 15, 20, ..., 96
Selected objects digests (candidates)
TLSH
Digest comparison function
6
IDs: 15, 96
Similar objects digests

Figure 20 – Operational phase of FSDS.

Algorithm 4.2 `FSDS Operational Phase`

```
input : Target system objects list ($targetList$), TLSH threshold ($t$), and radius ($r$) for searching similar objects in the hash table (main table).
output : List of similar pairs ($list$).
procedure $OperationalPhase(targetList : list < string >, t : int, r : int)$
    foreach $object$ in $targetList$ do
        $digest \leftarrow GenerateDigestTLSH(object)$
        $position \leftarrow DistanceFunction(digest, referencePoint)$
        $candidatesList \leftarrow findCandidatesOnMainTable(position, r)$
        foreach $candidate$ in $candidatesList$ do
            $score \leftarrow CompareDigestsTLSH(digest, candidate)$
            if $score \leq t$ then
                $similarObjList \leftarrow addToList(digest, candidate)$
            end if
        end for
    end for
    return $similarObjList$
end procedure
procedure $Main(targetList : list < string >, t : int, r : int)$
    $mainTable \leftarrow LoadMainTable()$
    $list \leftarrow OperationalPhase(targetList, t, r)$
    return $list$
end procedure
```

be candidates for similarity to the queried object. We can increase/decrease the threshold value according to the level of similarity demanded.

4.2.8 Using FSDS

`FSDS` proof of concept was implemented in C language and requires the `TLSH` tool. When initialized, the forensic practitioner needs to provide the hash table size (number of buckets) and choose one of the following options:

- **Digest data set (reference list) insertion:** The examiner provides a list of digests to be inserted in the `FSDS` structure.
- **Searching procedure:** The examiner chooses one of the following options:
  - **All objects:** All objects with a difference equal or less than a predefined threshold t with the queried object are returned.
  - **First object:** The search looks for one object with a difference of at most t with the queried object, stopping the search when finding the first one.
  - **Closest object:** The object with a smaller distance in comparison with the queried object is returned. No threshold is established, only a maximum number of buckets to visit.
- **Main table information:** Shows information about the structure, such as the number of filled and empty buckets and distribution of the digests in the main table.

4.3 Assessment

The tests performed in this work used the following resources: Linux Ubuntu 16.10 64-bit, Quad core (3425MHz) processor, 500 GB SATA hard disk drive (5,400 rpm), and 4 GB of memory. We measured the times using the `time` command (sys + user times) available on Linux distributions. We repeated all executions 20 times and took the average, with exception of `TLSH` brute force operational phase due to its large processing time. We took care of clearing the cache each time to prevent previous results influencing new ones.

Our experiments were conducted using *t5-corpus* as data set (ROUSSEV, 2011). The corpus is a collection of real-world files containing various file types having a total of 4457 objects (1.78 GiB) as summarized in Table 5. The average object size is 419 KiB. We extracted from *t5-corpus* 20 random objects to perform our experiments. The only restriction imposed for the object selection was that all of them had at least one similar object in the set.

Table 5 – *t5-corpus* file type statistics.

	html	**text**	**pdf**	**doc**	**ppt**	**xls**	**jpg**	**gif**
# of files	1,093	711	1,073	533	368	250	362	67
Avg. size (KB)	66	345	590	433	1,003	1,164	156	218

4.3.1 Impact of the reference point

We performed several experiments in the chosen data set to find the best reference point. Our goal is to locate the one that better spread the digests in the FSDS structure to reduce the number of comparison in the final step and hence to avoid false positives. We considered different elements in our experiments, such as object type (xls, doc, ppt, pdf, text, jpg), object size (ranging from 4 KB to 17 MB), entropy, and digest entropy.

An analysis of several experiments taking into consideration the four elements aforementioned was performed. After each test varying the parameters, we counted the number of empty buckets in the structure. Our results indicated no pattern regarding our choices. In general, the type, size, and entropy of an object caused a small impact in the digest spread in the main table. All tests presented similar results: $\approx 44\%$ of buckets filled up with digests on average. Due to the maximum main table size of 2089, we chose a structure of 2^{11} buckets, enough to comport all of our data set objects.

Since we could not identify any influence for choosing a good reference point, we picked a random one to continue our assessments. Although we believe that using any reference point will produce similar results (on average) as shown in our experiments, future work is necessary to corroborate our assumptions, using larger data sets. Furthermore, in this work, we have used an FSDS structure composed of a single table and reference point to store the object digests. Future studies also encompass extending *FSDS* to work with multiple tables, each having its own reference points. This way, a query object is compared to all reference points and the one(s) with the smallest distance, have the lookup process performed, in the same way as presented in 4.2.6. This improvement will allow us to work with larger data sets more efficiently.

4.3.2 Memory requirement

Before creating the FSDS structure, we need to create TLSH digests for each object in our reference list. The results must be stored in a database, XML file, or even in ordinary files to allow quick access to the digests for later identification. Next, we need to assign IDs to each digest, create the FSDS main table, and insert the data into the structure. In the end, the amount of memory required by FSDS is given by Eq. 4.3:

$$M = n \cdot (s_{digest} + s_f + s_{ID}) \tag{4.3}$$

Here n is the number of digests of our reference list, s_{digest} the size of a TLSH digest, s_f the size of the object name, and s_{ID} the size of the ID in the structure.

In our experiments, we have adopted the following values for these parameters: $n = 4457$, $s_{digest} = 70\ bytes$, $s_f = 12\ bytes$, and $s_{ID} = 4\ bytes$. We believe it is more than enough to allocate 12 characters for each object name as well as 4 bytes (corresponding to an integer variable in modern systems) for the ID, which are more than enough given the 4457 objects of our set. These choices led us to a memory requirement of $\approx 375\ KiB$. The TLSH brute force approach required $\approx 357\ KiB$, a small difference compared to FSDS strategy using the same parameters and data set (*t5-corpus*).

4.3.3 Lookup efficiency

We also evaluated the effectiveness of our strategy in finding similar objects. We compared FSDS with the naive brute force concerning the three searching modes implemented in our approach: *All objects*, *First object*, and *Closest object*. Brute force requires an all-against-all comparison for the modes *All objects* and *Closest object*. The *First object* mode depends on the objects in the data set and their similarity with the queried one. Supposing that we have at least one object with similarity below the threshold with respect to the queried item, we expect to find it with half of all comparisons on average. Considering the *t5-corpus*, it is expected to find a similar item with 2228 comparisons on average.

Performing the same search procedure using the FSDS strategy is more efficient. Table 6 shows our results for performing 20 lookups with random objects extracted from *t5-corpus* using the three searching modes with *FSDS*. We established a threshold of t=30 for low false positive rates (OLIVER *et al.*, 2013). However, we are not interested in measuring false positives rates, but the potential of FSDS strategy in locating similar objects. To this end, we evaluated *t5* using TLSH brute force to establish the ground truth of the set. Our next experiment aims to find the same similar objects pointed by TLSH but using FSDS instead, where we expect to have the same results but with a significant reduction in the number of comparisons. We first used the threshold t value as the number of visited buckets (as discussed in Sec. 4.2.7). However, after some experiments using other values ($r = \{10, 20, 50\}$), we found out that that $r = 20$ has the best balance between the number of digest comparisons and efficiency in finding similar items according to our experiments.

The results summarized in the table present the object queried followed by the score of the most similar object in the set and the results for the three modes. The *All objects* mode encompasses the total number of comparisons of a given search, the number of similar items found in relation to the expected one, and the score of the most similar object found. The *First object* and *Closest object* modes present the number of comparisons necessary in the search and the score of the similar object found.

Table 6 – FSDS-TLSH: Lookup efficiency

		Searching modes						
Random objects	**Min.**	**ALL objects**			**First object**		**Closest object**	
(*t5-corpus*)	**score**	**#Comp.**	**Found / Total**	**Min. score**	**#Comp.**	**Score**	**#Comp.**	**Min. score**
000649.text	5	203	6 / 6	5	5	11	203	5
001816.text	5	193	5 / 5	5	9	9	193	5
000047.xls	22	97	1 / 1	22	20	22	97	22
000073.html	19	200	1 / 1	19	179	19	200	19
000836.text	23	138	1 / 1	23	17	23	138	23
001434.text	8	212	5 / 5	8	67	10	212	8
001443.text	12	235	2 / 2	12	72	12	235	12
003056.html	9	245	5 / 9	9	3	14	245	9
001519.html	8	235	8 / 9	8	1	16	235	8
001531.html	15	189	3 / 3	15	11	15	189	15
001708.html	10	228	5 / 9	10	30	10	228	10
002138.html	0	229	2 / 2	0	4	0	229	0
004347.html	30	134	1 / 1	30	50	30	134	30
002038.html	0	286	6 / 6	0	3	0	286	0
000659.text	7	213	6 / 8	7	22	11	213	7
000935.text	18	221	3 / 5	18	86	18	221	18
001435.text	10	200	10 / 10	10	12	15	200	10
001619.doc	27	219	1 / 1	27	69	27	219	27
001723.html	25	167	1 / 1	25	96	25	167	25
002161.ppt	27	47	1 / 1	27	6	27	47	27
Σ	-	3891	-	-	762	-	3891	-

According to our results, there is a significant reduction in the average number of comparisons using FSDS in the three modes in relation to TLSH brute force. Considering the *All objects* and *Closest object* modes, only 3891 comparisons are necessary in contrast to the $((4457-20)\cdot 20 =)$ 88740 comparisons of brute force (4.38 % of the comparisons). Even though we could not find all similar objects in some cases, such as in the search for the object 003056.html, which found 5 objects from a total of 9, we were able to locate at least the most similar one as indicated by the minimum score. When comparing the *First object* mode of the two strategies, we were able to identify a similar item in all cases. In 35% of cases, we could not find the most similar one. However, the number of comparisons dropped from the $(\frac{(4457-20)}{2})\cdot 20 =)$ 44370 of brute force to only 762 using FSDS (1.72%), showing the efficiency of this strategy mode.

4.3.4 Time efficiency

We measured the time for a single TLSH digest generation and comparison over the average object size to estimate the preparation (Eq. 4.4) and operational (Eq. 4.6) phases using TLSH brute force and FSDS strategy (Eq. 4.1 and 4.7). Table 7 summarizes our results. The following parameters were adopted in the experiments: $T_{calc} = 1.16\cdot 10^{-1}$ (sec.), $T_{comp} = 1.29\cdot 10^{-3}$ (sec.), $T_{create} = 1\cdot 10^{-6}$ (sec.), $T_{insert} = 1\cdot 10^{-6}$ (sec.), $T_{load} = 3.4\cdot 10^{-4}$ (sec.), $L = 170.65$ and $r = 20$. All of these parameters were estimated by using the corresponding functions of the code provided in our GitHub page (see Sec. 4.2).

We also applied brute force over *t5* to measure the necessary time to accomplish both phases to corroborate the previous results got from the formulas. The preparation phase was completed in 487.88 sec., while the operational one in 24,730.25 sec. The difference is small but acceptable value which we believe is due to the `time` command limitation in detecting tiny differences of time and other low costs operations not accounted for in the estimation. The difference is around 6.35% for both preparation and operational cases (on average).

Table 7 – FSDS vs. Brute Force: Estimated Preparation and Operational costs

Strategy	**Preparation phase (sec.)**	**Operational phase (sec.)**
Brute force	517.01	26,102.94
FSDS (All objects mode)	522.76 (+1.11%)	1,502.41 (-94.24%)

Table 7 shows that brute force took for an all-against-all comparison about 7 hour and 15 minutes of processing time while FSDS, only 25 minutes (5.76%). The preparation phases of both strategies were similar, showing another benefit of FSDS since the extra cost for using it is not significant (see Sec 4.3.5). The reduction of the number of comparisons (discussed in the previous section) is significant, even though an analysis of other data set sizes is still required to measure more precisely the improvements regardless of size. However, the hours took by brute force in the whole process with *t5-corpus* data set were reduced to minutes with FSDS.

The asymptotic complexity of FSDS is $O(q \cdot L)$. Since the value of L tends to increase according to n (more objects will be placed in the same position in the hash table, requiring more digest comparisons), in the worst case, the complexity of this approach becomes the same as brute force. However, we showed that, in practice, we achieved a significant reduction in the time to find similar objects with FSDS than brute force.

4.3.5 Break-even point

The FSDS preparation phase requires time to create the main table, calculate the TLSH digests of all objects in the reference list, and insert them into the structure (Eq. 4.1). On the other hand, the brute force approach requires only the time to calculate the digests of the reference list objects, given by Eq. 4.4.

$$T_{BFpp} = n \cdot T_{calc} \tag{4.4}$$

To compensate the extra cost of FSDS in relation to brute force in term of queries:

$$T_{FSDSpp} + T_{FSDSop} = T_{BFpp} + T_{BFop} \tag{4.5}$$

where T_{BFop} is the operational phase of TLSH brute force, given by Eq. 4.6.

$$T_{BFop} = q \cdot (T_{calc} + (r \cdot T_{comp})) \tag{4.6}$$

We can ignore the costs of creating and loading the hash table (T_{create} and T_{load}) and inserting elements in the structure (T_{insert}) which can be very low compared to the other ones, as well as reading the digests from any database in the brute force approach. Then, we can estimate the break-even point by:

$$q' = \frac{r \cdot T_{comp}}{(r \cdot T_{comp}) - ((L+1) \cdot T_{comp})} \tag{4.7}$$

where q' represents the number of objects that need to be searched to compensate the extra costs of FSDS in relation to brute force.

Using the parameters presented in Sec. 4.3.3, we can calculate q'. Our results yield that 1.04 queries are necessary, which means that performing two queries only are enough for amortizing the extra cost of FSDS over brute force. The reason for the low cost is due to the generation digest process taking longer than the comparison one.

4.4 Conclusions

In this chapter, we proposed FSDS, a new strategy to efficiently analyze large digests data sets based on the TLSH tool. We performed experiments showing the efficiency of our approach in comparison to the simple brute force method in three different scenarios, where a reduction about 95% in time was observed with a minimum impact on precision. Future work encompasses extending *FSDS* to work with multiple tables instead of only one. This way, each table will have its own reference point, which we believe will allow the strategy to perform better over large data sets. We also plan to develop and test other distance functions to reduce even more the number of comparisons and hence the required time for forensics investigations. Another interesting work to be done is the comparison between FSDS (after extending it as discussed before) and other strategies, considering the efficiency of such approaches in finding similar objects.

The next chapter presents another problem faced with digital forensic investigations: The high number of false positives. We show that common structures found in many different objects are the reason for many false positives, and when we removed them, the similarity search is improved significantly in relation to the quality of matches.

5 The impact of common blocks on Approximate Matching

5.1 Introduction

The problem of current AM functions is that too many matches with irrelevant results (i.e., many false positives) are produced due to common structures found in many objects of the same and different file types. Since these structures may repeat in many different files, they are not suitable for assessing similarity in some contexts. This issue was first addressed by Garfinkel et al. (GARFINKEL *et al.*, 2010), who called them common blocks. Examples of such common data are header/footer information, color palettes, font specifications, or other data structures belonging to particular software vendors.

To differentiate the importance of similarity from a practitioner perspective, it is important to find a strategy to separate the content generated by users from content generated by applications. In this chapter, we present how we use approximate matching techniques to identify and remove common blocks and how our proposal can contribute to digital forensic investigations. Since we will use AM for dealing with common blocks and these tools work with the concept of features when creating the similarity digests, we will use the terms common blocks and common features as synonyms.

This chapter analyzes the effects of common blocks in approximate matching and shows how to use AM to identify and filter out these blocks. We discuss how common blocks are spread across different objects, their frequency, and how to improve the quality of matches. We also measure how precision/recall rates are affected by common blocks and show how the recommended threshold score to identify similar matches used by some tools (in particular for `sdhash`) is affected. We highlight that although common blocks can be useful in some scenarios (file type identification, for example), we believe this sort of information should be avoided because of the many irrelevant results produced when black/white lists are used. Future work encompasses exploring other applications for common blocks.

5.2 Related work

The concept of common blocks was first discussed by Garfinkel, S. L. et al. (GARFINKEL *et al.*, 2010) where the authors presented hash-based carving for content identification: The idea is to hash hard drive sectors (fixed-size pieces of data between 512 and 4,096 bytes) and try to match a block to a given file. The authors utilized the term *distinct* for the first time referring to blocks that occur only once in their test-corpus. Foster, K. (FOSTER, 2012) and

Garfinkel, S. L. and McCarrin, M. (GARFINKEL; MCCARRIN, 2015) continued the study of hash-based carving, and during their tests, they found many common blocks across files which made it difficult to prove the existence of a given file on a media under investigation. The solution adopted by those authors was to use *distinct* blocks only.

Besides using a database to filter blocks that appeared several times, Foster, K. (FOSTER, 2012) also proposed rules to identify the common (less relevant) blocks. The first rule was to ignore blocks with low entropy (i.e., repetition of the same character, `NULL` blocks, etc.). The second rule addressed blocks with repeating *n*-grams. In a follow-up work, Garfinkel, S. L. and McCarrin, M. (GARFINKEL; MCCARRIN, 2015) recommended additional rules as the entropy calculus was insufficient:

- *Ramp test:* Deals with blocks belonging to a special structure found in Microsoft Office documents, the Sector Allocation Table (SAT) (RENTZ, 2007);
- *White space test:* Searches and removes blank lines of 100 spaces, each terminated by a newline character. This pattern is mostly found in `JPEG` files;
- *The 4-byte histogram test:* 4-byte values, either repeating or alternating 4-byte values, are searched and eliminated. This pattern was found in Apple QuickTime and Microsoft Office file formats.

According to Gutierrez-Villarreal, F. J. (GUTIERREZ-VILLARREAL, 2015), the rules were redundant, and they proposed replacing them by a single one. Also, by focusing their research on `JPEG` files only, they found out that using blocks (4,096-byte segment of a file) with an entropy of 10.9 or higher removes many of the common blocks.

Garcia, J. (GARCIA, 2018) also explored how common blocks affect the similarity assessment by showing mismatches between fragments of data due to the common structure found among objects. They compared two approaches to extract the common blocks, one using the usual block-based hashing (disk sector level) and another using a rolling hash algorithm (similar to `ssdeep/mrsh-v2`) to explore the fragment hash uniqueness on `JPEG` images and compressed file archives. They report a successful detection of `JPEG` files inside compressed archives, arguing that many compression algorithms ignore high entropy data (a previous test on the data segment is done beforehand). Since `JPEG` files encompass this category, most content of such files is stored without compression. Later, one can correlate a `JPEG` file with a compressed archive and yet find some similarity (in case the same image is inside the archive).

However, the aforementioned references have some constraints when it comes to similarity detection. First, some authors restricted their applications to finding exact duplicates while we are interested in finding similar data. Second, previous research often focused on relatively large blocks for analysis; however, the larger the block, the more likely to encompasses changes in the object content, resulting in a different hash and preventing from finding similar blocks. Lastly, previous work ignores the block alignment problem, i.e., adding or removing a single byte at the beginning of an object will shift the subsequent bytes and change the representing hash. For this reason, we explore the use of Approximate Matching tools (which deal with the aforementioned issues efficiently) to perform object identification (of similar content).

5.3 Similarity classes

Whenever AM tools provide a score > 0 from the comparison of two objects, the similarity detected can be related to three classes, as shown below:

User-generated content (`UGC`): Data created by users, such as text, pictures, tables, etc.

Template content (`TC`): Data created by users that repeats over many different files. An example is a company's document template, where every file created by this company will contain the same header, footer, logo picture, etc. This is a form of similarity less relevant but could be useful for practitioners in specific cases. However, it may also lead to many irrelevant results.

Application-generated content (`AGC`): Data created by applications. An example is a file-header information with metadata required to access the file. This information is usually shared among (almost) all files of the same type and, in some cases, even with files of different types.

Since similarity can occur in these three forms, the definition of these classes is important so the digital forensic practitioners can perform their searches with their purpose in mind. Besides, we also need to provide methods for highlighting each similarity class accordingly in a way that AM results can be more specific to the practitioner's goals. In this thesis, we focus on providing a new approach for finding mostly `UGC` matches since we believe this is the most wanted form of similarity in scenarios like the KFF involving the use of black/white lists.

5.4 Impact of excluding common blocks for AM

In this section, we provide a detailed study about the impact of removing common blocks from the similarity digest of AM and how it would impact the number of matches.

5.4.1 Research direction, design decisions and implementation

The impact of common features on the behavior of approximate matching algorithms is not well explored. Therefore, this work addresses the following research questions:

*RQ*1 What are the common features? How frequently do they appear? How do they spread across various file types?

*RQ*2 How does ignoring common features impact the similarity detection (i.e., number of matches)?

*RQ*3 Is there a clear threshold N for which common features are ignored in the data set at hand?

*RQ*4 How does removing common features affect the runtime efficiency of the algorithm?

It is important to note that we are not interested in measuring the accuracy or detection capabilities of the algorithms but strictly to focus on the impact of common features on the similarity matches.

5.4.1.1 Procedure overview / workflow

To analyze the impact of common blocks, we performed several experiments. While for `RQ1` we compared the behavior of `sdhash` and `mrsh-v2`, the remaining tests concentrated on `sdhash` for three reasons: (a) It appears to be the most widespread approach; (b) it uses a constant and shorter feature size than its competitor; and (c) it produces more features as shown in Sec. 5.4.2.1. Additionally, `sdhash` utilizes the Shannon entropy to exclude undesirable features, eliminating some of the most common blocks by default, e.g., the ones composed by only values of 0s or 1s. Regardless of this choice, we expect similar outcomes for `mrsh-v2` although it has to be validated in future work.

Definition For a better and common understanding, we define the following terms:

S denotes a data set of files l and $|S|$ denotes its cardinality;

f denotes a feature where each f belongs to one or more files;

l represents a file and it is composed by a k-tuple of features, e.g., $l = (f_0, f_1, ..., f_{k-1})$. Note, features are not unique and may repeat in l;

$\mathscr{T}$ denotes a tuple containing all features for all $l \in S$ (order not important). Note, features are not unique and may repeat;

$\mathscr{F}$ denotes the set containing all features from $\mathscr{T}$. Recall that sets only contain unique elements;

$tf(f, \mathscr{T})$ denotes the feature frequency[1] and is the raw count of f in $\mathscr{T}$;

$itf(f, S)$ denotes the inverse term frequency[1] and is the raw count of $l \in S$ that contain f, i.e., the number of files containing f one or more times;

Common feature f is a feature where $itf(f, S) > N$ where N denotes a threshold.

[1] Terms are borrowed from information retrieval field as they are similar.

Procedure for RQ1 In order to understand the spreading of common features, we modified `sdhash` and `mrsh-v2` to store the extracted features in a database. Next, we executed both modified implementations on S and counted the frequencies of each feature. Lastly, we analyzed the frequencies which allowed us to compare `sdhash` and `mrsh-v2` as well as provide an overview of how many common and unique features exist in S. Results are discussed in Sec. 5.4.2.1. Additionally, we manually analyzed some common features which are highlighted in Sec. 5.4.2.2.

Procedure for RQ2 To measure the impact of ignoring common features on similarity matches (number of matches), we compared the matching behavior of `sdhash` to the proposed algorithm (based on `sdhash`, named `NCF_sdhash`) for various N. If a feature is common according to our definition, it will be ignored during processing. All ignored features are not represented in the similarity digest. Findings are summarized in Sec. 5.4.3.1.

Procedure for RQ3 To identify a valid threshold, we used some predefined N-values and manually inspected the matches. This allowed us to assess the best N for S as well as various file types. Results are presented in Sec. 5.4.3.1.

Procedure for RQ4 Our modified version is compared to the original version with respect to runtime efficiency, i.e., the time to generate and compare digests. Results are outlined in Sec. 5.4.3.6.

Differentiation of similarity To assess whether a match was related to user-generated content, application-generated content, or template similarity, we manually investigated the matches. When two files had no visual similarity, i.e., no common text, picture, table, or other user-generated elements, we classified the match as application-generated content similarity. A match was considered template similarity when the same layout repeated over several files, but their content was different. For instance, two `html` pages, ‘contact us’ and ‘our organization’, where both files had identical colors, elements disposition, menu bar, headlines, logo, etc., but their content was different. In this work, we focus on user-generated content which will be considered true positives; template and application-generated content are considered false positives.

5.4.1.2 Database implementation

We chose SQLite as the relational database management system for its simplicity and being open source[2]. In our experiment, three tables were created to store all extracted feature hashes including related metadata such as offsets. An overview is depicted in Fig. 21. The field highlight with an asterisk (*) is the index (SQLITE, 2019a).

[2] SQLite is slower than other databases which impairs runtime efficiency tests. If the focus is efficiency, a custom build storage solution as pointed out by Foster, K. (FOSTER, 2012) should be preferred.

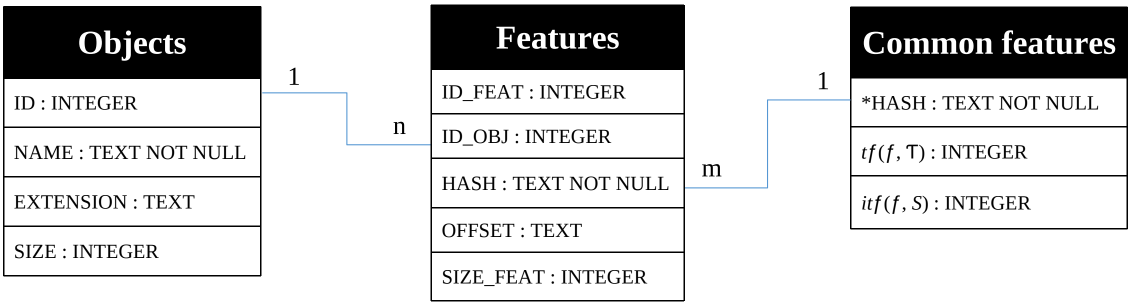


Figure 21 – SQLite database tables used to store all features (distinct and common) and related metadata.

The *Objects*-table contains information about the processed file such as name (path), size (bytes), and extension (file type). The *Features*-table stores data about the features themselves, e.g., feature hash, offset (position where the feature content is stored in the file), and feature size. Note, a file consists of many features and each one has a distinct entry in the features table (even if the same feature occurs multiple times, they all have different offsets). The final table, *Common features*, acts as counter storage for each feature f and contains $tf(f_i, \mathscr{T})$ for $0 < i \leq |\mathscr{T}|$ as well as $itf(f,S)$ for all $f \in \mathscr{F}$. For instance, for a set $S = \{A,B\}$, if feature f_1 occurs 10 times in file A and twice in file B, then $tf(f_1, \mathscr{T}) = 12$ and $itf(f_1,S) = 2$.

5.4.1.3 Implementation changes to existing tools

To perform our assessment, both `sdhash` and `mrsh-v2` required some modification to cope with a database. The database and developed applications can be downloaded from Github (C++ programming language): <https://github.com/regras/cbamf>. Specifically, the following changes were made:

DB creation: The feature extraction process in each tool was modified to insert the features and required metadata into the database.

No common feature (*NCF*): The second modification was slightly more complex as it performs queries in the database. In other words, before proceeding with an identified feature, the database is queried to check if it is a common feature, where 'common' depends on the parameter N. If a feature is common, it is discarded; otherwise, it will be further processed and added to the similarity digest. The new versions are named `NCF_sdhash` (based on `sdhash` 3.4) and `NCF_mrsh-v2`.

Lastly, we also decided to use FNV-1a (64 bits) for the feature hashing algorithm as suggested by Kameyama, A. S. (KAMEYAMA *et al.*, 2018) to improve the runtime efficiency without affecting the tool's precision.

5.4.2 Experimental results on common features

The data set S used for the experiments is the same *t5-corpus* (ROUSSEV, 2011) adopted earlier (see Sec. 4), which established itself as a default set in this domain. For validation, we performed a manual comparison of several matches to classify them according to their similarity type (user-generated content, application-generated content, or template content). Since the number of matches is too large, we only analyzed a sample of the them using either the appropriate software (e.g., MS office, browser etc.) or `Bless`[3] editor for binary files. These results can be found on Appendix B.

5.4.2.1 Common blocks overview in t5-corpus

Table 8 shows the statistics for the extracted features in S separated by tool. Notice that $max(tf(f,\mathscr{T}))$ returns the number of the most frequently counted feature while $max(itf(f,S))$ for all $f \in \mathscr{F}$ looks for the most common (wide spread) feature and returns the number of files sharing it (e.g., the last row in Table 8 indicates that the most common feature was found in 843 different files). Here, we can see that `sdhash` extracted more features than `mrsh-v2` (about 3.5 times), which was expected since `mrsh-v2` comes with a higher compression rate: `sdhash` produces fixed-size features of 64 bytes while `mrsh-v2` has variable-size features. In our experiment, we verified that the average feature size for `mrsh-v2` was 215.3 bytes.

Table 8 – Feature statistics across S.

Parameter	`sdhash`	`mrsh-v2`
$\lvert\mathscr{T}\rvert$	31,387,592	8,842,032
$\lvert\mathscr{F}\rvert$	27,203,732	8,049,461
$max(tf(f,\mathscr{T}))$	153,037	8,141
$max(itf(f,S))$	843	790

The frequencies of the extracted features are presented in Table 9, where the left column shows the analyzed condition followed by the two algorithms on the right. For instance, row $X > 10$ means that `sdhash` found 39,069 features that occurred more than ten times in $\mathscr{T}$. In contrast, the other part of the table (described with N values) focuses on the number of files containing particular features, e.g., last row indicates that for `mrsh-v2` we found ten features that were in more than 400 files. The results also showed that a significant number of features repeats frequently, e.g., `sdhash` found 2,663 features that repeated more than 100 times (this also means many features repeat within the same file). As indicated by the last part of the table, some features were widely spread among files, e.g., 579 features appeared in more than 50 files.

[3] <https://github.com/bwrsandman/Bless> (last accessed 2019-10-21).

Table 9 – Feature frequencies across S.

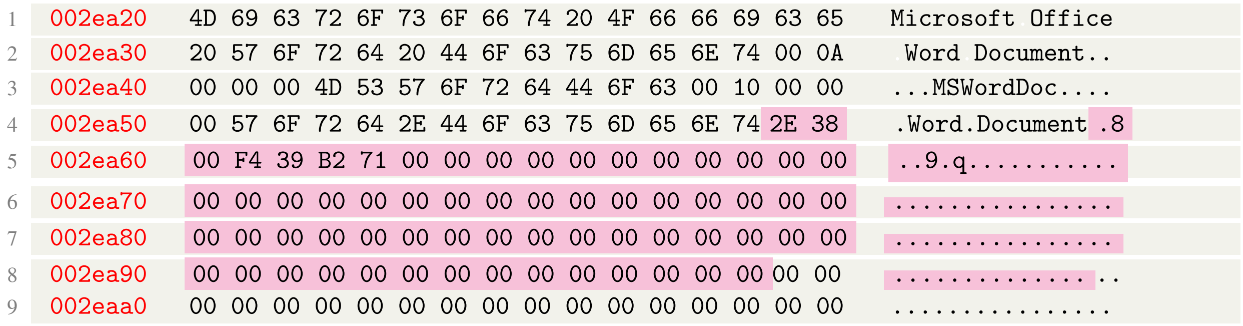
Condition	sdhash	mrsh-v2
$count(f \mid tf(f,\mathscr{T}) = X)$		
$X = 1$	25,861,720	7,751,709
$X = 2$	942,182	225,740
$X = 3$	145,236	28,392
$count(f \mid tf(f,\mathscr{T}) > X)$		
$X > 3$	254,594	43,620
$X > 5$	124,413	22,100
$X > 10$	39,069	8,044
$X > 20$	16,840	3,318
$X > 50$	5,986	1,232
$X > 100$	2,663	526
$X > 200$	1,124	231
$X > 400$	471	81
$X > 800$	185	32
$X > 2,000$	63	11
$X > 10,000$	6	0
$count(f \mid f \in \mathscr{F} \wedge itf(f,S) = N)$		
$N = 1$	26,467,390	7,886,026
$N = 2$	567,267	140,064
$N = 3$	68,719	12,251
$count(f \mid f \in \mathscr{F} \wedge itf(f,S) > N)$		
$N > 3$	100,356	11,120
$N > 5$	43,845	5,419
$N > 10$	5,386	1,356
$N > 20$	1,676	488
$N > 50$	579	200
$N > 100$	287	115
$N > 200$	115	55
$N > 400$	17	10

5.4.2.2 Most common features for each file type

Table 10 shows the features that repeated the most across different files of the same type and includes the feature's FNV-1a hash, the number of files having the given feature, and a brief description of the feature's content. The doc feature was related to necessary structural information common in Microsoft Office Word documents and is illustrated in Fig. 22 (highlighted area). It corresponds to the final part of a stream name followed by some setting and padding information. The doc feature was found in 442 files which is 83% of all doc's (533) in *t5-corpus*. Although not all doc's had this particular feature, variations of it were found in other files (see Fig. 23). While a different feature was selected, it belongs to the same file structure information. We believe the same happened in the remaining 91 *doc* files, where some specific changes affected the feature selection process. The most common feature of pdf, jpg, and gif files were related to color space information. In the case of html files, the common feature was associated with a well-known piece of java script code. For text, we found that all 18 files shared the same template, but they differed in content.

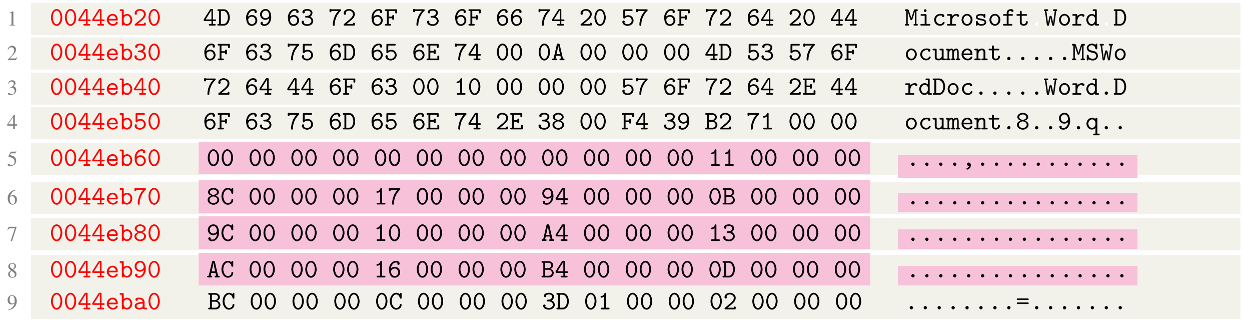
Table 10 – Most repeated features per file type and their content.

File type	**FNV-1a hash**	$itf(f,S)$ **for all** l **of the same type**	**Feature content**
`doc`	`c5e7aeb2482c56c0`	442 / 533	Necessary stream of compound files, specific of Microsoft Office Word documents.
`ppt`	`ef9a5a76d0df0c16`	357 / 368	Part of a document summary information stream with application defined properties.
`pdf`	`d5fb4ee41392d833`	347 / 1,073	Piece of an indirect object of a pdf stream, belonging to RGB color space.
`xls`	`b3310ce89e000aa4`	226 / 250	Font specification.
`jpeg`	`f0a05cdcac5796d4`	108 / 362	RGB color palette.
`html`	`cbac5aaf609ccf54`	61 / 1,093	Sample of a well-known piece of java script code to make web pages have rollover images.
`text`	`69c06bea6c3a3f10`	18 / 711	Part of a template content.
`gif`	`c91811dfd69ce32b`	5 / 67	Related to a global color table, which is a sequence of bytes representing RGB color triplets.



```
002ea20   4D 69 63 72 6F 73 6F 66 74 20 4F 66 66 69 63 65   Microsoft Office
002ea30   20 57 6F 72 64 20 44 6F 63 75 6D 65 6E 74 00 0A    Word Document..
002ea40   00 00 00 4D 53 57 6F 72 64 44 6F 63 00 10 00 00   ...MSWordDoc....
002ea50   00 57 6F 72 64 2E 44 6F 63 75 6D 65 6E 74 2E 38   .Word.Document.8
002ea60   00 F4 39 B2 71 00 00 00 00 00 00 00 00 00 00 00   ..9.q...........
002ea70   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
002ea80   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
002ea90   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
002eaa0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Figure 22 – Excerpt of the `000260.doc` file corresponding to the most common feature over this kind of format. The Bless hexadecimal editor is used to show the binary structure of the file. The highlighted area represents the feature.



```
0044eb20   4D 69 63 72 6F 73 6F 66 74 20 57 6F 72 64 20 44   Microsoft Word D
0044eb30   6F 63 75 6D 65 6E 74 00 0A 00 00 00 4D 53 57 6F   ocument.....MSWo
0044eb40   72 64 44 6F 63 00 10 00 00 00 57 6F 72 64 2E 44   rdDoc.....Word.D
0044eb50   6F 63 75 6D 65 6E 74 2E 38 00 F4 39 B2 71 00 00   ocument.8..9.q..
0044eb60   00 00 00 00 00 00 00 00 00 00 00 00 11 00 00 00   ....,...........
0044eb70   8C 00 00 00 17 00 00 00 94 00 00 00 0B 00 00 00   ................
0044eb80   9C 00 00 00 10 00 00 00 A4 00 00 00 13 00 00 00   ................
0044eb90   AC 00 00 00 16 00 00 00 B4 00 00 00 0D 00 00 00   ................
0044eba0   BC 00 00 00 0C 00 00 00 3D 01 00 00 02 00 00 00   ........=.......
```

Figure 23 – Excerpt of the `004964.doc` file corresponding to a feature similar to the one presented in Fig. 22. The Bless hexadecimal editor is used to show the binary structure of the file. The highlighted area represents the feature.

5.4.2.3 Common features across different file types

This section presents common features that were found across different file types; results are summarized in Table 11. Many features appeared across different files, especially

```
000f2d0f  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000f2e00  52 00 6F 00 6F 00 74 00 20 00 45 00 6E 00 74 00  R.o.o.t. .E.n.t.
000f2e10  72 00 79 00 00 00 00 00 00 00 00 00 00 00 00 00  r.y.............
000f2e20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000f2e30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000f2e40  16 00 05 01 FF FF FF FF FF FF FF FF 03 00 00 00  ................
000f2e50  10 8D 81 64 9B 4F CF 11 86 EA 00 AA 00 B9 29 E8  ...d.o........).
000f2e60  00 00 00 00 00 00 00 00 00 00 00 00 20 E5 BE 1D  ............ ...
000f2e70  DB 1E C8 01 BB 07 00 00 40 08 00 00 00 00 00 00  ........@.......
000f2e80  50 00 69 00 63 00 74 00 75 00 72 00 65 00 73 00  P.i.c.t.u.r.e.s.
```

Figure 24 – Snippet of the `001025.ppt` showing the first occurrence of $f =$ `5d60dae303171ac8` at offset `0xF2E0B`.

among `doc`, `ppt`, and `xls`, which are all compound files. For instance, $hash(f) =$ `536857624aa47c38` was part of a sector allocation table (SAT) data structure of compound files (RENTZ, 2007). However, we also found features shared by compound, `jpg`, and `pdf` files like `d5fb4ee41392d833`, which was part of a color space object (RGB). This was due to embedding objects like images into other file types (in our example all identified objects contained pictures). We also found features related to font specifications shared by different file types, too.

We also analyzed instances where features repeated within the same file. For instance, `5d60dae303171ac8` occurred 1,014 times in S (in 843 different files). One of the files containing it was *001025.ppt*. The feature was part of a compound file data structure related to a root directory entry of a stream, where the string *Root Entry* had to be present. The `ppt` contained four similar snippets (see Fig. 24) at different offsets: `0xF2E0B`, `0xF360B`, `0xF6E0B`, and `F720B`. Besides the feature, the majority of the bytes shown in the figure are identical among all four offsets.

Table 11 – Samples of common features that appeared on different file types.

f	$tf(f,\mathscr{T})$	$itf(f,S)$	$itf(f,S)$ separated by file types
`5d60dae303171ac8`	1,014	843	`doc` (404), `ppt` (265), `xls` (174)
`eee894cd42564cc9`	634	582	`doc` (286), `ppt` (295), `xls` (1)
`c02fde95428198dc`	540	531	`doc` (186), `ppt` (267), `xls` (78)
`ef9a5a76d0df0c16`	691	484	`doc` (92), `ppt` (357), `xls` (35)
`c5e7aeb2482c56c0`	468	467	`doc` (443), `ppt` (2), `xls` (22)
`d5fb4ee41392d833`	615	457	`doc` (14), `jpg` (49), `pdf` (347), `ppt` (47)
`536857624aa47c38`	451	437	`doc` (122), `ppt` (243), `xls` (72)
`ce5c0a5b70cca619`	3,608	402	`doc` (31), `jpg` (108), `pdf` (185), `ppt` (76), `xls` (2)
`3c0dc7d9b4044951`	224	220	`doc` (6), `xls` (214)
`1a5918d3d2ad6ffe`	228	203	`doc` (3), `ppt` (200)
`87b92f4dc954a121`	193	116	`doc` (14), `jpg` (49), `pdf` (6), `ppt` (47)

The feature that repeated the most in S ($max(freq(\mathscr{F}))$ = 153,037) is shown in Fig. 25 and belongs to template similarity in `pdf`s. For instance, it repeats 16,092 times in `001958.pdf`; another 144 files shared this feature one or more times. It is part of the cross-reference table (xref), which contains the references to all the objects in a pdf document. An object, in this case, is represented by one entry of 20 bytes, consisting of an offset (first 10 bytes), a space separator, the object generation number, another space separator, and a letter ‘f’ or ‘n’ indicating whether the object was free or in use. The final two bytes are the characters CRLF (0x0D0A) (ADOBE-SYSTEMS, 2008).

```
000bdf40   66 0D 0A 30 30 30 30 30 30 30 30 30 30 20 36 35   f..0000000000 65
000bdf50   35 33 35 20 66 0D 0A 30 30 30 30 30 30 30 30 30   535 f..000000000
000bdf60   30 20 36 35 35 33 35 20 66 0D 0A 30 30 30 30 30   0 65535 f..00000
000bdf70   30 30 30 30 30 20 36 35 35 33 35 20 66 0D 0A 30   00000 65535 f..0
000bdf80   30 30 30 30 30 30 30 30 30 20 36 35 35 33 35 20   000000000 65535
000bdf90   66 0D 0A 30 30 30 30 30 30 30 30 30 30 20 36 35   f..0000000000 65
000bdfa0   35 33 35 20 66 0D 0A 30 30 30 30 30 30 30 30 30   535 f..000000000
```

Figure 25 – Feature aecec3a6185401f1 from 001958.pdf that was found most frequently (153,037 times) in pdfs.

5.4.3 Impact on similarity detection

This section highlights the impact of removing the common features from digests. Thus, when comparing two digests, common features will not impact the similarity score.

5.4.3.1 Summary of number of matches in the data set

Table 12 shows the number of file matches performing an all-against-all comparison (excluding self-comparisons) on *S* (*t5-corpus*) using sdhash and NCF_sdhash which equals ($^{4457*4456}/_{2} =$) $9,930,196$ comparisons. Column one is the range of the similarity score; column two the number of file matches for sdhash followed by the number of file matches for NCF_sdhash for various *N*. Note that both tools return scores ranging from 0 to 100. However, we omit 0 scores as we are only interested in comparisons with some level of similarity.

Table 12 – Number of file matches by score range using sdhash and NCF_sdhash for ALL file types, discarding common features with occurrences $>$ N.

Score	**sdhash**	**NCF_sdhash for *N***					
		3	**5**	**10**	**20**	**50**	**100**
= 1	2,992	65	93	152	195	253	311
≥ 1	9,220	409	622	1,188	1,541	2,123	2,371
≥ 10	1,795	241	356	745	963	1,249	1,262
≥ 21	1,038	181	267	563	799	925*	925*
≥ 50	459	79	114	237	414	475	472
≥ 90	86	20	21	55	58	85	85
= 100	18	6	6	15	15	30	30

*Note: The same numbers in two columns do not mean that the sets of matches are identical.

There was a significant reduction in the number of matches when excluding common features: sdhash returned a total of 9,220 matches (score ≥ 1), while dropping common

Table 13 – Number of file matches by score range using sdhash and NCF_sdhash for doc files, discarding common features with occurrences $> N$.

Score	**sdhash**	**NCF_sdhash for *N***					
		3	**5**	**10**	**20**	**50**	**100**
= 1	1,082	7	6	10	9	19	44
≥ 1	4,095	49	81	97	100	135	194
≥ 10	607	33	62	62	61	69	72
≥ 21	166	25	48	47	49	50	50
≥ 50	15	6	13	12	15	14	14
≥ 90	0	0	0	0	0	0	0

features reduced it to 409 (-95%), 1,188 (-87%), and 2,371 (-74%) for *N* equal to 3, 10, and 100, respectively. As expected, the more restrictive we were (smaller *N*), the lower the number of matches.

To better understand the results, the upcoming sections focus on each file type results. Specifically, we compare all files to a given type against *S*.

5.4.3.2 Compound file type (doc, ppt, xls)

Compound files are known for storing numerous files and streams within a single file, in a hierarchical way, similar to a file system. The streams are further divided into small blocks of data (called sectors) used to store both user and internal control data. The entire file consists of a header and a list of all sectors. Each sector has a fixed-size (usually 512 bytes) defined in the header (RENTZ, 2007).

Tables 13, 14 and 15 summarize our findings. Our results show a similar behavior among all three types: lots of matches for the original sdhash for low score ranges and a significant reduction when removing the common features. For the upcoming detailed analysis, we focused on doc's but expect a similar behavior for the others.

Table 14 – Number of file matches by score range using sdhash and NCF_sdhash for xls files, discarding common features with occurrences $> N$.

Score	**sdhash**	**NCF_sdhash for *N***					
		3	**5**	**10**	**20**	**50**	**100**
= 1	42	9	13	35	26	21	25
≥ 1	133	27	54	98	95	106	108
≥ 10	36	7	16	36	36	37	37
≥ 21	16	4	9	16	16	16	16
≥ 50	2	0	0	1	2	2	2
≥ 90	0	0	0	0	0	0	0

Roussev, V. (ROUSSEV, 2011) mentioned that for compound types, matches with scores below 21 contain many false positives and should be neglected. Thus, we focused on the 166 matches with a score ≥ 21. After performing a manual comparison of all matches,

Table 15 – Number of file matches by score range using `sdhash` and `NCF_sdhash` for `ppt` files, discarding common features with occurrences $> N$.

Score	sdhash	NCF_sdhash for *N*					
		3	**5**	**10**	**20**	**50**	**100**
= 1	1,252	10	23	29	32	44	44
≥ 1	1,952	70	90	112	115	169	171
≥ 10	55	36	36	37	36	37	39
≥ 21	23	22	23	24	24	23	23
≥ 50	8	9	8	8	7	7	8
≥ 90	2	2	2	2	2	2	2
= 100	1	1	1	1	1	1	1

we conclude that 120 cases were not similar regarding user-generated content (UGC) and 28 matches were classified as template similarity (TC). The remaining 18 matches were similar in terms of UGC.

When considering the results of `NCF_sdhash`, we see a significant reduction; for $N = 3$ we only had 25 matches, in which four were classified as application-generated content (*AGC*) similarity and five as TC. The remaining 16 comparisons were UGC similarity (compared to `sdhash` two were missed). Increasing $N = 5$ resulted in 48 matches and also included 16 UGC matches plus four *AGC*; the other matches were related to TC. For $N \in \{10, 20, 50, 100\}$ we had similar results as $N = 5$; all missed two matches.

Focusing on matches with scores < 21 for `sdhash`, we randomly sampled 20 out of the 3,929 (4,095-166) total. 18 had no user-generated content similarity, and the remaining two had template similarity. On the other hand, `NCF_sdhash` returned 24 (49-25) matches: Seven template similarity, eight application-generated, and nine user-generated content matches. Out of the UGC case, three matches were cross file type comparisons which were matches between different file types (e.g., `doc` vs. `html`); the other six combined similar content as well as template similarity. The 24 matches for `NCF_sdhash` did not include the two matches identified through random sampling for `sdhash`.

In other words, removing common features reduced the number of matches significantly. For instance, for scores ≥ 1, `sdhash` had 4,095 cases while `NCF_sdhash` returned between 49 matches ($N = 3$; best case) and 194 ($N = 100$; worst case), a reduction of 99% and 95%, respectively. The reduction dropped to values between 85% and 70% when considering only matches with scores ≥ 21 (the recommended value for `sdhash`). However, 70% is still a significant reduction considering that the digital forensic practitioner may have to compare matches manually.

For the compound file type, using $N = 3$ revealed the best results. We argue that a significant reduction in the number of matches outweighs the two additional matches (UGC) identified by `sdhash`.

Remark Compound files usually have a minimum size; for the types discussed here it is three sectors (1,536 bytes). Consequently, an empty document will have at least 1,536 bytes of structural information which impacts the final similarity score especially for small files. Since small files have more application-generated than user-generated content, many (undesirable) matches may occur.

5.4.3.3 PDFs

Out of the 92 matches for `sdhash` ≥ 21 in Table 16, only 38 pairs could be manually evaluated; 11 matches included *UGC*. In the remaining pairs at least one of the files was corrupted[4]. A closer look at the *AGC* matches (27) revealed that the majority of features were related to color information (e.g., `d9e1c063e9c0ba1c`); we also found some features (`ca80692484c3235c`) corresponding to a `pdf` object containing Adobe's Extensible Metadata Platform (XMP) data, a package to add metadata to images (but also other media files). Another feature (`4c815162434ce18d`) contained bytes of the XMP data object and lots of blank spaces.

The impact of removing the common features was again significant. `NCF_sdhash` with $N = 3$ returned exactly the 11 *UGC* matches found by `sdhash` plus one extra pair that `sdhash` scored with 20. Raising $N = 5$ resulted in 21 matches; the additional nine matches were related to application-generated similarity. Thus, for `pdfs`, $N = 3$ worked perfectly.

Table 16 – Number of file matches by score range using `sdhash` and `NCF_sdhash` for `pdf` files, discarding common features with occurrences > N.

Score	sdhash	NCF_sdhash for *N*					
		3	5	10	20	50	100
= 1	492	33	39	62	104	128	163
≥ 1	1,684	91	125	286	393	488	674
≥ 10	191	20	33	109	117	161	171
≥ 21	92	12	21	76	76	88	88
≥ 50	45	4	7	37	37	47	45
≥ 90	31	3	3	27	17	29	29
= 100	0	0	0	0	0	0	0

5.4.3.4 TEXT and HTML

`Text` and `html` are flat file types that do not contain *AGC* information. However, `html` files may contain markup elements or scripting languages (e.g., java script). As a consequence, Roussev, V. (ROUSSEV, 2011) suggested using matches with a score ≥ 5 (compared to ≥ 21).

An overview of the `text` results is given in Table 17. The 27 matches found by `sdhash` consisted of 25 *UGC* matches, one related to *TC* and the last one was not actually

[4] We found that despite the `.pdf` extension, these objects are not *pdf* files but edited `html` files with a few line feed (hex:`0A`) and space (hex:`20`) characters inserted into their beginning. The list of these objects can be found in our Github page: <https://github.com/regras/cbamf>.

a text file (although file extension was `text`, the file was a `doc` and matched another `doc`). Running `NCF_sdhash` and $N = 3$ returned 26 matches, where 23 fell into *UGC* similarity, two were related to *TC*, and the last one was a cross file type comparison with one of the corrupted `pdf` files. Two matches were missed: in one case there were many changes throughout the file; in the other the `text` file was converted into `html`. Setting $N = 5$ or higher solved this problem; all 25 *UGC* matches were found. In summary: the exclusion of the common features for `text` was less effective than other file types but did harm for $N \geq 5$.

Table 17 – Number of file matches by score range using `sdhash` and `NCF_sdhash` for `text` files, discarding common features with occurrences > N.

Score	sdhash	NCF_sdhash for *N*					
		3	5	10	20	50	100
= 1	14	6	8	13	7	9	8
≥ 1	57	35	39	45	41	42	41
≥ 5	27	26	27	26	26	26	26
≥ 10	25	23	25	25	25	25	25
≥ 21	20	18	19	19	19	19	19
≥ 50	5	6	6	6	6	6	6
≥ 90	0	0	0	0	0	0	0

With respect to `html`, results were different and are shown in Table 18. `sdhash` returned 1,052 matches with scores ≥ 5. In comparison, `NCF_sdhash` reduced this number for small *N*'s but found more matches for higher *N*'s (discussed later). Due to a large number of matches, we randomly sampled 30 cases in each analysis. `sdhash` had no *UGC* match in all 30 samples; 28 were template similarity cases and two showed *AGC*. `NCF_sdhash` with $N = 3$ had six *UGC* cases and 24 *TC* (for another 30 samples - all different from the first 30). For $N = 5$ and 30 new samples, 25 cases showed template similarity, four cases *AGC* and one *UGC* similarity (an embedded object with minor changes). The tool `sdhash` found all *UGC* matches present in the 90 samples, while `NCF_sdhash` (for $N = \{3,5\}$) missed one match. To conclude: $N = 3$ had the best cost/benefit scenario for `html` files.

Table 18 – Number of file matches by score range using `sdhash` and `NCF_sdhash` for `html` files, discarding common features with occurrences > N.

Score	sdhash	NCF_sdhash for *N*					
		3	5	10	20	50	100
= 1	43	16	23	24	33	53	55
≥ 1	1,215	185	281	617	857	1,275	1,278
≥ 5	1,052	152	227	568	787	1,099	1,099
≥ 10	936	139	202	510	721	976	976
≥ 21	759	111	158	409	643	765	765
≥ 50	394	57	83	175	349	409	407
≥ 90	56	15	16	26	39	57	57
= 100	17	5	5	14	14	29	29

Remark: More `NCF_sdhash` matches than `sdhash` Table 18 suggests that the number of results increased for `NCF_sdhash` for some N's (e.g., compare column `sdhash` vs. $N = \{50, 100\}$). This behavior was the opposite to other file types which is due to hash collision (`sdhash` uses SHA-1 for hashing the features while `NCF_sdhash` uses FNV-1a). We confirmed this by looking into our database: The most frequent feature has 61 different occurrences ($\omega = 61$). Consequently, `NCF_sdhash` for $N = 100$ should have identical results to `sdhash` since no feature was removed.

For instance, the last row (= 100) shows 17 matches for `sdhash` but 29 for `NCF_sdhash` with $N = \{50, 100\}$. The 12 new matching pairs involved two files (`001326.html` and `003467.html`) and received `sdhash` scores between 92 and 98. The number of features extracted was 661 by `sdhash` and 660 by `NCF_sdhash` (for $N > 50$). The difference on both was related to a single feature that had a collision with other feature when inserted into the digest (bloom filter) of the file. Since this feature was different from the ones of the other files under comparison, removing it made the similarity score increase to 100. It is important to mention that changing a single file may impact several comparisons, as shown by these two files that affected 12 matches. For `NCF_sdhash` with $N = 50$ we had the same situation, but some other comparisons were affected since in this case two `html` features were removed as they repeat 55 and 61 times each.

5.4.3.5 JPEG and GIF

Bytewise approximate matching algorithms work less well on images (but are good for detecting embedded images in compound files). Consequently, no matches were revealed for scores above 21, as shown in Tables 19 and 20. We investigated the 24 `jpg` pairs having a `sdhash` score >= 10 where two matches showed some similarity: a similar `jpg` image was found inside a `ppt` file. However, the pictures were not identical, and thus we categorized it as *AGC* similarity. The other 22 cases had no visual similarity.

Selecting 30 random samples from the 883 (907-24) matches (with score < 10), we found one case (score 3) where one picture was a scaled version of the other. These two `jpgs` had 555 and 1,497 features, respectively. The overlap was 129 features, where 16 were exclusive to these two `jpgs` ($\omega = 2$ for these 16 features) and related to header and EXIF information. Specifically, `97e7cd722414356e` was part of the JFIF header of both files, and features `54fbfe2c46f4bed8` and `581f94d602fd9ac6` were related to EXIF data, such as date time and camera information. This match was hard to classify: on the one hand it is *AGC*; on the other hand it ties it to a particular user. More details are discussed in Sec. 5.4.4: *Differentiating between user-generated and other types of content*. When looking at the scores < 10 and $3 \leq N \leq 20$, only one match had similarity related to template (pictures shared the same background, fonts, and colors). We did not analyze the matches for $N = \{50, 100\}$.

`gifs` had a similar behavior and only one match (`gif` to `text`) was found by `sdhash` which was a false positive. `NCF_sdhash` also had a false positive (with an `html` file) for some N-values.

For both file types, `sdhash` worked suboptimal which is a known challenge for bytewise approximate matching. However, as shown by our results, `NCF_sdhash` reduced the matches without impacting the quality of the results. $N = 3$ worked reliably in both cases.

Table 19 – Number of file matches by score range using `sdhash` and `NCF_sdhash` for `jpeg` files, discarding common features with occurrences > N.

Score	sdhash	NCF_sdhash for N					
		3	5	10	20	50	100
= 1	547	3	2	3	2	8	10
≥ 1	907	4	3	5	4	63	69
≥ 10	24	0	0	0	0	0	0
≥ 21	0	0	0	0	0	0	0

Table 20 – Number of file matches by score range using `sdhash` and `NCF_sdhash` for `gif` files, discarding common features with occurrences > N.

Score	sdhash	NCF_sdhash for N					
		3	5	10	20	50	100
= 1	1	0	0	1	1	1	0
≥ 1	1	0	0	1	1	1	0
≥ 10	0	0	0	0	0	0	0

5.4.3.6 Performance test

Besides evaluating how the similarity was affected by removing the common features, we also measured the runtime efficiency as well as the compression rate for `NCF_sdhash` and compared to `sdhash`. The results are shown in Table 21. Here we present tests measuring the time taken for creating digests for each file in S and then performing an all-against-all comparison (excluding the self-comparison) using both tools. We also measured the final digest size of all files of the set to examine how the compression rate was affected. In our tests, we did not include the offline steps (i.e., time to extract all features and insert them into the db). Although this task requires a long time, it does not impact the investigation as it can be done offline or at any time. It is also done only once.

The test environment is an i7-5500U CPU @2.40 GHz processor, 8 GB of memory, 1 TB SATA 3Gb/s hard disk drive (5,400 rpm), and NVIDIA GeForce 920M, running an Elementary OS 0.4.1 Loki 64-bit (built on Ubuntu 16.04.2 LTS). Each experiment ran 20 times; results were averaged. The cache was cleared every run to prevent falsification of the results. We also turned off all unneeded system services and stop unnecessary applications (KIM *et al.*, 2012). The measurement was done using Linux `time` command (`sys` + `user` times).

Table 21 – Runtime and digest size of `sdhash` and `NCF_sdhash` for *t5-corpus*; includes the measurement and standard deviation (±).

	sdhash	**NCF_sdhash for** N		
		3	**10**	**100**
Digest generation (sec)	87 s ± 0.92	108 s ± 0.42	110 s ± 0.41	110 s ± 0.41
All-against-all comparison (sec)	433 s ± 11.62	413 s ± 11.52	452 s ± 12.53	458 s ± 15.82
Digest size (bytes)	64,321,035	62,594,719	63,786,954	64,138,018

As shown by the results, `NCF_sdhash` is about 24% to 26% slower with respect to digest generation time which is related to querying the db (verifying if a feature is common or not). As expected, the runtime is independent of N as shown in the Table 21. We also measured the time to perform an all-against-all comparison utilizing the existing db. `NCF_sdhash` ($N = 3$) was slightly faster than `sdhash` which is due to the removed features resulting in fewer bloom filters (note the code related to the comparison function did not change). On the other hand, higher N-values for `NCF_sdhash` removed fewer features and thus have similar times to `sdhash`. The last parameter measured is the digest size generated for S. Here, `NCF_sdhash` was superior since it removed the common features, resulting in smaller digests. The lower the N-value, the larger the reduction achieved.

Remark It is important to highlight that the only optimization performed in the `NCF_sdhash` code was the creation of database indexes for the common feature table. As mentioned before, the creation of the database is required only once and can be done offline; afterwards it can be used for all investigations. `NCF_sdhash`'s bottleneck (when creating digests) are SQLite queries. The complexity for verifying whether a feature is common or not is $O(log(|\mathscr{T}|))$ (using database indexes, (SQLITE, 2019b)). Other optimizations focus on SQLite itself (PUROHITH *et al.*, 2017). On the other hand, one could move to a customized storage solution, which should improve the query performance (FOSTER, 2012). With respect to a database update: The features of the new objects are extracted (using the tool) and inserted into the database which is not time critical.

5.4.4 Discussion

Based on the experiments described in the previous section, we will discuss the lessons learned and the impacts of removing the common features, starting with the research questions.

RQ1. What are the common features? How frequently do they appear? How do they spread across file types?

Foster, K. (FOSTER, 2012) stated that most files are made up of distinct blocks/features that identify a specific file which also holds for our S: Given the 31 million features, only

0.8% were common (for $N = 3$). However, as pointed out by Garfinkel, S. L. and McCarrin, M. (GARFINKEL; MCCARRIN, 2015), features that appear to be unique could be uncommon if the data set is expanded.

The common blocks spread widely among the same or different file types although different file types showed different behavior. For instance, compound file types (`doc`, `ppt`, `xls`) have a high degree of 'default similarity' as they share a similar internal structure which is in contrast to flat types (`text`, `html`) and compressed types (`jpg`, `gif`).

For compound types the proposed `NCF_sdhash` tool can reduce the number of false positives significantly (similar results were obtained for `pdf`s and `jpg`s). `NCF_sdhash` worked particularly well for small *N*s. However, more research is needed to see if this holds for larger sets, too.

For flat types the removal of common features had almost no impact as most of their contents are user-generated data by design. However, `NCF_sdhash` may be superficial when processing flat types that contain layout information such as `html` to reduce the impact of template similarity.

For compressed types (images) bytewise approximate matching is not the most efficient tool to detect similarities. Thus, removing features had only little effect on the results.

Further tests are required to see if these statements hold, e.g., expanding the test data by `zip, rtf, bmp, mp3`, and so on.

Common features were also found across different file types (see Sec. 5.4.2.2 and 5.4.2.3). Often these features belong to structure information, color space, font specifications, or even some well-known code used to accomplish a particular task. For our test, we found an overlap among compound files, `pdf`, and `jpg`, where all shared color information related to the embedded images/image.

Given that our test set was small, more comprehensive experiments are needed on larger sets. However, given that *N* is flexible, a user can define what a common block is. A more comprehensive data set also has the advantage that it will contain *all* common features and that it can be used across multiple different scenarios (forensic cases). On the other hand, examiners could build case specific databases to filter out common blocks, e.g., the crime scene consisting of many devices belonging to one company. While it requires additional processing time, it could exclude features not found in a general set, e.g., metadata information, such as the owner, application version, or proprietary file types. Furthermore, template similarity may be reduced.

Table 22 – Number of true positive matches (user-generated content) found by sdhash and NCF_sdhash for the most significant file types.

File type	**Known UGC matches**	sdhash	NCF_sdhash **for** N	
			3	**5**
doc	18	18 (100%)	16 (89%)	16 (89%)
pdf	12	11 (92%)	12 (100%)	12 (100%)
text	25	25 (100%)	23 (92%)	25 (100%)
html	7	7 (100%)	6 (86%)	1 (86%)

RQ2. How does ignoring common features impact the similarity detection (i.e., number of matches)?

For most tests removing the common features had a positive impact on the number of matches, e.g., reducing them from 9,220 to 409 ($N = 3$ and score ≥ 1, see Table 12). It did not work equally well for the different file types but effectively for *compound types* and pdfs. Having less matches will be time saving from an investigator perspective. On the other hand, the quality of our results (true positives) remained similar: having low Ns usually found UGC matches although some few matches were missed. Table 22 presents the number (and percentage) of true positives found by sdhash and NCF_sdhash (for the most promising values of $N = \{3,5\}$) in relation to the total number of known matches (a.k.a. recall rate (DAVIS; GOADRICH, 2006)). As mentioned, due to the high number of matches, a complete analysis is impossible. The values reported in the table correspond to the manually identified matches (as discussed in Sec. 5.4.1.2, paragraph one) which were restricted to: (1) doc/pdf where we analyzed all matches with score ≥ 21, (2) text/html where all matches with scores ≥ 5 were considered, and (3) html where, due to the high number, we sampled 90 matches. While one may say that NCF_sdhash is missing matches, we argue that

- reducing manual labor significantly is most important, and
- that not all evidence has to be found during an initial run (i.e., finding one piece of evidence in 409 has a similar impact than finding three pieces in 9,220.

To conclude, our results indicate that the internal file structure interferes (negatively) in the similarity identification process when the focus is UGC. Another evaluation is provided in the next sections, where we analyze how the score and recommended threshold value of sdhash are impacted by the removal of common features for a subset of the *t5-corpus*.

RQ3. Is there a clear threshold N for which common features are ignored in the data set at hand?

In our experiments, an N-value between 3 and 5 worked best in most classes. It significantly reduced the number of matches and still identified relevant matches. Unfortunately, NCF_sdhash missed a few matches which we considered difficult (e.g., a lot of changes performed over the whole document). This is similar to the results published by Foster, K. (FOSTER, 2012), who referred to blocks that repeat three or more times as common blocks.

Comparing $N = 3$ and $N = 5$: In the latter case, `NCF_sdhash` found more true positive cases but at the cost of some extra false positives. For N-values of 10 or more, we noticed an increase in the false positives, where many were related to template similarity.

RQ4. How does removing common features affect the runtime efficiency?

`NCF_sdhash` negatively impacts the runtime efficiency (processing time) as shown in Sec. 5.4.3.6 due to the db lookups. However, this difference is insignificant (for our sample set) and may be even less if using a more performant database. Furthermore, we argue that processing time is relatively cheap and it is more important to reduce the needed manual labor as discussed in the last section.

5.4.4.1 Differentiating between user-generated and other types of content

One challenge we faced in Sec. 5.4.3.5 was how to treat EXIF information as it can be seen as user-generated, template or application-generated content. Regardless of the category, “EXIF headers [...] can help the investigator to verify the authenticity of a picture” and is valuable evidence (ALVAREZ, 2004). In other words, in the case of blacklisting it should not be ignored. Every camera, user, etc. is unique, and thus there should not be too many images having the same EXIF information. Depending on N or the size of the database, they may not be considered common.

More generally: Before starting to remove the common features, the forensics investigator needs to define the objective of the search, finding (i) user-generated content (UGC); (ii) application-generated content (AGC); or (iii) template content (TC). While we focused on the first (i), there may be cases where the desired matches are related to AGC or TC. Another example is looking for template similarity, where an examiner has to find documents created by a particular company without considering their content. In such a case, every document sharing features related to the template of that company will matter. The N-value should be adapted according to the investigation goals. Next section provides experiments contrasting N and the kind of similarity desired in a search.

5.4.4.2 Other applications

Apart from using the common feature database to remove undesirable features, one may use the it for other purposes, such as:

Assessing random samples quality The database could be used for identifying objects on a device by looking for fragments of it, e.g., parsing unallocated space (GARFINKEL *et al.*, 2010; FOSTER, 2012). If found, the quality of the fragment can now be assessed. In other words, the probability a certain file was on the system is higher if a distinct feature is found.

File type discovery Common features may also be used for file type identification. Given an unknown byte sequence (e.g., disk sector, object fragment), `sdhash` can be used to extract features and compare them to the database. If there are matches, we can correlate them to the file types.

Future work is necessary to assess the significance of such approaches.

5.5 Removing common blocks under different scenarios on digital forensic investigations

This sections explores how the removal of common blocks impacts digital forensic investigations with different goals. More precisely, we measure recall and precision rates for AM. We analyze, for different scenarios, how the similarity detection is impacted and show that many matches that occurred due to application-generated content have their score zeroed out when common blocks are removed. Consequently, the number of matches that practitioners have to deal with is significantly reduced. We also analyzed how the score produced by AM changes and show that many matches with low scores, ignored before, are of interest now.

5.5.1 Research direction, design decisions and implementation

Roussev, V. (ROUSSEV, 2011) found that `sdhash` matches with scores of < 21 contain many irrelevant results and recommended to ignore them (except for text files where the author recommended ignoring scores < 5). By removing the common features, we expect that most of the similar content of two matched files is related to user-generated content, and even low `sdhash` scores present relevant matches. This way, a new interpretation of the score produced by `sdhash` is necessary.

This section addresses the following research questions:

RQ1 How does the removal of common features impact digital forensics investigations for the different classes of similarity?

RQ2 How are precision and recall rates affected by the removal of common features?

RQ3 How is the recommended threshold value of 21 for `sdhash` affected by removing common features?

5.5.1.1 Procedure overview

To answer our research questions and assess how similarity is affected by removing the common features for the different similarity classes, we simulated a digital forensic investigation where a seized media is compared to a database of known files. We used `sdhash` and `NCF_sdhash` to compare the two sets (see Sec. 5.5.2) against each other. We considered three different scenarios:

Scenario I: We are interested in finding file matches related to *UGC* and/or *TC*. Any match related to these types is considered true positive; otherwise, false positive.

Scenario II: Here, we are only interested in finding *UGC* matches, considered as true positives. Matches related to *TC* and/or *AGC* are considered false positives.

Scenario III: This scenario ignores *TC* matches to remove their impact on investigations. Matches related to *UGC* are true positives, and those from *AGC* are false positives.

To determine the similarity class of a match, we manually investigated all matches reported by either `sdhash` or `NCF_sdhash` (score > 0). To perform manual comparisons, we either used the appropriate software (e.g., MS office, specific web browser etc.) and, in case of binary comparisons, we used `Bless`[5].

When the files of a match had no visual similarity, such as common text, pictures, tables, or other elements created by users, we classified it as application-generated content (*AGC*). For template content (*TC*) matches, files need to have the same layout but differ in their content. An example is two `doc` files from the same company where both have identical font specifications, elements disposition, header/footer with the company information, logo, etc., but the content is different.

5.5.1.2 Terms and metrics used for the evaluation

We present here the terms, definitions, and metrics used for the rest of this chapter. The metrics used for the evaluation are based on those used in the field of information retrieval (OLSON; DELEN, 2008).

Score (s): the score returned by the AM function.

Threshold (t): value used to separate matches from non-matches.

Common feature: f, where $|f| > N$ (i.e., a feature f is considered common if it repeats more than N times across different files in a given corpus).

Match: a comparison between two files where the score $s \geq t$.

[5] <https://github.com/bwrsandman/Bless> (last accessed 2019-15-05)

True positive (tp)**:** a match of two similar files.

True negative (tn)**:** a non-match of two different files.

False positive (fp)**:** a match of two different files (false match).

False negative (fn)**:** a non-match of two similar files (false non-match).

Precision: the ratio of the number of relevant results retrieved (tp) to the total number of results retrieved ($tp + fp$), as depicted in Fig. 26.

Recall: the ratio of the number of relevant results retrieved (tp) to the total number of relevant results ($tp + fn$), as depicted in Fig. 26.

F_1 **score:** harmonic mean of recall and precision, combining these two metrics into one that better distinguishes good results (close to 1) from bad ones (close to 0).

$$Precision = \frac{tp}{tp + fp} \qquad Recall = \frac{tp}{tp + fn} \qquad F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

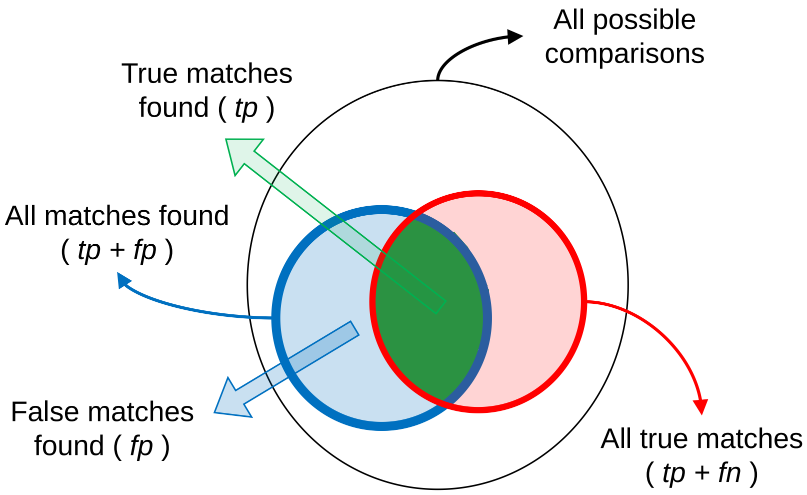

All possible comparisons
True matches found (*tp*)
All matches found (*tp* + *fp*)
False matches found (*fp*)
All true matches (*tp* + *fn*)

Figure 26 – Precision and recall rates for matching evaluation.

5.5.1.3 Common feature database and `NCF_sdhash` implementation

We used the same database and `NCF_sdhash` implementation as the one presented in Sec. 5.4.1.2. Besides, we adopted the same N values for `NCF_sdhash` as before and included an additional one: $N > 2$. From now on, the following nomenclature is used to refer to the different settings of `NCF_sdhash` with respect to N: low ($N > \{2,3,5\}$), mid ($N > \{10,20\}$), and high ($N > \{50,100\}$) values. The database scripts and all tools used in this work can be found in the GitHub page: <https://github.com/regras/cbamf>.

5.5.2 Experimental results

The same *t5-corpus* was used for our experiments. We broke the corpus into two sets: *Known data set* and *Target data set*. The first one was used as the digital forensics investigator database and the second one to simulate a seized media under analysis. The objects of the *Target data set* were compared against the investigator's database to look for similar files. We limited the target set to 100 objects (76.95 MiB) to simplify our manual analysis. For each file type, we randomly selected between 5 and 20 objects. Table 23 summarizes both sets. The complete list of the objects that compose each set can be found in Appendix B.

Table 23 – Number of files per type on both data sets (extracted from *t5-corpus*)

	html	**text**	**pdf**	**doc**	**ppt**	**xls**	**jpg**	**gif**	Σ
Target data set	20	10	20	20	10	10	5	5	100
Known data set	1073	701	1053	513	358	240	357	62	4357

5.5.2.1 Ground truth

Measuring precision and recall rates of the algorithms requires to know the similarity class of each comparison. Thus, we manually compared the 507 unique matches of `sdhash` and `NCF_sdhash` to determine the class of the match. Table 24 summarizes our results.

Table 24 – Number of file matches per similarity class (ground truth)

Similarity class	**Number of file matches**
User-generated content (*UGC*)	45
Template content (*TC*)	93
Application-generated content (*AGC*)	369

It is important to note that we were not interested in measuring the accuracy or detection capabilities of `sdhash`. Instead, we want to evaluate how the removal of the common features impacted similarity detection based on AM functions.

5.5.2.2 Target data set vs. Known data set

Comparing all files from the *Target data set* and *Known data set* required a total of $(4357 * 100 =)$ 435,700 comparisons. Table 25 shows the number of matches for `sdhash` and `NCF_sdhash` for different score ranges.

We can see a significant reduction in the number of matches for `NCF_sdhash`, especially for low N values. The removal of common features reduced the score of many matches; some cases were filtered out completely. For instance, some matches having $s = 100$ for `sdhash` had $s = 0$ for `NCF_sdhash` for $N > \{2,3\}$ (e.g., `002123.html` vs. `002096.html` a *TC* match). More details are provided in Table 26 in which the removal of common features made s decreases as N got lower. Template content matches are challenging to detect and remove since

Table 25 – Number of file matches by score range using sdhash and NCF_sdhash for the sets comparison, discarding common features with occurrences $> N$.

Score	sdhash	NCF_sdhash for N						
		2	3	5	10	20	50	100
= 1	92	8	9	18	13	10	19	18
≥ 1	454	78	103	151	171	188	222	265
≥ 10	187	46	75	105	130	143	143	148
≥ 21	131	28	49	69	98	108	111	112
≥ 50	56	9	18	34	40	54	57	57
≥ 90	20	8	8	14	21	20	20	20
= 100	9	3	3	9	9	9	9	9

they depend on the number of files sharing the same layout stored in the reference database. In our experiments, we had only a few instances of each template available, which is the reason why low N values worked well in removing related features and decreasing the similarity score.

Table 26 – The impact of common features removal on the score of some file comparisons. All cases reported here are Template Content matches.

Query set	Known set		Score (0 - 100) for N						
File	File	sdhash	2	3	5	10	20	50	100
002123.html	002096.html	100	0	0	100	100	100	100	100
000214.html	003083.html	84	2	17	75	84	84	84	84
004338.html	004509.html	81	0	0	0	0	81	80	80
000251.doc	002145.doc	72	0	66	70	71	71	72	72
003751.html	002789.html	62	0	0	0	46	45	61	62
000986.ppt	003662.ppt	11	10	10	10	10	10	10	10
004338.html	000918.html	4	0	0	0	0	4	4	4

Table 27 presents a few examples of matches related to application-generated content. Some matches having high scores reported by sdhash had $s = 0$ for NCF_sdhash (e.g., 002394.doc vs. 004066.doc); the content shared between the matched files was only related to *AGC*. In other cases, removing common features just reduced s (e.g., 001675.pdf vs. 000746.pdf), showing that besides the common structure data, the objects shared some *UGC*. A third case shows that some comparisons (e.g., 001675.pdf vs. 002203.pdf) had higher scores for NCF_sdhash than sdhash, even tough the files were visually different (no *UGC*)[6].

Removing some undesirable features also made *UGC* features prevail and increase the similarity score, as shown in table 28 (e.g., 003049.pdf vs. 003046.pdf). The disposition of the remaining features may have influenced the score[6]. Some matches had about the same scores (e.g., 000380.xls vs. 000397.xls).

Remark. In the first part of this chapter, we found that a few scores had minor changes due

[6] As stated in section 2.4.5.4, sdhash/NCF_sdhash store features into a set of bloom filters (max. of 160 features per filter). The comparison function evaluates the Hamming distances among the filters from each object, selects the maximum value and average all results. We believe that removing some features allowed the matching features to be stored in the same filter (they were more easily stored separately before), increasing s.

Table 27 – The impact of common features removal on the score of some file comparisons. All cases reported here are Application-Generated Content matches.

Query set	Known set		Score (0 - 100) for N						
File	File	sdhash	2	3	5	10	20	50	100
002394.doc	004066.doc	56	0	0	0	0	0	0	0
003047.pdf	001939.pdf	45	0	0	0	0	0	0	0
001675.pdf	000746.pdf	41	0	21	20	21	20	31	32
000698.doc	004419.doc	38	0	0	0	0	0	0	0
001675.pdf	002203.pdf	24	0	63	66	69	71	55	58
000047.xls	000380.xls	21	21	20	20	22	21	21	21
001239.jpg	002627.jpg	17	0	0	0	0	0	9	13

Table 28 – The impact of common features removal on the score of some file comparisons. All cases reported here are User-Generated Content matches.

Query set	Known set		Score (0 - 100) for N						
File	File	sdhash	2	3	5	10	20	50	100
002245.html	002238.html	100	100	100	100	100	100	100	100
003299.pdf	003296.pdf	91	95	95	96	98	90	90	90
003049.pdf	003046.pdf	59	92	92	92	93	54	54	54
000380.xls	000397.xls	41	45	41	50	41	41	41	41
001645.doc	001646.doc	23	33	31	32	31	26	24	25
001329.html	001330.html	5	13	13	13	13	13	6	6
004915.html	004914.html	0	18	18	18	18	18	18	18

to hash collisions since sdhash uses 160-bit SHA-1 as hash function and NCF_sdhash adopts the smaller FNV-1a.

5.5.2.3 Impact on similarity score over different scenarios

Here, we focus on analyzing matches with $s \leq 21$ to measure the impact on threshold t of removing common features. Roussev, V. (ROUSSEV, 2011) recommended $t = 21$ to identify relevant matches. After removing the common features, our hypothesis is that even the matches having $s < 21$ will present significant UGC since most features related to TC and AGC were excluded.

Table 29 and 30 show the number of file matches by score (divided by the matching class). For instance, consider $s \geq 15$: sdhash had 156 matches, where 30 were UGC, 47 AGC, and 79 TC. Based on these results, we calculated precision, recall, and F_1 score for the three different scenarios, as presented in the following sections.

Scenario I - Removing only *AGC*

In this scenario, all NCF_sdhash versions had better results than sdhash (that behaves as if $N = \infty$) for $t \leq 21$ considering precision (see Fig. 27). The best setting was $N > 20$ with many undesired matches being removed and many templates considered as tp due to the

Table 29 – Number of file matches per score and per class - Part I: sdhash and NCF_sdhash ($N > 2,3,5$).

Score >=	**File Matches in the form: *#matches* (*UGC - AGC - TC*)**			
	sdhash	**N >2**	**N >3**	**N >5**
21	131 (29 - 31 - 71)	28 (18 - 6 - 4)	49 (20 - 11 - 18)	69 (25 - 10 - 34)
20	137 (29 - 36 - 72)	31 (19 - 8 - 4)	53 (21 - 13 - 19)	77 (28 - 13 - 36)
19	138 (29 - 37 - 72)	32 (19 - 9 - 4)	55 (21 - 15 - 19)	79 (28 - 15 - 36)
18	141 (29 - 38 - 74)	33 (20 - 9 - 4)	56 (22 - 15 - 19)	82 (29 - 15 - 38)
17	147 (29 - 43 - 75)	36 (21 - 9 - 6)	60 (23 - 15 - 22)	86 (30 - 15 - 41)
16	151 (29 - 45 - 77)	37 (21 - 9 - 7)	63 (24 - 16 - 23)	88 (31 - 16 - 41)
15	156 (30 - 47 - 79)	39 (21 - 9 - 9)	63 (24 - 16 - 23)	89 (31 - 16 - 42)
14	157 (30 - 47 - 80)	40 (21 - 10 - 9)	65 (25 - 16 - 24)	90 (31 - 16 - 43)
13	161 (30 - 50 - 81)	43 (23 - 10 - 10)	67 (26 - 16 - 25)	93 (32 - 18 - 43)
12	170 (31 - 58 - 81)	43 (23 - 10 - 10)	68 (26 - 17 - 25)	96 (32 - 20 - 44)
11	178 (31 - 64 - 83)	43 (23 - 10 - 10)	69 (26 - 18 - 25)	98 (32 - 20 - 46)
10	187 (32 - 72 - 83)	46 (24 - 10 - 12)	75 (28 - 20 - 27)	105 (33 - 25 - 47)
9	197 (32 - 79 - 86)	49 (24 - 12 - 13)	77 (28 - 21 - 28)	110 (33 - 29 - 48)
8	211 (32 - 93 - 86)	52 (25 - 13 - 14)	78 (29 - 21 - 28)	114 (34 - 31 - 49)
7	224 (33 - 104 - 87)	56 (27 - 14 - 15)	82 (30 - 22 - 30)	120 (36 - 33 - 51)
6	242 (35 - 120 - 87)	58 (27 - 16 - 15)	83 (30 - 23 - 30)	124 (36 - 34 - 54)
5	258 (36 - 134 - 88)	59 (28 - 16 - 15)	84 (30 - 23 - 31)	125 (37 - 34 - 54)
4	273 (36 - 148 - 89)	60 (28 - 17 - 15)	87 (30 - 25 - 32)	129 (37 - 37 - 55)
3	304 (36 - 179 - 89)	63 (28 - 18 - 17)	89 (30 - 26 - 33)	129 (37 - 37 - 55)
2	362 (37 - 236 - 89)	70 (30 - 22 - 18)	94 (32 - 29 - 33)	133 (38 - 40 - 55)
1	454 (41 - 321 - 92)	78 (30 - 30 - 18)	103 (33 - 37 - 33)	151 (42 - 52 - 57)
0	507 (45 - 369 - 93)	507 (45 - 369 - 93)	507 (45 - 369 - 93)	507 (45 - 369 - 93)

small number of files sharing the same layout. `sdhash` had the worst results, where the decrease of t had a negative impact due to a large number of fp matches. On the other hand, removing the common features resulted in many undesirable matches being ignored, having a less significant impact on `NCF_sdhash` (except for $N > 100$) when decreasing t.

Considering now the Recall metric (Fig. 28), no algorithm found all similar matches. `sdhash` and `NCF_sdhash` with $N > 20, 50, 100$ had the best results. For this metric, we had a bad influence of templates for `NCF_sdhash` with low N settings. Many template matches were removed from results due to the limited number of models in our database. As N increased, the features related to templates were not considered common anymore, and the matches became relevant again. For $N > 3$, we found only 33/93 template matches, while for $N > 20$ we had 91/93 ($t = 1$).

Fig. 29 shows the results for the F_1 score. We had the best combination between precision and recall for $N > 20$. We could see that `sdhash` results degraded significantly for threshold $t < 21$ due to the high number of false positives (bad precision). On the other hand, low t values increased the performance of `NCF_sdhash` (with low and mid N values). However, mid N values are the ones recommended when template matches are relevant for investigations. Besides, using $t > 0$ showed to be beneficial and should be taken into consideration when working with `NCF_sdhash`.

Table 30 – Number of file matches per score and per class - Part II: NCF_sdhash ($N > 10, 20, 50, 100$).

Score >=	**File Matches in the form: *#matches* (*UGC* - *AGC* - *TC*)**			
	N >10	**N >20**	**N >50**	**N >100**
21	98 (26 - 13 - 59)	108 (26 - 11 - 71)	111 (26 - 13 - 72)	112 (26 - 15 - 71)
20	102 (27 - 14 - 61)	113 (26 - 14 - 73)	114 (26 - 15 - 73)	114 (26 - 16 - 72)
19	105 (28 - 16 - 61)	115 (27 - 15 - 73)	115 (27 - 15 - 73)	116 (27 - 16 - 73)
18	106 (29 - 16 - 61)	117 (28 - 16 - 73)	119 (28 - 17 - 74)	118 (28 - 17 - 73)
17	110 (30 - 16 - 64)	122 (29 - 17 - 76)	123 (29 - 18 - 76)	122 (29 - 17 - 76)
16	114 (31 - 18 - 65)	125 (31 - 18 - 76)	124 (30 - 18 - 76)	123 (30 - 17 - 76)
15	115 (31 - 19 - 65)	127 (31 - 19 - 77)	125 (30 - 19 - 76)	125 (30 - 19 - 76)
14	116 (31 - 19 - 66)	128 (31 - 19 - 78)	127 (30 - 19 - 78)	128 (30 - 20 - 78)
13	118 (33 - 19 - 66)	129 (32 - 19 - 78)	128 (30 - 19 - 79)	133 (31 - 23 - 79)
12	122 (33 - 22 - 67)	134 (33 - 22 - 79)	132 (30 - 22 - 80)	139 (31 - 28 - 80)
11	124 (33 - 23 - 68)	137 (33 - 23 - 81)	137 (31 - 25 - 81)	142 (31 - 30 - 81)
10	130 (33 - 27 - 70)	143 (33 - 28 - 82)	143 (31 - 29 - 83)	148 (31 - 35 - 82)
9	135 (34 - 30 - 71)	147 (34 - 30 - 83)	147 (32 - 31 - 84)	155 (32 - 40 - 83)
8	138 (34 - 33 - 71)	151 (34 - 32 - 85)	151 (32 - 33 - 86)	162 (33 - 44 - 85)
7	142 (36 - 34 - 72)	158 (36 - 35 - 87)	161 (35 - 37 - 89)	173 (35 - 50 - 88)
6	144 (36 - 35 - 73)	159 (36 - 35 - 88)	165 (36 - 40 - 89)	181 (36 - 57 - 88)
5	146 (37 - 36 - 73)	163 (37 - 38 - 88)	172 (37 - 46 - 89)	188 (36 - 64 - 88)
4	148 (37 - 38 - 73)	166 (37 - 40 - 89)	176 (37 - 49 - 90)	212 (37 - 86 - 89)
3	151 (37 - 41 - 73)	169 (38 - 42 - 89)	190 (38 - 62 - 90)	231 (38 - 104 - 89)
2	158 (38 - 46 - 74)	178 (39 - 50 - 89)	203 (39 - 74 - 90)	247 (39 - 119 - 89)
1	171 (41 - 55 - 75)	188 (42 - 55 - 91)	222 (42 - 89 - 91)	265 (42 - 133 - 90)
0	507 (45 - 369 - 93)	507 (45 - 369 - 93)	507 (45 - 369 - 93)	507 (45 - 369 - 93)

Scenario II - Searching for UGC only

For the second scenario, we are interested in finding only UGC matches. Fig. 30 shows our results regarding precision. Notice that low N values stood out in this experiment since they were responsible for removing many template matches - the lower the N, the better the precision. `sdhash` had the worst results since it detected many *TC* matches as relevant, once it does not distinguish the class of similarity. All algorithms had low precision values mostly due to templates, which were harder to remove even for `NCF_sdhash` (see tables 29 and 30). In most cases, decreasing to $t = 1$ had a small impact on precision.

The recall rates are shown in Fig. 31. The worst results were obtained for low N values, in which we could not identify a few similar files with too many changes (differences). Besides, we found a particular case where six comparisons of `html` files were between identical objects in our database. This way, all features related to them became common for some `NCF_sdhash` settings; $N > 2$ and $N > 3$ produced $s = 0$, while `sdhash` and others had $s = 100$. As t decreased, we had similar results for `sdhash` and other `NCF_sdhash` settings. For recall, it is worth to accept small scores since many additional matches were found; most cases reached more than 90% at $t = 1$, while for the recommended $t = 21$, they had about 60%.

Given the F_1 score results (Fig. 32), we can conclude that for scenarios where tem-

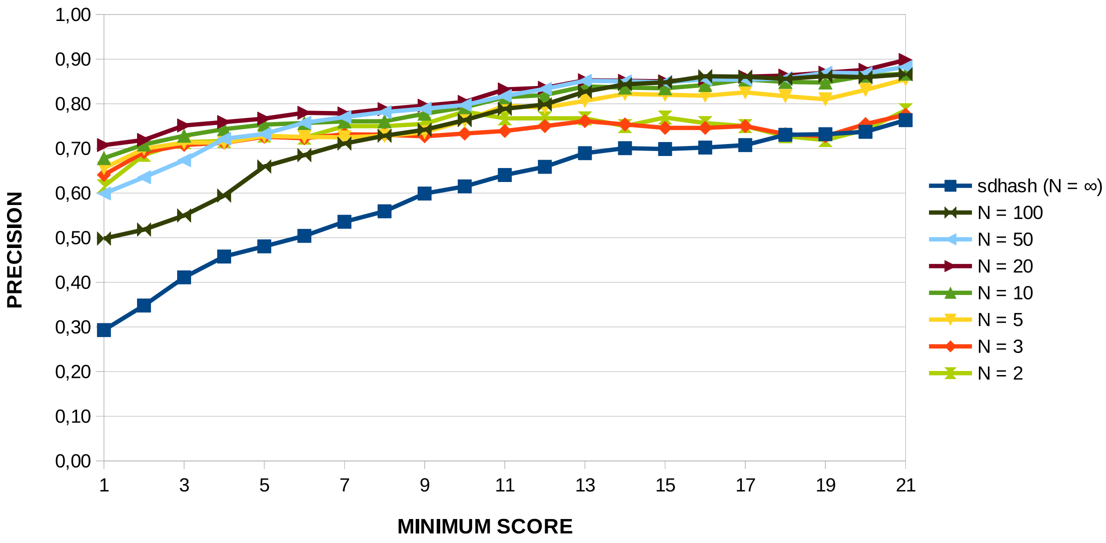


Figure 27 – Scenario I: Precision vs. score

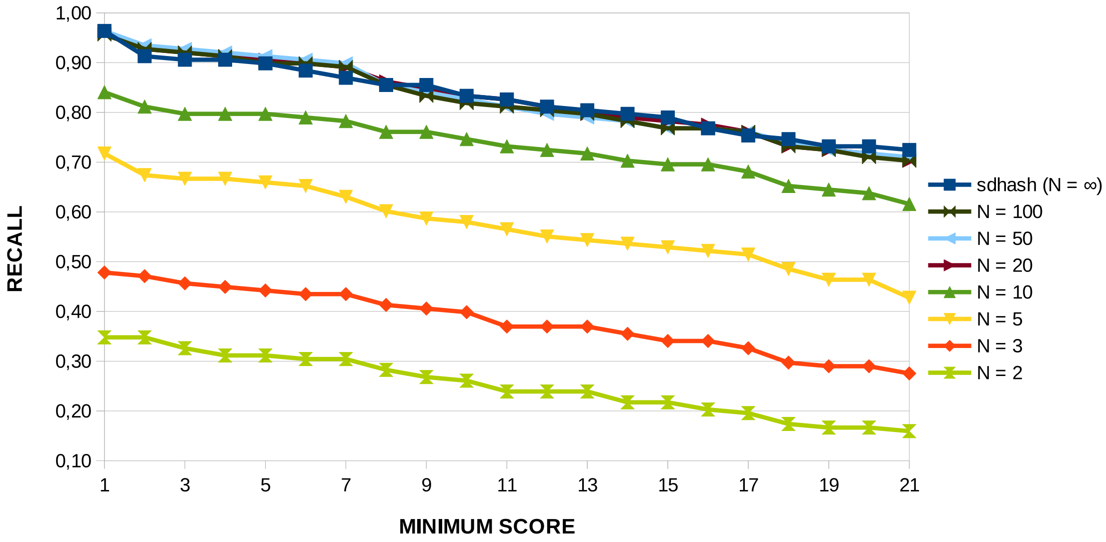


Figure 28 – Scenario I: Recall vs. score

plates matches are not relevant, low N values are recommended due to its good balance between precision and recall. sdhash had the worst results which degraded significantly for $t < 21$. For NCF_sdhash with low/mid N settings, it is worth looking for matches with $t \geq 1$ since even low values tend to present relevant matches.

Scenario III - No template matches

The third scenario does not consider template matches and seeks to analyze how they influenced precision and recall rates. NCF_sdhash was superior regarding precision (Fig. 33). Normally, the lower the N, the better the precision. Low N values in NCF_sdhash are more prone to remove *AGC* (see tables 29 and 30) since many features repeating in a few

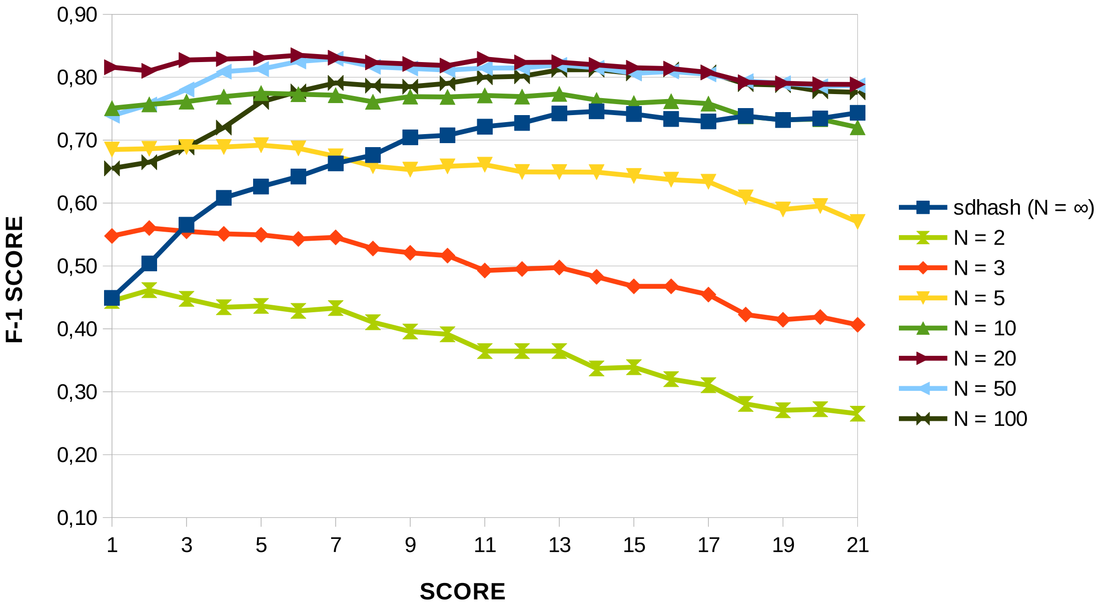


Figure 29 – Scenario I: F_1 vs. score

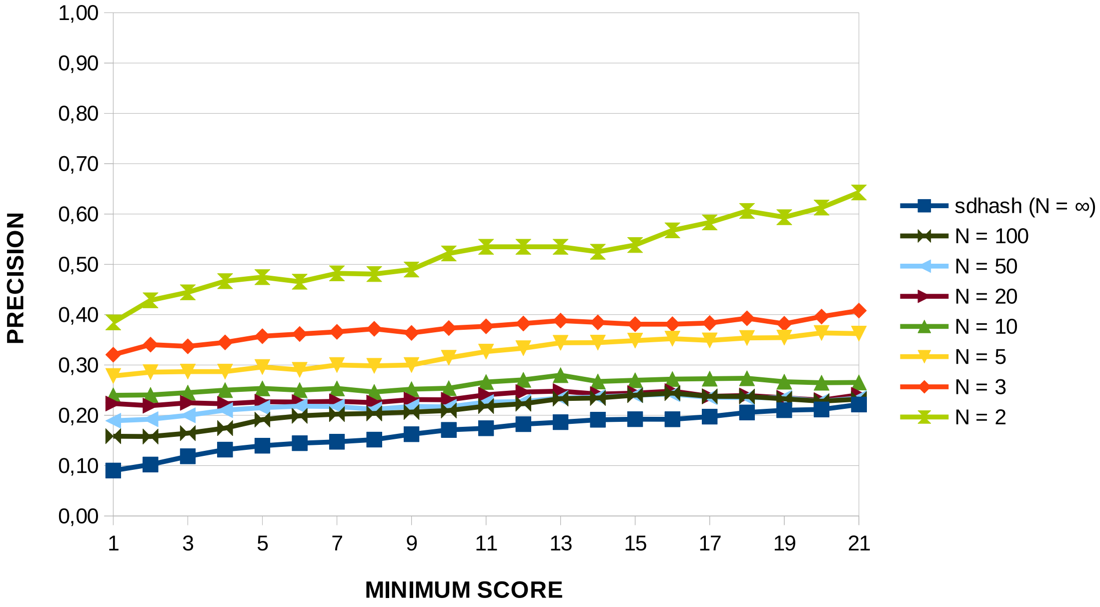


Figure 30 – Scenario II: Precision vs. score

files are considered common. An example is a match of two pdf files where sdhash and some NCF_sdhash settings detected as similar, but the files were different. By using $N > 2$ or $N > 3$, we could remove many features shared by those files and with a few other ones (e.g., a feature related to a font specification) and have $s = 0$. As we decrease t, sdhash results dropped from 50% to 10%, a significant degradation on its performance. Lower values of N presented a less aggressive degradation on these precision results due to the small number of false positives compared to sdhash.

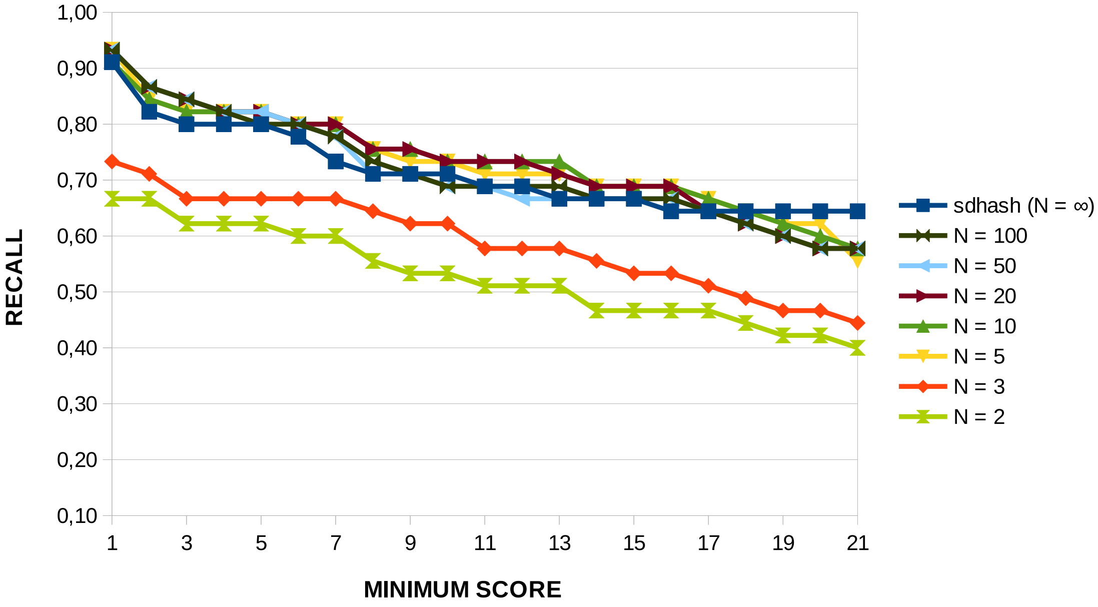


Figure 31 – Scenario II: Recall vs. score

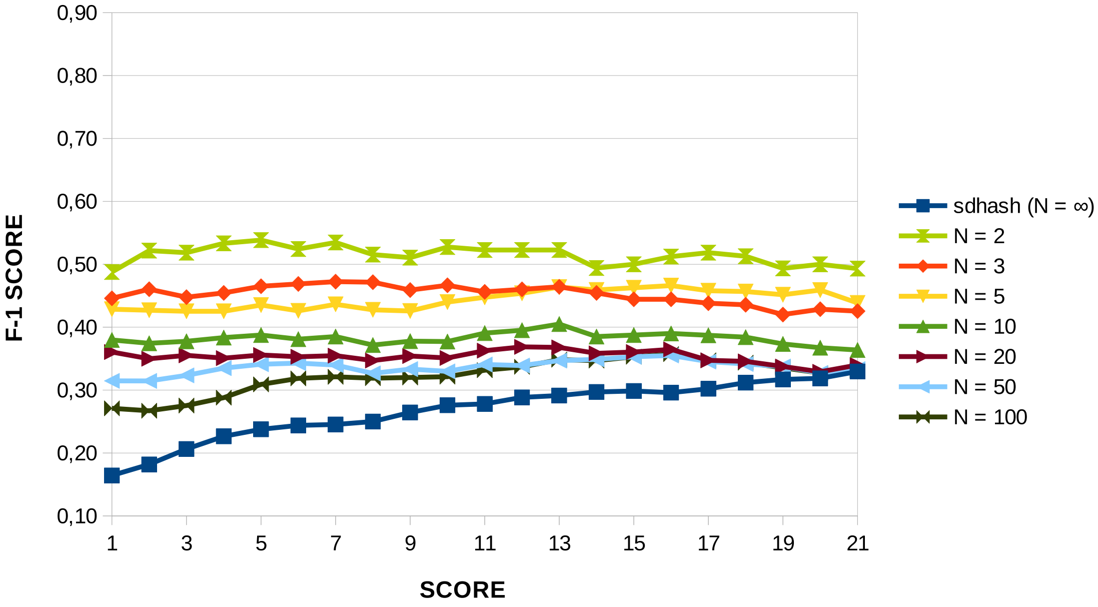


Figure 32 – Scenario II: F_1 vs. score

For recall (Fig. 34), we had the same results as scenario II since in both cases we ignored TC matches. Finally, the results of F_1 score (Fig. 35) for decreasing t showed a poor performance of `sdhash` again, while all settings of `NCF_sdhash` with low/mid N values had similar/better results. For these settings, no significant degradation was noticed for low t values, and considering them as relevant results can be beneficial.

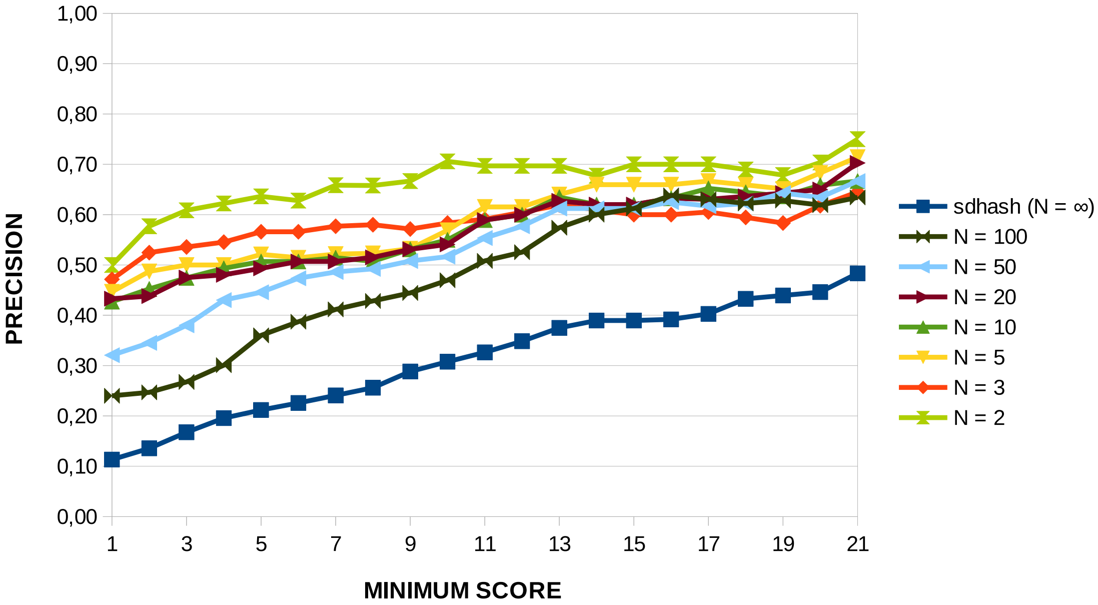


Figure 33 – Scenario III: Precision vs. score

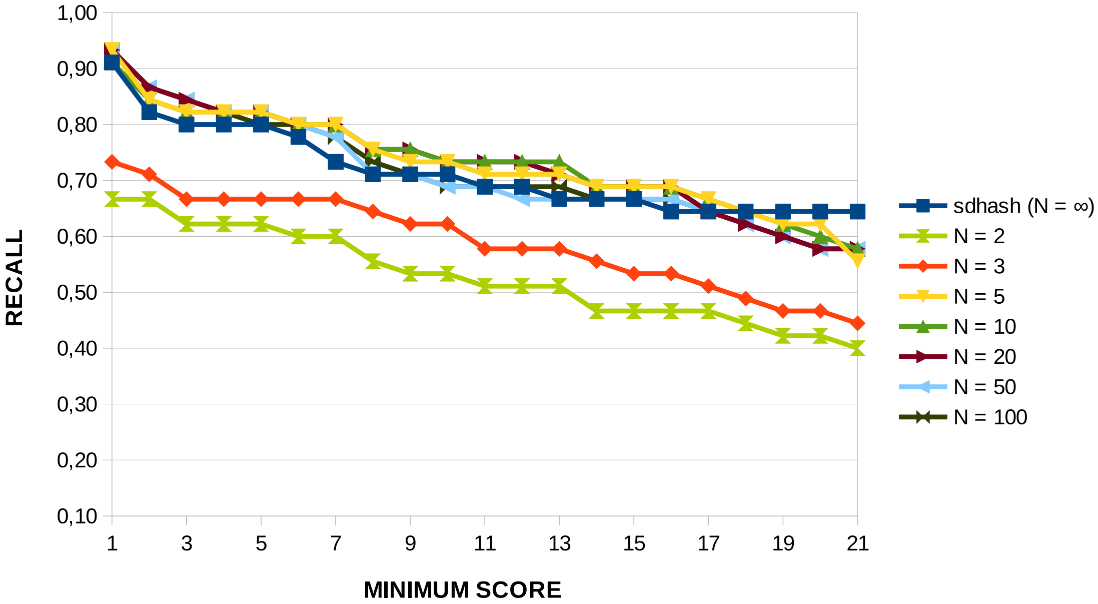


Figure 34 – Scenario III: Recall vs. score

5.5.3 Discussion

Based on the experiments described in the previous sections, we discuss here the lessons learned, specifically, the correlation of removing common features and the similarity score with respect to the different scenarios considered in our experiments.

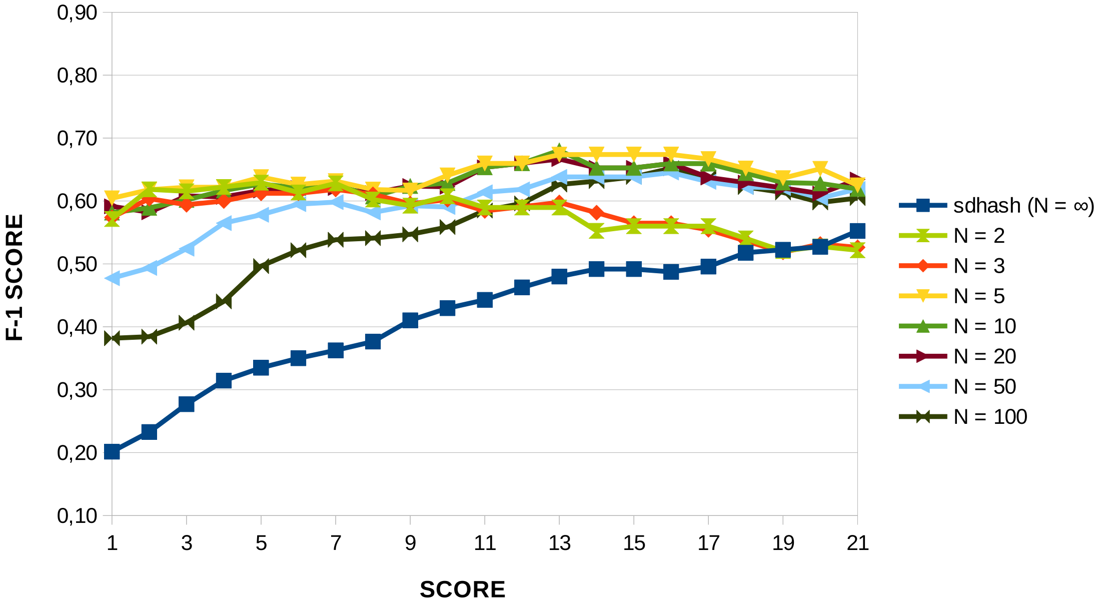


Figure 35 – Scenario III: F_1 vs. score

RQ1. How does the removal of common features impact digital forensics investigations for the different classes of similarity?

By removing the common features, many AGC matches were filtered out. TC matches were a problem for low N-values in NCF_sdhash, as well as finding a few UGC matches of high degree of complexity (too many modifications on the files content). Mid and high-values of N in NCF_sdhash had similar or better results than sdhash in all cases.

With respect to template matches, we had many instances of a few files sharing the same layout in our database. Low N-values in NCF_sdhash removed many of these matches from the results but also kept some of them. Consequently, scenario I and II were impacted negatively, where we could neither identify nor remove all TC matches effectively. We can confirm this assumption by observing the increase in recall (scenario I vs. scenario III) and precision (scenario II vs. scenario III). In the first case, recall dropped significantly since many templates were removed, while in the second case, some TC matches that we could not remove from the results were detected as fp.

NCF_sdhash with low N settings (except for $N > 5$) also underperformed in the detection of a few matches between files with too many modifications (differences). Besides, some identical files found in the database contributed for degradation in results (see Sec. 5.5.2.3). sdhash and all settings with $N > 5$ had more than 91.00% of recall. NCF_sdhash using $N > 2$ and $N > 3$ had 66.67% and 73.33%, respectively. The reason for higher recall rates of sdhash and most versions of NCF_sdhash is due to the common features. By removing them, some instances had $s = 0$ since the number of features related to UGC were too small or nonexistent. In such cases, the match may happened because common features were still present in the similarity assessment.

In short, sdhash tended to detect many matches related to template/application-generated content and NCF_sdhash with low N-values is inadequate for template detection or comparisons with a high level of complexity which do rely on the common features. Although mid and high values of N could perform well for template detection in this particular data set, future work is necessary to separate this sort of match since it is hard to know the number of files sharing the same template in a large set.

RQ2. How are precision and recall rates affected by the removal of common features?

In general, removing the common features increased precision, and for low N values, decreased recall. sdhash performed the worst regarding precision in all scenarios due to the presence of the common features. NCF_sdhash had the best results for low N values, except for scenario I, where mid-values of N had the best results for detecting many TC matches. For recall, mid and high-values of N had similar/better results than sdhash. Low N-values for NCF_sdhash performed worse, especially for scenario I where many TC matches were missed. Given the F_1 score (balance between precision and recall), we can see that sdhash performance dropped significantly in all scenarios with the decrease of t. On the other hand, low/mid N values of NCF_sdhash presented the best results as t dropped to 1.

In our experiments, using $N > 20$ had better results than higher N values ($N > 50$ or $N > 100$) for all cases. We believe that no further benefit is achieved for higher values of N. The number of true positives was about the same with a significant reduction in the number of false positives. Besides, when considering template similarity as a relevant result (scenario I), $N > 20$ should also be the one adopted given its better precision and recall rates.

RQ3. How is the recommended threshold value of 21 for sdhash affected by removing common features?

Our experiments revealed that many matches were left out by using $t = 21$ for sdhash. From the 45 UGC matches, only 29 were found. Using $t = 1$ allowed us to increase the number of UGC to 41 at the cost of many additional AGC matches (321/369). TC were also benefited from using lower t values (92/93). Although recall increased, precision dropped significantly for the threshold reduction. Besides, F_1 score showed that it is not worth using $t < 21$ for sdhash due to the significant degradation in its performance.

On the other hand, NCF_sdhash showed improvements when choosing lower threshold values. The best performances of all settings were with $N > 20$, where many UGC and TC were found (42/45 and 91/93, respectively) at a small cost of AGC matches (55/369) compared to sdhash. F_1 score values showed minor improvements when using low t values. For this reason, we believe that low thresholds should be considered for digital forensic investigations given that many relevant matches were found. For $N > 2$, the best F_1 value was using $t = 2$ (scenario

I) and $t = 5$ (scenarios II and III), while for $N > 20$ we had $t = 6$ (scenario I), $t = 12$ (scenario II), and $t = 13$ (scenario III).

Remark: The accuracy, defined as $(tp+tn)/(tp+tn+fp+fn)$, is not presented here since `sdhash` and all `NCF_sdhash` versions had similar values ($> 99.00\%$) due to the enormous number of true negatives pointed out by both algorithms.

5.6 Limitations

This work comes with three limitations. First, bytewise approximate matching algorithms do not work equally well for all file types, since our experiments showed that common features depend on the file type. Second, our experiments included a lot of manual testing and random sampling. It is possible we misclassified matches or we ended up with poor samples drawn from the data set. Lastly, our results are not of general nature but only valid for our test set. For instance, for the t5-corpus, a $3 \leq N \leq 5$ works well; however other sets may require a different value. Future work is necessary to confirm if these values work for larger data sets.

5.7 Conclusions

In this chapter, we analyzed the impact of common features (and their exclusion) for approximate matching. We first explained what common features are and showed that they are shared across files of the same and different types, often relating to application-generated content or template similarity. In the sequence, we removed the common features to observe their impact on the amount and quality of matches. By excluding the less important features, the number of matches was significantly reduced with an acceptable loss in the similarity detection; in some cases, we obtained approximately 87% fewer matches compared to traditional tools. This practice also benefited precision and recall rates, where different settings aided each metric differently based on the goal of the investigation. Finally, we also analyzed the impact on the threshold score of AM and showed that, after removing the common features, all scores from 0 to 100 produce relevant matches for a small cost in the number of false positives.

As next steps, we want to verify if the results got here with `sdhash` will sustain when changing the AM tool to `mrsh-v2`. Besides, we want to explore different database solutions or create a customized structure to store common blocks to improve efficiency. Additionally, other data sets need to be analyzed to confirm if the N-values used in this work can be adopted universally, as well as testing different file types, such as `zip`, `rtf`, `bmp`, `mp3`, and so on. Although this thesis focused on removing common blocks, we highlight that this sort of data could be useful in other scenarios. We plan to study how we could use a template or application-generated content to the benefit of digital forensic investigations. For instance, we could apply common blocks concepts to the file type discovery or assessment of random sample quality.

6 Mitigating sdhash limitations with Jaccard similarity

In this chapter, we present a theoretical analysis of the similarity detection capabilities of one of the most popular AM tools, the `sdhash`, and show some improvements regarding the similarity score produced by it. By changing the comparison function of `sdhash`, the new tool presented more precise scores and shows the real amount of similarity shared between objects; besides, its results are easier to interpret.

6.1 Introduction

One of the most popular and used AM tools is `sdhash`, because of its good detection capabilities, usability, and for being open source. Many researchers have target `sdhash` and tried to understand its benefits and limitations when applied to digital forensic investigations. However, most of the tests using `sdhash` focused on empirical analysis using real-world/controlled data to make estimations about the tool internals. The problem is that no theoretical analysis was performed so far, showing, for example, the minimal amount of similarity shared between objects to be highlighted by `sdhash`, in which scenarios the tool would produce reliable results, and so on.

Another problem with `sdhash` is that the score produced in the similarity comparison process is difficult to understand. Roussev, V. (ROUSSEV, 2011) suggests that every comparison with $score \geq 21$ should be taken as similar; otherwise, the chances for a false positive is too high, and the match should be ignored. The only exception is for text files, which are considered similar for $scores \geq 5$. When looking at the score produced (ranging from 0 to 100), it is natural to interpret it as a percentage value of similarity; however, Roussev argues that such interpretation is wrong. Also, we do not have a clue about which object is the most similar one in case many objects present similarity; the only classification is being similar and not similar. Furthermore, `sdhash` does not separate resemblance from containment results, leading practitioners to wrong interpretations in some contexts.

Given the problems mentioned above, in this chapter, we aim at providing the following contributions:

- A theoretical analysis of `sdhash` (that could be extended to other bloom filter-based tools, such as `mrsh-v2`), showing its detection capabilities and limitations;
- A new AM tool, called `J-sdhash`, which is an improved version of `sdhash` using the Jaccard distance to assess similarity.

In the following sections, we present our research questions, theoretical analysis, and the new `J-sdhash` tool with tests corroborating the improvements made. Details about the working process of `sdhash`, required to the understanding of all contributions of this chapter, can be found in Sec. 2.4.5.4.

6.2 Research direction, forms of similarity, and data set used for the experiments

6.2.1 Research questions

This chapter focus on answering the following research questions:

RQ.1 What is the minimum amount of similarity that `sdhash` can detect?

RQ.2 What to expect from `sdhash` when assessing similarity? In which cases/scenarios will it work?

RQ.3 What is the impact of the α variable (and its default value) on the similarity assessment?

RQ.4 Can we improve the similarity assessment process of `sdhash` to make it easier to understand and accurate?

6.2.2 Forms of similarity

The similarity between objects can be found in many forms. In the following, we present some scenarios related to the kind of similarity one can find when comparing two objects. We will consider these scenarios for the theoretical evaluation of `sdhash`.

1. **New version of objects:** The most common case of similarity is the one where we have an object and a new version of it with a few changes. For instance, a document and a revised version of it where some typos were fixed or new lines were added.
2. **Block of similar content:** We may found cases where two different objects have a block of similar content (e.g., a paragraph, embedded picture, table, etc.). In such a case, we can have two variations:
    a) **Single block:** Only one similar block of content is shared between the objects, placed at the beginning, ending, or middle of them.
    b) **Multiple blocks:** Many blocks of similar content (with the same or different sizes) are shared and spread all over the objects.

3. **Object fragment:** The last scenario correlates an object and a small piece of it, such as a fragment. For instance, we could verify if a certain picture is embedded into a `doc` file or, given a single book page, we can verify to which book that page belongs to.

In all of these scenarios, the size of the objects under comparison influences the results. We have comparisons of equal (or very close) object sizes and also comparisons of objects of very different sizes. The AM tools are usually tested in all of these scenarios and variations of it. FRASH framework (BREITINGER *et al.*, 2013) proposes four tests to evaluate AM functions simulating all of these similarity forms; the tests are discussed below with their respective purposes, covering most of the similarity forms presented here. Note that only scenario 2(b) was not covered directly on the tests for being a variation of 2(a).

- **Random noise resistance (sim. form: 1):** Given two identical objects F and F', what is the minimum number of bytes that need to be changed all over F' (in a random way) to receive a non-match?
- **Single common block correlation (sim. form: 2(a)):** Given two different objects F_x and F_y that share only a single common piece (bytes) of data, what is the smallest size of the common data for which the similarity tool can reliably correlate the two targets as similar?
- **Alignment robustness (sim. form: 2(a)):** Given two identical objects F and F', what is the minimum number of bytes that need to be inserted at the beginning of F' to receive a non-match?
- **Fragment detection (sim. form: 3):** Given an object F and a piece/fragment of it ($Frag$), what is the smallest size of $Frag$ for which the similarity tool can reliably correlates the fragment to its source?

FRASH framework presents an empirical evaluation of some AM tools using the tests mentioned above. In our work, we seek to assess the theoretical capabilities of AM functions, in particular of `sdhash`, focusing on its limitations given each test. Our theoretical evaluation is towards a future practice of this kind of analysis on the AM field, which we believe is of extreme importance. Although we consider tests with controlled and real-world data necessary, we also feel that the AM field lacks more theoretical analysis showing at least the basic constraints of the tools.

6.2.3 Use of synthetic data set to enhance evaluations

Data sets composed of real-world data have the advantage of being the kind of data found during digital forensic investigations, with different file types that have unique characteristics; some types may favor or hinder the analysis of AM tools. For instance, bytewise tools are good for assessing similarity of `text`, `compound files`, `pdf`, among other types, but they do not perform well when dealing with `jpg`, `gif`, and other compressed files.

When assessing the detection capabilities of AM, although real-world data is preferable, it has the limitation that it is hard to know the real similarity shared among objects in a large and representative set. On the other hand, we can use synthetic data sets instead, since they are easily controlled and we can adjust them as we need. In this work, we create our synthetic data set using data provided by the `/dev/urandom` library to compare the comparison function proposed in this chapter with the one already in use by `sdhash`. By using a controlled data set, we can evaluate which function produces results closer to the real similarity of objects.

We create several objects composed of random data, controlling their sizes and the amount of similarity shared by each pair. We developed a shell script to create our synthetic data set, simulating the many forms of similarity discussed in the previous subsection. The script is available on our GitHub page: <https://github.com/regras/J-sdhash>.

Two different data sets of random data were created to improve our experiments. The first one is used in the single common block test and consists of three different sets of object pairs, where each pair has a similar block (corresponding to p_s percentage of their size) placed at the beginning, middle, or ending of the objects. We varied the percentage p_s according to the following values: 90, 80, 70, 60, 50, 40, 30, 20, 10, 5, and 1%. The chosen object sizes were: 10 KiB, 512 KiB, 1 MiB, and 5 MiB. We used these values for size since they represent different categories of objects: small objects, the average size of the *t5-corpus* (see Sec. 4.3), usual object size of multimedia files, and large objects.

The second data set created is used for the random noise resistance experiment. In this set, for each object, we have a duplicate of it with *r%* of randomly modified bytes; the modifications are randomly chosen and include one of the following ones: Insertion, Substitution, Removal, Swapping, and Replacement. We used the same object sizes as before (10 KiB, 512 KiB, 1 MiB, and 5 MiB) and others based on the number of bloom filters an object has. Since we are dealing with random data, we used the constant of 9650 bytes for defining the size of an object having a single bloom filter (see Sec. 6.3.1); this way, the chosen sizes (number of bloom filters) ranged from 01 to 50 and included the following ones: 55, 60, 65, 70, 75, 80, 85, 90, 95, and 100 bloom filters (object sizes vary from 9650 to 965000 bytes). The percentage r varies according to the values: 0.5, 1.0, 1.5, 2.0, 2.5, 5.0, 10.0, and 25.0%.

6.3 Theoretical analysis of sdhash

The usual mainstream for newly proposed AM tools is as follows: The digest generation process is presented with some concepts about how it works for detecting similarity, followed by a discussion of the digest comparison function. Next, the tool's capabilities are assessed by empirical tests (sometimes using the FRASH framework (BREITINGER *et al.*, 2013)). Although we think this sort of test is of extreme importance, we believe the AM field lacks a more deep and complementary theoretical analysis of the detection capabilities of the tools.

A theoretical analysis would allow us to understand the tools' limitations in a way that just empirical tests may fail (or present incomplete information) due to the data set at hand not being vast and representative enough. In this chapter, we focus our analysis on `sdhash` for being one of the most known and used AM tools. We present details about `sdhash` features, some characteristics of this tool, and what to expect when comparing two objects. We also analyze different scenarios and show how the tool behaves and what are its constraints. In the end, we show some limitations of `sdhash` and propose a new tool aiming to mitigate them.

Definitions For the rest of this section, we will use the following terms:

F_x**:** the x-th object of a given data set.

$bf_{x,i}$**:** the i-th bloom filter of object F_x.

$|bf_x|$**:** number of bloom filters of object F_x.

f_{max}**:** maximum number of features inserted into a single bloom filter (default: 160).

$|F_x|$**:** number of features of object F_x

g_i**:** number of features of bloom filter i.

$G(bf_i, bf_j)$**:** function that returns the number of features in common between bf_i and bf_j.

e_i**:** number of bits set to one in a given bloom filter i.

$e_{i,j}$**:** number of common bits set to one between bloom filters i and j.

e_{min}**:** minimum number of overlapping bits between bf_i and bf_j, given by Eq. 2.6.

e_{max}**:** maximum number of overlapping bits between bf_i and bf_j, given by Eq. 2.5.

C**:** cutoff point, defined by Eq. 2.7.

m**:** bloom filter size (bits).

$score$**:** similarity score, calculated by Eq. 2.9.

6.3.1 Object size and its relation to the number of object features

The `sdhash` digest size varies between 2.6% (ROUSSEV, 2010) to 3.3% (BREITINGER; BAIER, 2013) of the input object size. Breitinger, F. et al. (BREITINGER *et al.*, 2014c) state that there is a relation between the number of features selected to represent objects and their respective sizes, estimating 160 features for every 10 KiB of data (using `sdhash` default parameters); this means that, on average, a single feature is selected to compose the object digest for every 64 bytes. To confirm such a statement, we used `sdhash` to create digests for a real-world data set (*t5-corpus*[1]) and our controlled data set (random data, Sec. 6.2.3). The statistics for both sets can be seen in Table 31.

Table 31 – Object bytes per feature on real-world and controlled data sets

	t5-corpus									Random data
	html	**text**	**pdf**	**doc**	**ppt**	**xls**	**jpg**	**gif**	**ALL**	
Total size (MiB)	68	234	603	220	352	277	54	14	1823	1139
#features (10^5)	9	38	100	30	56	52	9	2	297	200
Obj. bytes/feature	78	64	63	78	66	55	61	63	64	60

Breitinger's statement was based on the *t5-corpus* as well. Although this is a good approximation for dealing with objects of different types, we need to keep in mind that the statistics change from one file type to another. For instance, `doc` files had the largest number of object bytes per feature; Roussev, V. (ROUSSEV, 2010) alerts that one of the reasons for this high number in comparison to other types may be due to about 8% of a `doc` file content being composed by only 0s. Since `sdhash` discards low entropy features, many features are filtered out, increasing the distance between the selected ones. On the other hand, the statistics for random data (including `jpg` and `gif`) are smaller and closer to the general value of 64. The reason for a smaller value is due to randomness; a variation on the feature bytes in random data is more likely to occur than other types having user-generated data (which follows the distribution of alphabets, a more well-behaved one with many repeating characters). This way, features with low entropy are more likely to be encountered, decreasing the average value of the popularity rank (R_{pop}) of features during selection (see Sec. 2.4.5.4.1).

For `xls`, we believe that the low relation of object bytes and the number of features is due to this sort of object has content mostly restricted to a subset of the ASCII set, composed by numbers and signals. The consequence is a decrease in the entropy score of most features and an increasing the number of the selected ones. On the other hand, `html` type had fewer features per byte than other types (except for `doc` files, which had the same value). For `html`, we attribute the results to the common elements present in this sort of object; for instance, tags, JavaScript code, style information, among others, may have contributed to many features being discarded for being duplicate.

[1] <http://roussev.net/t5/t5.html> (last accessed: Nov 30, 2019).

Assuming that for every 64 bytes a feature is selected, we can estimate the number of features ($|F_x|$) of object F_x given only its size with Eq 6.1. Note that this equation provides the number of selected features, i.e., those that will be inserted into the bloom filters. For more accurate results, substitute the 64 value for the one in Table 31 according to the corresponding file type.

$$|F_x| = \frac{s(F_x)}{64} \quad (features) \tag{6.1}$$

Here, $s(F_x)$ returns the object size (in bytes). Besides, we can also estimate the number of bloom filters $|bf_x|$ an object F_x with $|F_x|$ features has using Eq. 6.2. The estimation is performed considering `sdhash` current implementation.

$$|bf_x| = \begin{cases} 0 & if \quad (|F_x| < 16) \\ 1 & if \quad (|F_x| \geq 16 \quad and \quad |F_x| \leq 160) \\ \left\lfloor \frac{|F_x|}{f_{max}} \right\rfloor + 1 & if \quad ((|F_x| \quad mod \quad f_{max}) \geq 20) \\ \left\lfloor \frac{|F_x|}{f_{max}} \right\rfloor & \text{otherwise} \end{cases} \tag{6.2}$$

The first condition describes the case where an object does not have enough features for creating a digest; the second, a case of an object having only a single bloom filter that represents its digest. The third one means that an additional bloom filter is only created if at least 20 features are available, while in the fourth, we have objects with many features ($|F_x| > 160$), but less than 20 were available for insertion in the last bloom filter. In this case, the exceeding features are dropped out. For more details, see Sec. 6.4.

The information got from Eqs. 6.1 and 6.2 are used for our theoretical analysis of `sdhash`, presented in the next sections.

6.3.2 Understanding sdhash features

To understand the similarity computed by `sdhash`, we need to understand which features are selected to create the object digest. Here, we are interested in knowing the content of the features and its relation with the entropy. Roussev, V. (ROUSSEV, 2010) performed empirical tests with `sdhash` to set some parameters in the algorithm, including defining the entropy values in which the so-called *weak features* are classified and later discarded. In this section, we present the characteristics of those features. In the filtering process (Sec. 2.4.5.4.1), features with entropy scores ≤ 100 and those > 990 (on a scale of 0 to 1000) are removed.

A feature is composed of a sequence of β bytes, and its entropy takes into consideration the probability of encountering each byte value in this sequence. The more bytes that are different from each other in a single feature, the higher the entropy value; this way, having all different bytes means maximum entropy (i.e., 1000), while having the same byte β times give us the minimum entropy (i.e., 0). The feature bytes and entropy relation can be better understood by the following relation:

$$entr(f_1 = (b_0, b_0, b_0, ..., b_0, b_0, b_1, b_1, b_1, b_1)_\beta) < entr(f_2 = (b_0, b_0, b_0, ..., b_0, b_0, b_1, b_1, b_2, b_2)_\beta)$$
$$< entr(f_3 = (b_0, b_0, b_0, ..., b_0, b_0, b_1, b_1, b_2, b_3)_\beta)$$

where function $entr(f_i)$ returns the entropy value of feature f_i in a normalized score (0 - 1000), and bytes b_0 to b_3 represent the bytes that compose the features.

However, one may ask: What is the minimum number of different bytes that makes `sdhash` to select a feature to compose the object digest? Considering the default feature size of `sdhash` (64 bytes), we present here some examples of features (and their entropy values) that would be selected. We compute the entropy for some artificial features that we created considering the formulas presented in Sec. 2.2, where we varied the bytes of each feature and observed the impact in the entropy value. Observe that the letters composing the features represent any byte value; in this example, we present the variations with the minimum entropy necessary for a feature to be selected.

- $entr(f_1 = (b_0, b_0, b_0, ..., b_0, b_0, b_1, b_1, b_1, b_1, b_1, b_1, b_1, b_1, b_1, b_1)_{64}) = 104$ (sequence of 54 'b_0' and 10 'b_1')
- $entr(f_2 = (b_0, b_0, b_0, ..., b_0, b_0, b_0, b_1, b_1, b_1, b_1, b_2, b_2, b_2, b_2)_{64}) = 111$ (sequence of 56 'b_0', 4 'b_1', and 4 'b_2')
- $entr(f_3 = (b_0, b_0, b_0, ...b_0, b_0, b_0, b_1, b_1, b_1, b_2, b_3, b_4)_{64}) = 102$ (sequence of 58 'b_0', 3 'b_1', and 'b_2, b_3, b_4')
- $entr(f_4 = (b_0, b_0, b_0, ..., b_0, b_0, b_0, b_1, b_2, b_3, b_4, b_5, b_6)_{64}) = 115$ (sequence of 58 'b_0' and '$b_1, b_2, b_3, b_4, b_5, b_6$')

Note: As the number of different bytes in a feature increases, the entropy becomes higher and requires fewer changes in comparison to blocks of the same byte. For instance, consider a feature f_0 with 64 b_0 bytes ($entr(f_0) = 0$); the feature f_1 shown in the example above has a higher entropy (104) than f_0 for having 10 bytes modified (a block of bytes b_0 was replaced by a block of bytes b_1); however, f_4 has fewer changes compared to f_1 (only six) and yet has a higher entropy because all of the six bytes changed are different from each other.

Not only low entropy features are discarded, but also those having near maximum values, such as the ones with entropy values > 990. Some examples of features with high entropy values are presented below.

- $entr(f_5 = (b_0, b_1, b_2, ..., b_{57}, b_{58}, b_{59}, b_{60}, b_{60}, b_{61}, b_{61})_{64}) = 989$ (sequence of 62 different bytes and duplicated 'b_{60}' and 'b_{61}')

- $entr(f_6 = (b_0, b_1, b_2, ..., b_{58}, b_{59}, b_{60}, b_{61}, b_{61}, b_{61})_{64}) = 987$ (sequence of 62 different bytes and 2 copies of 'b_{61}')
- $entr(f_7 = (b_0, b_1, b_2..., b_{59}, b_{60}, b_{61}, b_{62}, b_{62})_{64}) = 994$ (sequence of 63 different bytes and duplicated 'b_{62}')
- $entr(f_8 = (b_0, b_1, b_2..., b_{60}, b_{61}, b_{62}, b_{63})_{64}) = 1000$ (sequence of 64 different bytes)

Given the above examples, we see that `sdhash` only removes features that bytes are all different or have at most one duplicated byte. If two or more bytes repeat over the feature, its entropy value will be smaller than the limit, and the feature will not be classified as weak.

In conclusion, `sdhash` requires a minimum of ($[6 * 100/64] =$) 9% and a maximum of ($[62 * 100/64] =$) 97% of different bytes in the feature content to consider it as relevant and to make it a candidate to be part of the similarity digest (although the feature selection process still must occur).

6.3.3 Estimating the similarity of bloom filters

The similarity score of `sdhash` is computed by comparing and averaging the digest (a set of bloom filters) of two objects. To perform a single bloom filter comparison, `sdhash` uses the formulas presented in Sec. 2.4.5.4.2 and then produces an average value of all bloom filters comparisons. However, one may wonder about the minimum number of bytes (or features) two objects must have in common to their bloom filter comparison produces a significant score. In this section, we focus on providing such analysis. More specifically, we want to answer the following question: How many features should two bloom filters (of `sdhash`) have in common to produce a $score \geq 1$ and/or a $score \geq 21$?

6.3.3.1 Problem definition

Knowing the minimum amount of data (or features) in common between two filters that produce a significant score is a challenging task, because: (1) We do not have control of the elements once they are inserted into the filter; (2) many parameters affect the score estimation. `sdhash` uses Eq. 2.8 to compute the score of two bloom filters; however, estimating the amount of features in common requires figuring out which $e_{i,j}$ (number of common bits) give us a $SF_{score}(bf_i, bf_j) \geq score_{min}$, where $score_{min}$ is the minimum score we are looking for. Solving the equation requires the values of some parameters, such as e_{max} and C, that vary according to the bloom filters, and we can not control them. Both variables require the number of features and bits set in each filter under comparison. Since the number of features is f_{max} in all filters with the exception of the last one (it varies from 20 to 160 in current implementation, see Sec. 6.4) and objects usually have many bloom filters, we will make a simplification and consider that all filters are full (have f_{max} features) since such assumption will not change results significantly (as shown in Sec. 6.3.3.3). To figure out the number of bits set, we present in the next section a technique for estimating it when the number of features in a bloom filter is given.

6.3.3.2 Number of bits set in a Bloom Filter

Measuring the theoretical boundaries of `sdhash` requires knowing the number of bits set to one for all bloom filters of two objects. The number of features in each bloom filter is known; all filters with the exception of the last one have f_{max} features, while the last one varies but can be easily found out by checking the digest header (ROUSSEV; QUATES, 2013).

The number of bits set to one in a filter may vary according to the number of features inserted on it and their content. In general, a filter has f_{max} features, and each one of them sets k bits in the filter; at most $f_{max} \cdot k$ bits are set. However, our experiments revealed that in practice, filters rarely have the maximum value of bits set. `sdhash` inserts features into a filter by hashing it and breaking the result into k parts, where only $log_2(m)$ bits are used to set the positions in the filter; by default, the 160 bits resulting from the SHA-1 hash function are broken into five parts, each having 11 bits used to set the positions within the filter. The hash function space of 2^{160} decreases to 2^{11} possibilities, increasing the chances for collisions in the filter level significantly. This way, features inserted into the bloom filters may present some collisions in any of the k bits set, resulting in fewer bits set to one than the maximum value after all features are inserted. Besides, the more features we insert, the more the probability for collisions. To clarify: A bloom filter collision means that all k hashes used to set bits into the filter for a given element had the same value for a different element; however, by saying bit collision, we meant that any of the k hashes collide (not necessary all of them).

Given the constraints above, we are interested in figuring out how to estimate the number of bits set given the number of features. First, we ran a few tests over a subset of *t5-corpus* (80 randomly selected objects, 10 for each file type) to observe how the number of bits set on the bloom filters were distributed. Table 32 summarizes our results for different file types and the whole subset. In the experiment, we counted the number of bits set to one in all digests, considering only filters with 160 features.

Table 32 – Number of bits set on bloom filters (with 160 features) on a subset of *t5-corpus*

	html	**text**	**pdf**	**doc**	**ppt**	**xls**	**jpg**	**gif**	**all subset**
Max.	682	703	692	697	686	695	679	686	703
Min.	647	633	632	628	638	636	645	641	628
Average	662.05	663.03	663.07	662.42	662.58	663.41	660.53	661.80	662.72
STD	10.77	8.92	8.74	9.01	8.77	9.05	6.96	9.62	8.94

For `sdhash`, the maximum number of bits set in a filter is $(160 \cdot 5 =)$ 800 bits. However, this is not the number we find on practical scenarios since some positions in the filter are set more than once. Table 32 shows that, on average, 662.72 bits are set to one in a full bloom filter when using real-world objects; many of the k parts of a feature happened to set a position within the filter in which a bit were already set (on average, this number corresponds to 17.16%). Although the average obtained here could be used for our estimations given the small variation over different file types, it is incomplete because we still need to figure out the statistics for filters with less than f_{max} features. It is important to highlight that there is no proportion of the number of features of a filter and the number of bits set. The more features we insert into the filter, the more collisions are likely to occur. For instance, inserting 100 features have more changes to have bit collisions than when we insert only ten features. Thus, the remaining question is: How can we estimate the number of bits set for any filter with any number of features ($< f_{max}$)? In the following, we propose the use of linear regression techniques to make such estimates.

Estimating the number of bits set in a bloom filter with linear regression techniques

One alternative we found to estimate the number of bits set in a filter given any number of features inserted on it is by using linear regression techniques. We choose the least square method to make our prediction, where we provide only the number of features, and the method returns the number of bits set. The technique requires a lot of pairs of data (number of features and number of bits set in the filter) to create an equation to predict the value we need. Since `sdhash` bloom filters require at least 20 features to be created (see Sec. 6.4), with the exception of the first filter that supports 16 features or more, we counted the number of bits set for the last filter of every file on *t5-corpus*. We had, on average, 28.20 (std. 5.45) pairs of data (number of features and bits set by them) for each value on the range of 20 to 159 (number of features normally found on a bloom filter). For bloom filters with exactly 160 features in the last filter we had 509 files. Given the unbalanced distribution, we randomly selected 28 pairs from these 509 files to create our statistics and balance the number of pairs for each value.

To apply the least square method, we broke our data set into three parts: set 1 varying the number of features on the last bloom filter from 20 to 50, set 2 varying from 51 to 100, and set 3 from 101 to 160. The reason for the division is due to the better results obtained when compared to a single set with all data together. The results of our prediction using the least squares method are shown in Figs. 36 - 38. We can see that the estimated continuous line fits better into the range of the real values in the three scenarios (dashed line) than when we considered all data set to create our estimator (see Fig. 39).

To estimate the number of bits set in a bloom filter given any number of features from 20 to 160, we can use Eq. 6.3 with the values provided in Table 33. We recommend to use such values for computing the theoretical value of e_i given bloom filter i.

$$e_i = a + b \cdot g_i \tag{6.3}$$

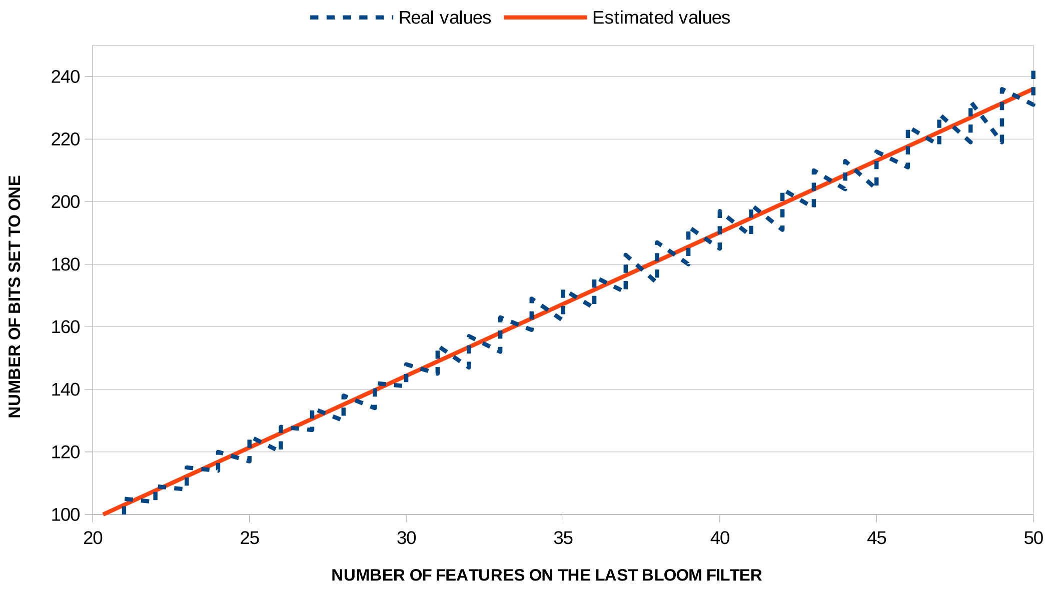


Figure 36 – Linear Regression - Least Squares technique: Using the number of features in the last bloom filter of set 1 (t5-corpus) to estimate the number of bits set.

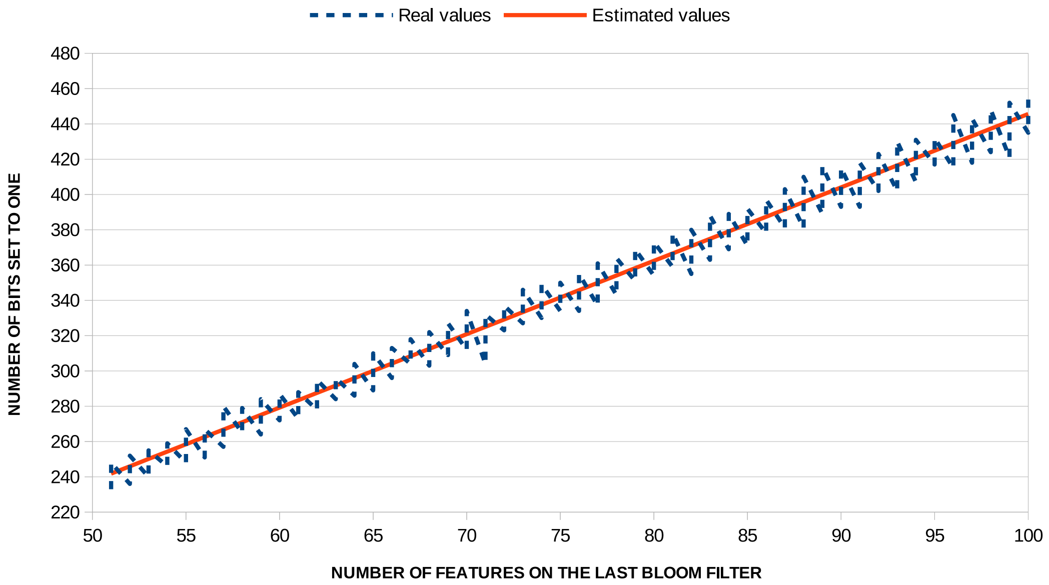


Figure 37 – Linear Regression - Least Squares technique: Using the number of features in the last bloom filter of set 2 (t5-corpus) to estimate the number of bits set.

6.3.3.3 Estimating the number of features in common

After knowing the number of bits set in any bloom filter with g_i features ($20 \leq g_i \leq 160$), we can estimate the minimum amount of data shared between bloom filters to sdhash produces $score \geq 1$ or $score \geq 21$. By estimating the number of shared features, we can correlate this number with the number of bytes in common between the objects the filters represent, using the relation introduced in Sec. 6.3.1. To summarize: For each selected feature inserted into the bloom filter, there is about β bytes (feature size) of similar data. For this reason, to find out the

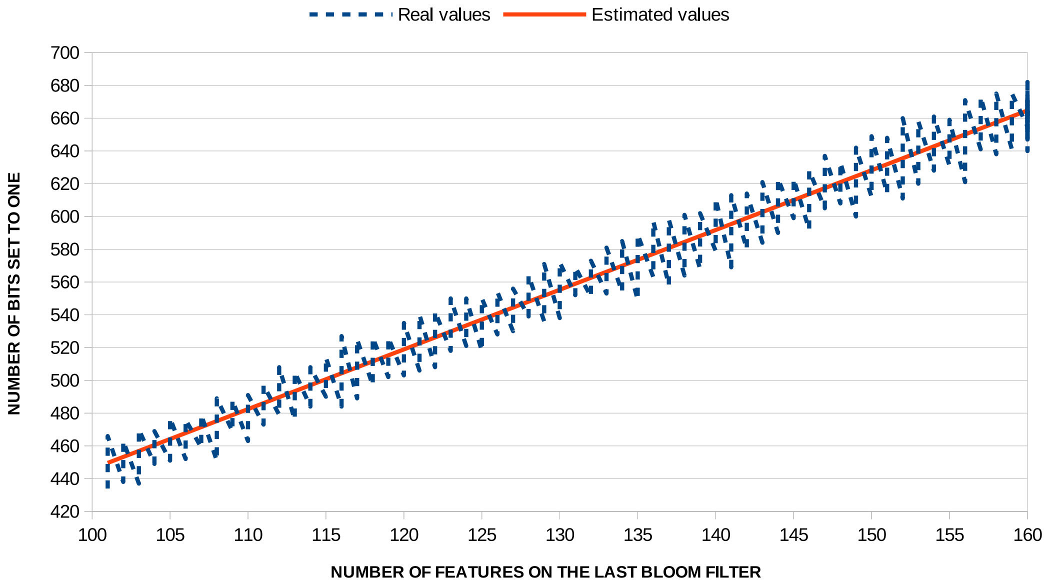


Figure 38 – Linear Regression - Least Squares technique: Using the number of features in the last bloom filter of set 3 (t5-corpus) to estimate the number of bits set.

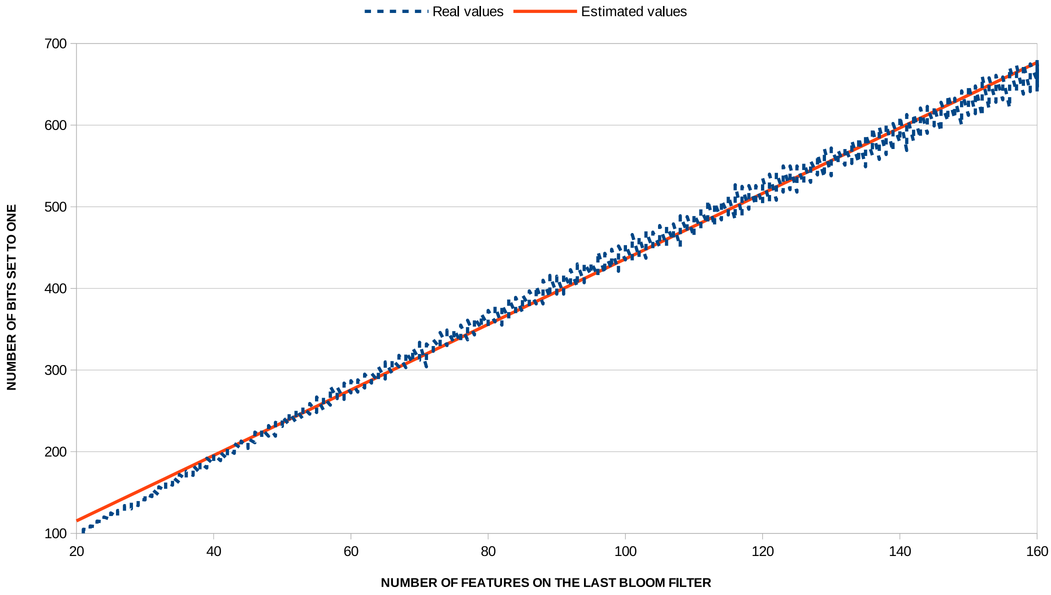


Figure 39 – Linear Regression - Least Squares technique: Using the number of features in the last bloom filter of all t5-corpus to estimate the number of bits set.

number of similar bytes between two objects F_x and F_y, we need to know the number of features they have in common. We use the information provided in the previous section to estimate the number of bits set in bloom filters. Note that in practice, this number may vary even for two filters with the same number of features. Although this behavior is not observed in our estimates, we highlight that we took into consideration real-world data for our analysis and our results are close to what we have in practice.

To estimate the number of features in common ($G(bf_i, bf_j)$) in bloom filters i and

Table 33 – Least Square technique - Parameters to estimate the number of bits set on a bloom filter for a given number of features.

Condition	a	b
$g_i < 20$	0	0
$20 \leq g_i \leq 50$	6.76	4.59
$51 \leq g_i \leq 100$	29.70	4.16
$101 \leq g_i \leq 160$	81.47	3.65
$g_i > 160$	0	0

j, we can use classical bloom filter analysis, and by manipulating Eq. 2.3, we get the following expression.

$$G(bf_i, bf_j) = \left\lceil g_i + g_j - \frac{ln(-1 + p^{k.g_i} + p^{k.g_j} + \frac{e_{i,j}}{m})}{k.ln(p)} \right\rceil \quad (features) \tag{6.4}$$

Here, we need to provide $e_{i,j}$ to get the number of features in common; in case we do not have this number, we can estimate it to get a desired score using Eq. 6.5.

$$e_{i,j} = \left\lceil (\frac{score}{100} \times (e_{max} - C)) + C \right\rceil \quad (bits) \tag{6.5}$$

The parameter e_{max} is given by Eq. 2.5 and C by Eq. 2.7. Here, we need to provide the number of bits set by each filter (e_i) using Eq. 6.3 (parameters are provided in Table 33).

When sdhash compares two objects, it provides a score related to the similarity shared by them. Having $score \geq 1$ means that some similar data was detected by the tool, but it was not enough for a significant similarity. Roussev, V. (ROUSSEV, 2010) recommends considering as similar all matches with $score \geq 21$ due to the high number of false positives got from smaller scores. Here, we will analyze the minimum amount of similar content (features) shared by two bloom filters to get the two minimum score values: $score \geq 1$ ($score_{min} = 1$) and $score \geq 21$ ($score_{min} = 21$).

6.3.3.3.1 Comparing two full bloom filters

We start our analysis with the following case: Given two full bloom filters (with f_{max} = 160 features), what is the minimum number of bytes (features) they must have in common to sdhash produce $score \geq 1$ and $score \geq 21$? Note that each filter maps 10 KiB of data. Using the equations presented before and data got from our linear regression approach to estimate the number of bits set in each filter, we got the following results (for sdhash default parameters):

$score_{min} = 1$: Minimum of 56 features, corresponding to $\approx$ 3584 bytes.

$score_{min} = 21$: Minimum of 80 features, corresponding to $\approx$ 5120 bytes.

As we can see from our results, 10 KiB objects require about 35% of similar content to sdhash identify some similarity and 50% to produce a significant score.

6.3.3.3.2 Comparing full vs. incomplete bloom filters

Here, we are interested in performing a single comparison between a full bloom filter with an incomplete one (with less than f_{max} features). Figs. 40 and 41 show some examples of comparisons where one of the filters was set by a number of features varying from 20 to 160. By analyzing the minimum number of features to get the minimum scores, we needed, in all cases, about 35% of features in common for having $score_{min} = 1$ and about 50% to have $score_{min} = 21$.

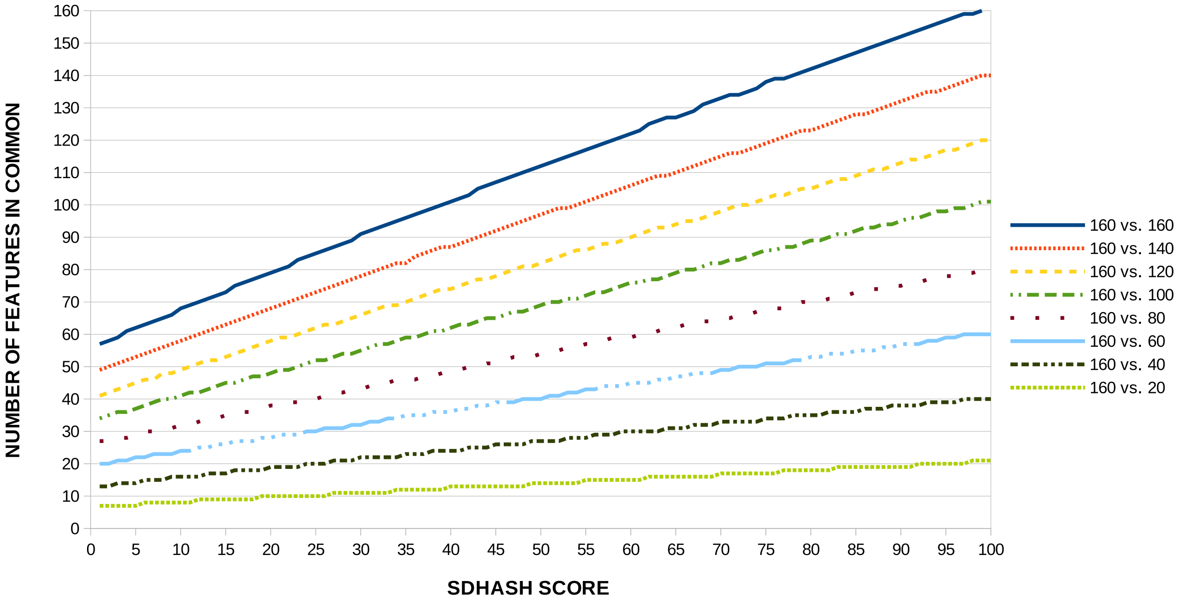


Figure 40 – Minimum number of features in common per similarity score of some bloom filter comparisons (full vs. incomplete filters).

6.3.3.3.3 Comparing two incomplete bloom filters

A third case involves comparing incomplete filters. We estimate the minimum number of features in common between all possible bloom filter comparison combinations, varying the number of features from 20 to 160. On average, we found that 33.18% (std. 1.13) of features in common is necessary to produce $score \geq 1$ (with values varying from 30.40% to 37.50%) and 47.61% (std. 1.30) to get $score \geq 21$ (with values varying from 44.44% to 53.13%).

6.3.3.3.4 Comparing one bloom filter to many

When `sdhash` is confronted with a case where object F_x has one bloom filter and object F_y has many, a comparison of the single filter of F_x is performed with all filters from F_y; the maximum similarity between all comparisons is chosen and returned as the final score. The constraints of this scenario are the same ones discussed in the three previous ones; about 33.18% of similarity is required for getting $score_{min} = 1$ and 44.44% for getting $score_{min} = 21$.

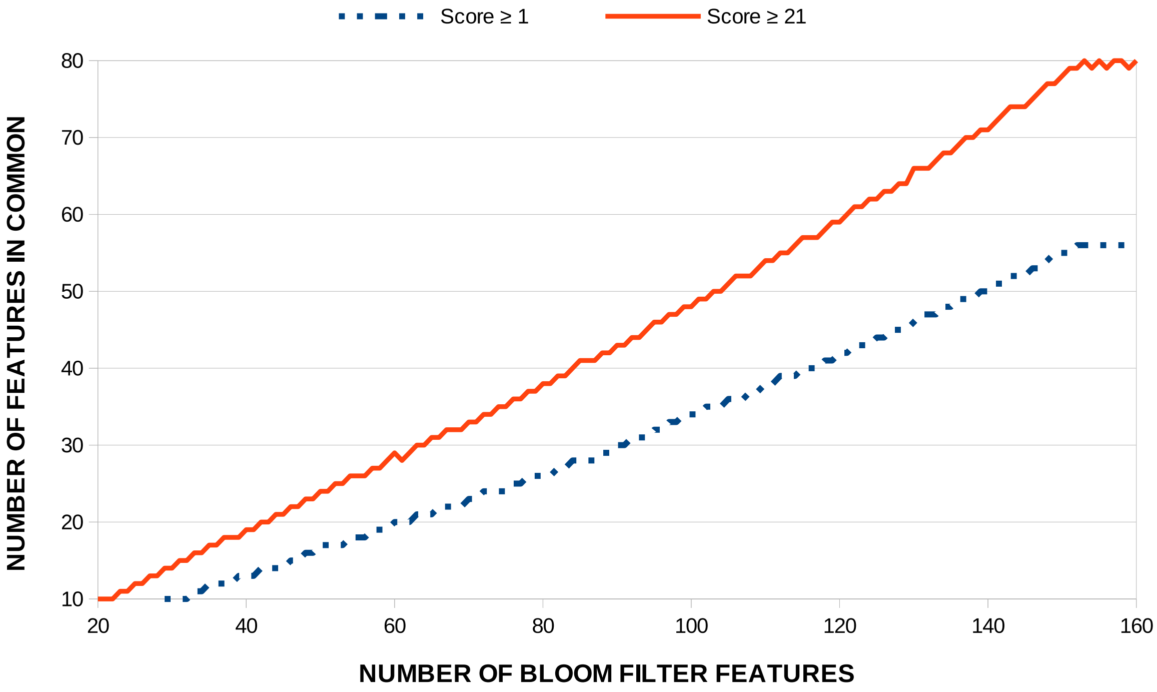


Figure 41 – Minimum number of features in common concerning the total number of features of a bloom filter to get the minimum similarity required.

The difference here is that the similar content must be located in a single block, mapped in only one of F_y filters.

6.3.3.3.5 Comparing many to many bloom filters

The last case discussed here is what we usually find in practice: Large files having many bloom filters each. By assessing the similarity of two objects F_x and F_y having $|bf_x|$ and $|bf_y|$ bloom filters, sdhash compares each filter of F_x to each filter of F_y and average results using the maximum values obtained in each step. In this case, we mostly have comparisons of full vs. full bloom filters, except for those related to the last filters of each object.

For simplicity, we consider only full bloom filters to perform our analysis, arguing that: (1) As the file size increases, the impact of the last filter becomes more irrelevant; (2) the impact of the last filter will interfere only when the similar part is located at the end of one of the objects; if it is in any other part, that will eliminate the influence of incomplete filters as long as they are not involved in the similarity.

Two main factors impact the score of sdhash: the objects' size and the similarity shared between them. The size plays an important role because of the average performed by the comparison function. The sum of all maximum scores (from the bloom filter comparisons) is divided by the number of filters of the smallest object. For this reason, as the object size increases, the similar content should also increase so the similarity can be detected.

To estimate the minimum amount of similarity that can be detected by `sdhash` for two objects with many bloom filters, we need to know how the similarity is spread over the files. First, we consider a single block of common content starting at the beginning of the objects; then, the next sections analyze the remaining cases.

It is important to highlight that, after selecting the maximum values of the bloom filter set comparison, the average is calculated and, in current `sdhash` implementation, the `round` math function is applied to the result; after average, values > 0.5 will be rounded to 1. In our analysis, we adopted the same criteria. Besides, we highlight that the average takes into consideration the smaller object size, which gives `sdhash` the ability to compare extremely large objects with small ones and be limited to the detection capabilities of the smallest file. Our goal is to figure out how many bytes (or features) are required to get $score_{min} = 1$ and $score_{min} = 21$ for two objects having many bloom filters. To this end, we create a script to simulate the minimum amount of similarity between objects as their sizes increase using `sdhash` equations presented in Sec. 2.4.5.4.2, along with Eqs. 6.4 and 6.5. Here, we increase their sizes to the point of having $score_{min} = 1$ and $score_{min} = 21$ and measure the percentage of similarity in comparison to the smallest object size. We summarize our results in Fig. 42.

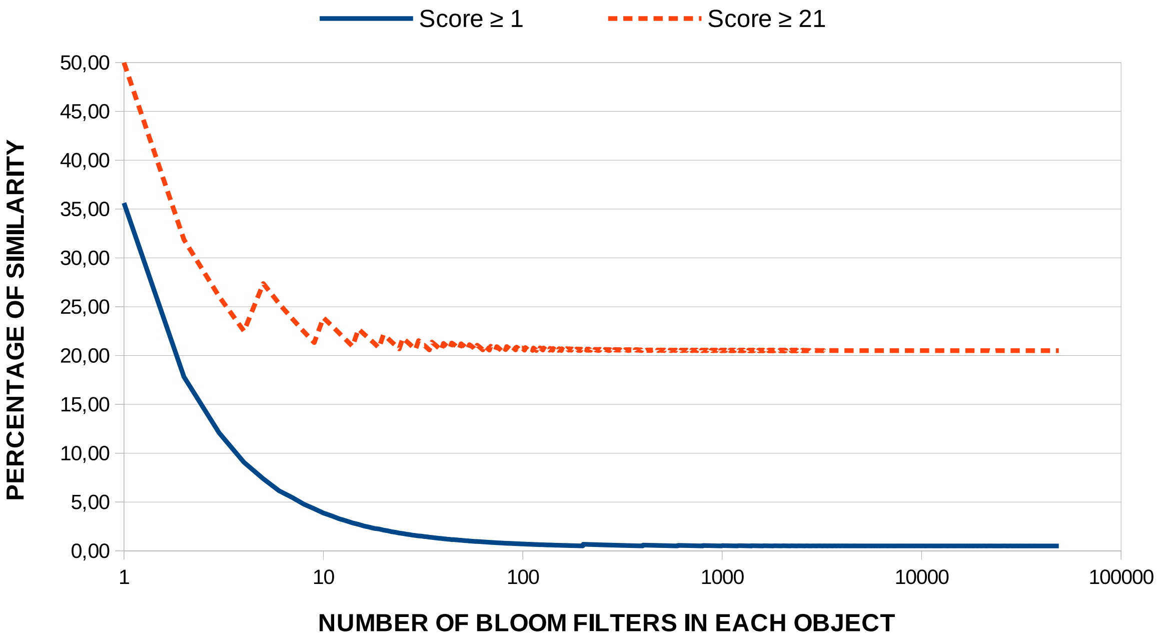


Figure 42 – Percentage of similar content between objects with many bloom filters to produce the minimum similarity scores.

Our results show that, for small objects, a significant amount of similarity is required. For instance, comparing two objects having one bloom filter each (≈10 KiB of data) requires at least 35% of similar content to get $score_{min} = 1$ and 50% to get $score_{min} = 21$. Increasing the objects' size to two bloom filters (≈20 KiB of data) caused the percentages to drop to 18% and 32%. With sizes of around 200 bloom filters (≈1.95 MiB), we only need 0.50%

and 20.50% to get $score_{min} = 1$ and $score_{min} = 21$, respectively, which we found out to be the minimum percentage values `sdhash` can get. Although we might expect to have a value of 1% for $score_{min} = 1$ and 21% for $score_{min} = 21$, we highlight that the percentages got in our tests were different due to `sdhash` being able to detect similar with less content, but being limited to its comparison function. The 0.5% value obtained for $score_{min} = 1$ is due to the *round* operation applied in the comparison function result, which makes the score goes to 1. The same behavior occurred for $score_{min} = 21$, got from a similarity of 20.5% detected.

The curve for $score_{min} = 21$ has a particular behavior in the graphic with some irregularities at the beginning, and, as the object size increases, it tends to stabilize at the minimum values (20.50%). The ups and downs behavior is due to the similar part shared between the objects being detected in a new bloom filter, which demands a large piece of similar content to produce a score. Remember that at least 56 features in common are required for having $score \geq 1$. One of the reasons for this initial effort for detecting similarity is due to the cutoff point (C) established by `sdhash` (see section 2.4.5.4.2); once we achieve the minimum amount of similar content, the increase in score becomes proportional to the number of overlapping bits.

6.3.3.4 The influence of α on sdhash score

To compute the cutoff point (C) value necessary in the score calculation of two bloom filters (Eq. 2.8), `sdhash` uses a α variable to decrease the chances for false positive matches. One element that is part of the equation is e_{min}, responsible for estimating the minimum number of overlapping bits due to chance of two filters; however, the estimate does not seen to be enough for decreasing the false positive rate since `sdhash` increases e_{min} by a factor of α. According to Roussev, some comparisons between random data (expected to be different from each other) produced $score \geq 1$, indicating similarity where there is not, leading to the adoption of this extra variable. Roussev argues that the α value was *"calibrated experimentally so that the comparison of the digests of random data consistently yields a score of zero"*. The default value is set to $\alpha = 0.3$. In this section, we show the impact of α on similarity and demonstrate through two general cases of bloom filter comparisons that this variable makes `sdhash` sub-optimal when comparing small objects with respect to its detection capabilities.

6.3.3.4.1 Comparing two full bloom filters

We showed in the last section that `sdhash` requires a minimum number of common bytes (features) between full bloom filters of 3854 (35%) and 5120 (50%) bytes (corresponding to 56 and 80 features) to produce $score \geq 1$ and $score \geq 21$, respectively. When we remove the interference of the α variable in the equation, making $C = e_{min}$, i.e., $\alpha = 0$, the new values for the same settings become:

$score_{min} = 1$: Minimum of 3 features, corresponding to $\approx$ 192 bytes (1.88% of the file size).

$score_{min} = 21$: Minimum of 40 features, corresponding to ≈ 2560 bytes (25% of the file size).

A significant reduction in the minimum number of similar bytes is observed when removing the interference of α in the calculus. This way, `sdhash` becomes more sensitive for detecting similarity, specially when dealing with small objects.

6.3.3.4.2 Comparing many to many bloom filters

When comparing objects with many bloom filters (large ones), we saw that as the object size increases, the minimum amount of similar data required by `sdhash` decreases, until the point of requiring 0.50% and 20.50% to get $score_{min} = 1$ and $score_{min} = 21$, respectively. Besides, we also showed that every once in a while, when the similar part is mapped into a new bloom filter, `sdhash` fails to detect this new part until we have a large portion of similar content (about 56 features) mapped into this new filter.

By removing the influence of α from the similarity assessment, we could observe two benefits: (1) A reduction in the minimum amount of similar content required by `sdhash` to detect similarity; (2) a small gap between the minimum amount of similarity detected by large and small objects. Our results are summarized in Fig. 43, where we show the influence of α when comparing the minimum percentage of similarity detected by the object size variation. As demonstrated by our results, we achieve the minimum similarity required faster without the influence of this variable, i.e., when $\alpha = 0$. We also can see that α impacts most small objects, which makes `sdhash` sub-optimal for this class of objects. However, the downside of removing the influence of this variable in the score is an increase in false positive matches. As a solution, we present in Sec. 6.6 a new way to deal with false positives and still have better detection capabilities than when using α.

6.3.3.5 Border effect on similarity

The similarity of two objects can be found at the beginning, ending or middle of objects; it can also be distributed in several pieces all over it, in a way that we have blocks of similar data but also blocks of different data. The *border features* are those features shared between objects in which some of their bytes are part of a similar block and the rest belong to the different part. We highlight that these features are considered as differences when assessing similarity due to their composition having some different bytes, resulting in different hashes. The question we seek to answer here is: How much does the border features influence similarity?

6.3.3.5.1 The presence of border features

The number of border features can vary between 0 and $\beta - 1$, where β is the feature size. Fig. 44 shows two objects having some bytes in common (at the beginning) and the influ-

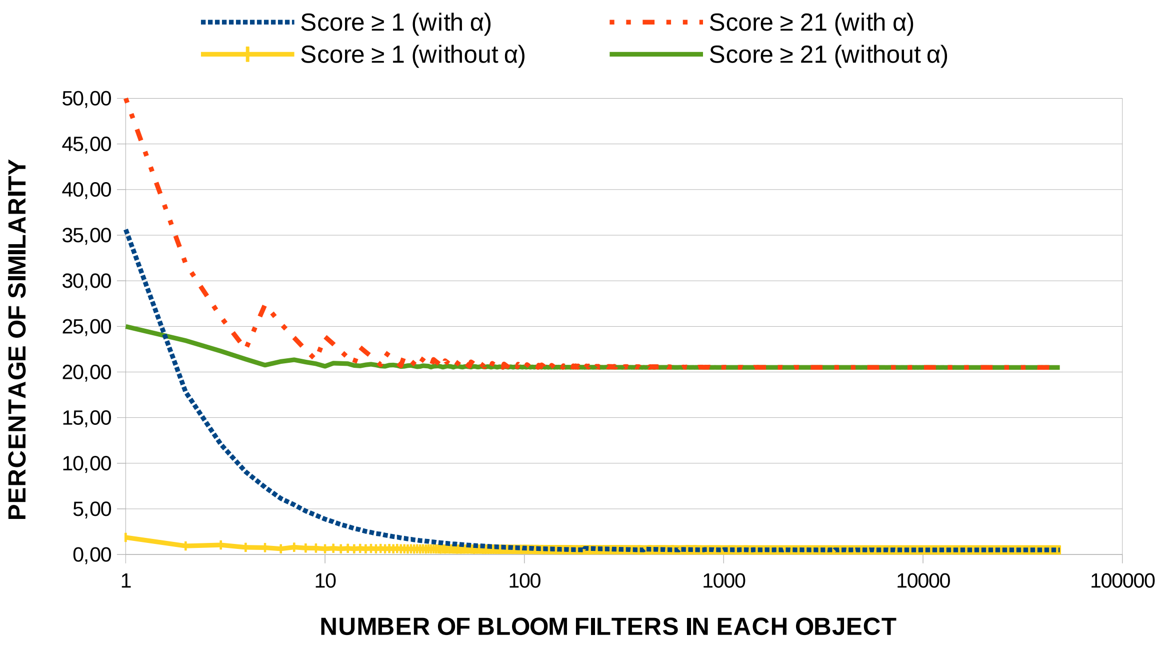


Figure 43 – Impact of the α variable on the similarity assessment when comparing objects having many bloom filters.

ence of the different parts in the feature extraction process. We emphasize that not all extracted features are selected to be part of the object digest, as discussed in Sec. 2.4.5.4.1. A feature is selected only if it has the lowest entropy value in a window of W features at least t times; this way, at most $\lfloor W-1/t \rfloor$ border features can be selected. In sdhash current implementation, the number of border features varies from 0 to $\lfloor 64-1/16 \rfloor = 3$. The value of 0 is got when features from similar and different parts of the object have lower entropy than the border features. Note that when the block with similar content is in any position of the object except for its beginning, we have two borders, and the number of border features can be doubled.

The number of features shared between objects can be computed as follows:

$$V = E + (2 \cdot B) + D \tag{6.6}$$

where V is the total number of features of the two objects, E the number of equal features, D the number of different features, and B is the number of border features (counted only once).

Let's consider two objects of the same size sharing $P\%$ of bytes in common broken into n pieces of dissimilar size (but all with significant content to have a few features extracted). By adapting Eq. 6.6 and providing the number of total features of the two objects and the value of P, we can estimate the number of equal features using eq. 6.7.

$$E = \left\lfloor \frac{(V \cdot P)}{100} \right\rfloor - (2 \cdot B \cdot n) \tag{6.7}$$

Given sdhash default parameters, we can have between $0 \leq B \leq 3$ for each border. The impact of border features for the particular case where two objects share a single block of common

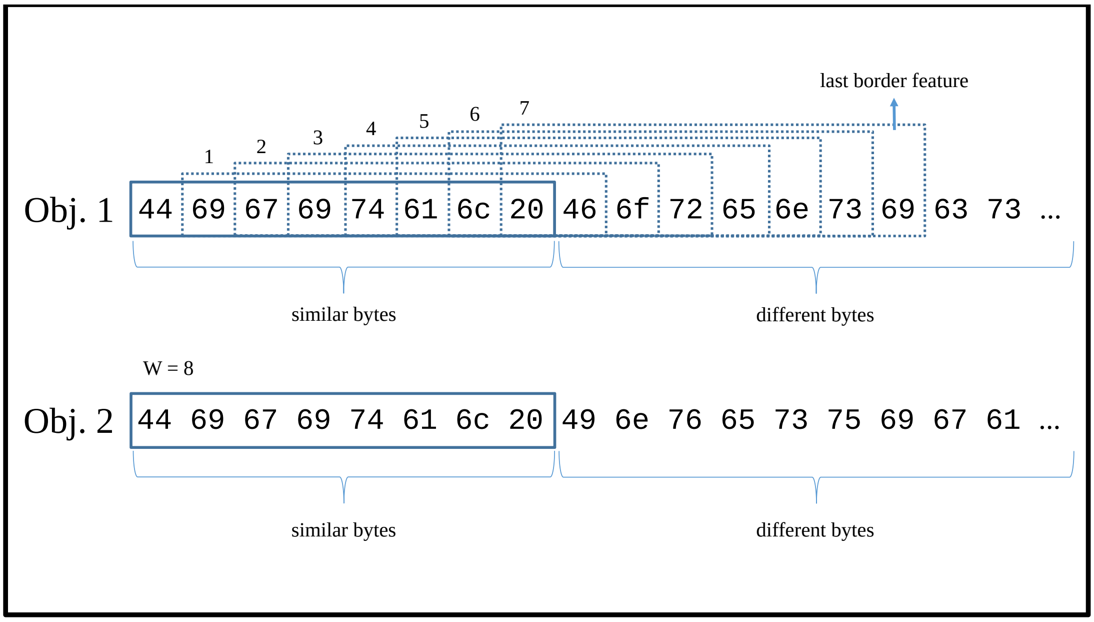

last border feature
1
2
3
4
5
6
7
Obj. 1
44 69 67 69 74 61 6c 20 46 6f 72 65 6e 73 69 63 73 …
similar bytes
different bytes
W = 8
Obj. 2
44 69 67 69 74 61 6c 20 49 6e 76 65 73 75 69 67 61 …
similar bytes
different bytes

Figure 44 – Border features definition

data ($n = 1$) can be seen in Fig. 45. In this experiment, we considered that `sdhash` selects a feature for every $W = 64$ (corresponding to the values found in Sec. 6.3.1); therefore, only one border feature is selected by each border. By our results, as the number of the total features in common (V) increases, the impact of the border features becomes less relevant in the similarity assessment compared to the set without the border influence. Since the number of features in an average object is high (e.g., an 1 MiB object has around 16384 features), we expect that the border features do not impact most scenarios related to a single block of equal content.

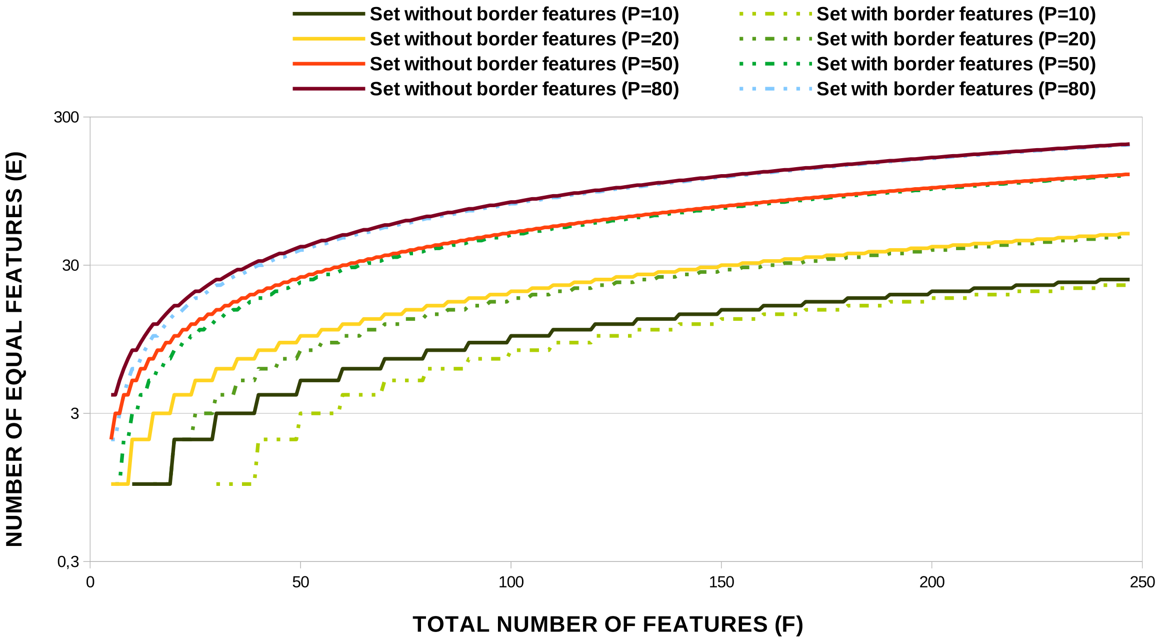


Figure 45 – Influence of border features on similarity for a single equal block case and the relation with objects of different sizes.

When similarity is fragmented all over the object, more influence of border features on similarity is expected. Fig. 46 presents a relation between the number of pieces (n) in which the similar content is broken into and object size (represented by the number of features). The optimal scenario is not having border features ($B = 0$) and similarity not fragmented ($n = 1$). For small objects, the similarity is barely detected as we increase n; only when the object size increases (as the similar content), the detection tends to the optimal case. To conclude: Small objects are sensitive to fragmented similarity, but as the object and similar content sizes increase, the impact becomes less relevant.

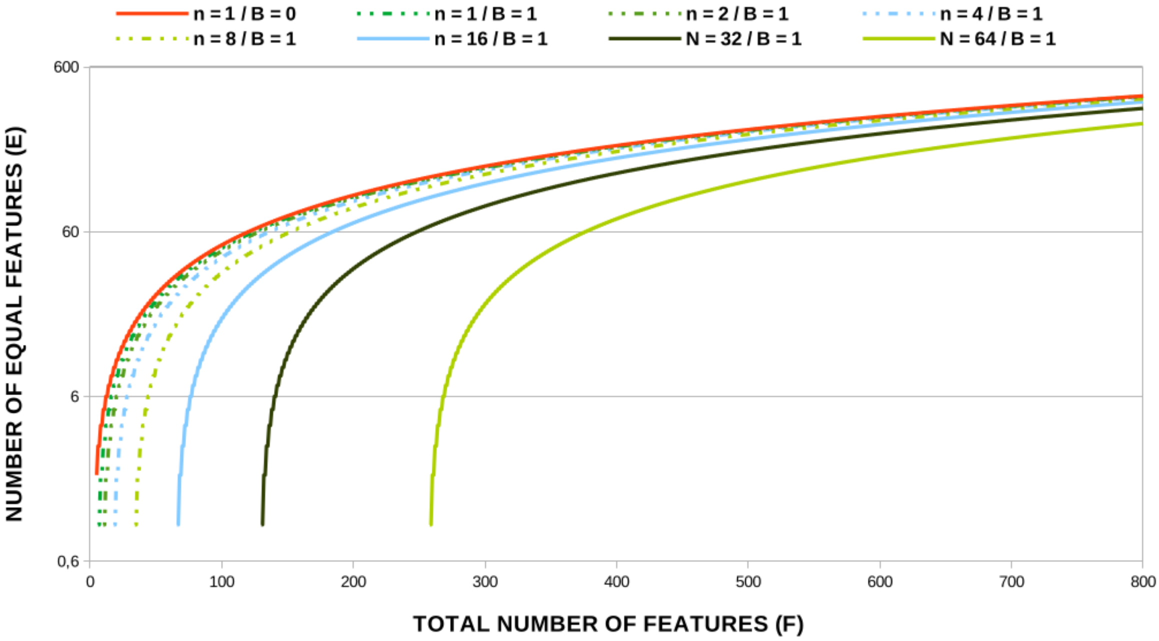


Figure 46 – Influence of border features on similarity by breaking equal data into n pieces for 50% of similarity.

6.3.3.5.2 Impact of border features on bloom filters

When inserting features into bloom filters, we may have cases where the border features become part of the filters with similar data, influencing (negatively) the similarity score. Next, we discuss some scenarios involving the insertion of border features into bloom filters. In each scenario, we consider full bloom filters and the maximum number of border features ($B = 3$). Here, we consider only the effect of one border, which means that either the similar content is placed at the beginning of objects or the similar content is too large that more than one bloom filter is necessary to store it.

Case 1: The border features impact only one bloom filter. Here, we have up to 157-159 equal features between two objects and 1-3 border features inserted in the same filter. We expect that the only impact will be the decrease of similarity score by a few points due to the border features. For instance, comparing two bloom filters having $E = 157$ and $B = 3$ will produce $score = 97$.

Case 2: The border features have no impact on similarity. In this case, two objects under comparison have more than one bloom filter. The first filter has 160 equal features, and the second one has the three border features along with the rest of the features (different ones). In this case, the first filter will have a maximum score ($score = 100$), and the border features will have no impact over similarity.

Case 3: The last case considers a scenario where the border features are split into different bloom filters. The first filter will have 158-159 equal features plus 1-2 border features. The rest of the border features will be placed on the second filter. In this case, the first filter's border features will decrease the similarity score produced to $score = 98$ or $score = 99$.

The impact of the border features depends on the object size, amount of similar content, and the number of pieces the similar part is broken into. As the similar content size decreases, and the number of pieces increases, we have more influence of border features, making `sdhash` struggles to detect similarity. However, some factors may attenuate the effects of border features or cancel it: (1) The feature selection process, for not picking up these features; (2) the border features position, as they are stored in a different bloom filter from the one having similar content-related features; and (3) the object and similar content sizes: The larger the object, the better `sdhash` performs. Besides, the more similar bytes we have, the less the effect of border features.

6.3.4 Understanding sdhash score and its limitations

Roussev, V. and Candice, Q. (ROUSSEV; QUATES, 2013) mention that *"sdhash comparison result should not be interpreted as a percentage of common content. Rather, it should be viewed as a confidence value that indicates how certain the tool is that the two data objects have non-trivial amounts of commonality."*. Besides, they propose a guide to interpret `sdhash` results, summarized below:

Strong (21-100) Strong indication of similarity with very few false positive matches.

Marginal (11-20) Similarity depends on the underlying data. For composite file types (`pdf`, compound files, etc.) similarity is mostly related to common blocks (see Chapter 5), while simpler file types (`txt` and `html`), may present significant results.

Weak (1-10) Most matches are false positives, except for simpler file types, where $score \geq 5$ may present matches with similarity.

Negative (0) No similarity found.

Looking at the interpretation of the `sdhash` score, although it seems to be simple (it considers two objects similar or not), in practice, it is hard to separate and prioritize results; also, some experts on the field argue about problems in the way it works (BREITINGER; BAIER, 2012c; BREITINGER; BAIER, 2013; RAFF; NICHOLAS, 2018). First, it is common sense to look at the score and interpret it as a percentage value (since it varies from 0 to 100); second, the labels are not equally distributed: the first range cover most of the possible values (21-100) (RAFF; NICHOLAS, 2018); third, `sdhash` does not distinguish between resemblance and containment and returns a single value representing both metrics. For the last case, consider the following scenarios: (1) When comparing two objects of the same size having only a couple of different bytes, `sdhash` may produce $score = 100$ (maximum similarity for resemblance); (2) when comparing a particular object with a small fragment of it, `sdhash` also may produce $score = 100$ (maximum similarity for containment). Here, `sdhash` produced the same results for two different cases. In practice, practitioners have to look for further information (e.g., objects size) to have a clue about the type of similarity related to the match.

6.3.4.1 sdhash score for single common block scenario

In this section, we seek to understand how `sdhash` score is influenced by the amount of similarity of two objects to better explain it and to find a relation (if it exists) between them. To this end, we performed experiments varying the object size (in number of bloom filters, from 1 to 100 - corresponding to about 10 to 1000 KiB) and amount of similarity (percentage value, from 5 to 95%) of two objects, and observed how the score behaves. We used `sdhash` comparison formulas (Sec. 2.4.5.4.2) to perform our analysis. Results are summarized in Fig. 47, where we simulate a scenario of a single block in common inserted at the beginning of objects.

We can see in the results that `sdhash` score approximates the percentage of content similarity as the object size increases for this particular scenario. Here, the score, which is very unstable and constantly changes for small object sizes, becomes closer to the real similarity value as the object size increases; at about 50 bloom filters (500 KiB of data), the score becomes more consistent and closer to the real similarity value. Besides, as the percentage of common data also increases, the score oscillation gets weaker, indicating that the more the similarity content, the more accurate the `sdhash` score becomes.

By removing the influence of α variable from the comparison calculus (Sec. 6.3.3.4), we see in Fig. 48 that `sdhash` score is very close to the real similarity even for small objects; the oscillations are less frequent, and for sizes as small as ten bloom filters (about 100 KiB of data), the scores are almost equal to the real similarity percentage (single common block scenario).

6.3.4.2 sdhash score and multiple common blocks scenario

We saw that for the simplest scenario, where we have a single block in common between objects, the `sdhash` score approximates the real similarity value as the object size

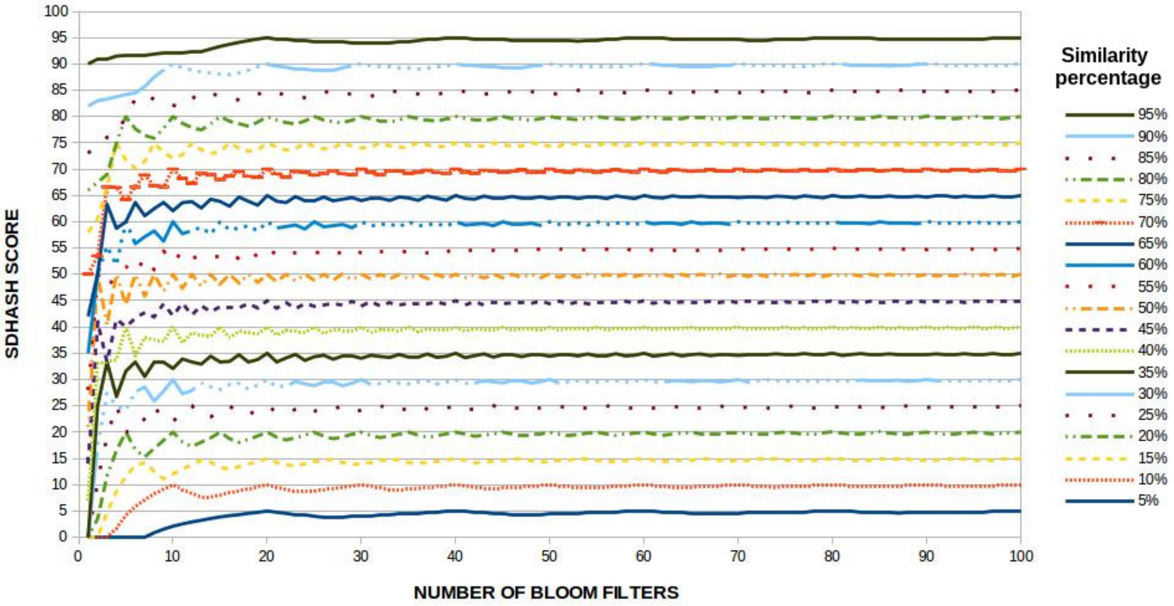


Figure 47 – Impact on the `sdhash` score for the object size and similarity percentage variation for a single common block scenario.

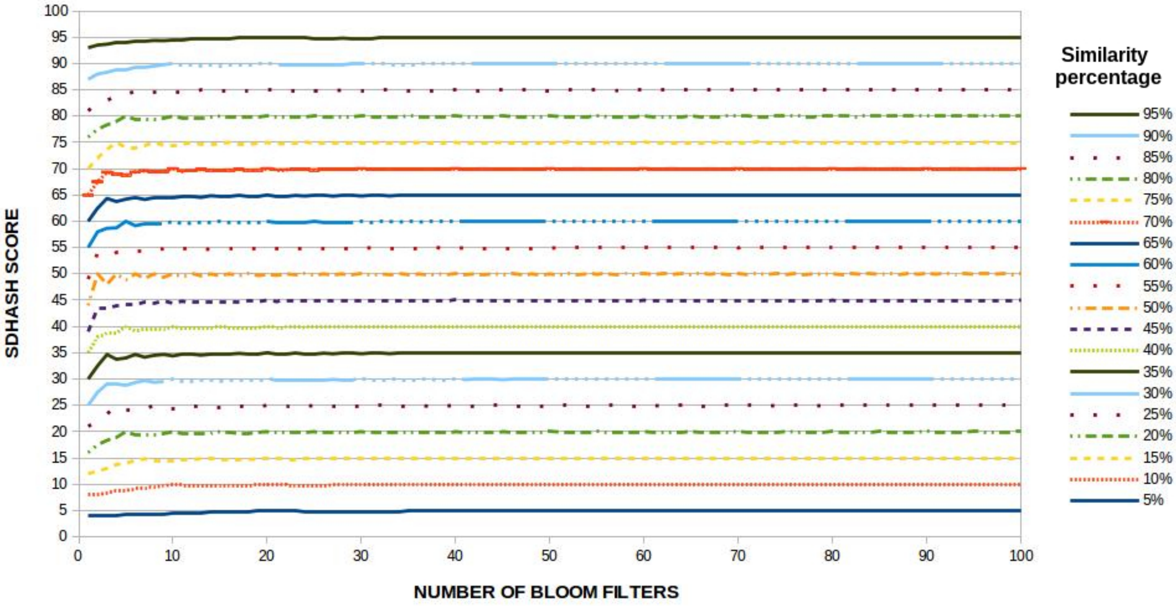


Figure 48 – Impact on the `sdhash` score (without α influence) for object size and similarity percentage variation for a single common block scenario.

increases (especially when removing α influence); however, how does the score behaves when similar content is spread all over the object?

Here, we break the similar content in several pieces (n parts) and see the impact on the similarity score. The break occurs with the following restriction: $n \leq |bf_x|$, which means that the number of breaks (of the similar part) is equal or smaller than the number of bloom filters of an object x. We used this restriction so that similar pieces were stored into different filters, since storing two pieces of similar data in the same filter may increase similarity. Given the high number of possibilities for parameters variation, we restricted our experiments to object sizes of 5, 10, 20, 25, 50, 100, and 500 bloom filters; to similarity percentage of 5, 10, 20, 25, and 50%; and to n having at most the number of filters of the smallest object.

Figs. 49 and 50 show the results for the particular cases where we have 50% and 20% of similar content for the object sizes under consideration. In both scenarios, we can see that, as n increases, the score tends to drop to its minimum value. For the first case, we know from Sec 6.3.3.3 that for 50% of similar data, `sdhash` produces $score = 21$ for small objects and, as the object size increases, the score produced becomes closer to the percentage of similarity, to the point of producing $score = 50$ for 50% of similar content. In Fig. 50, for 20% of similar content, the score drops to 0 as n increases since the amount of similar content spread over the object bloom filters is not enough to produce a significant score (the number of features in each filter will be less than the minimum amount for $score \geq 1$). Note that: (1) The smaller the size of the objects and the similar content, the faster the curve falls, and (2) in all cases, the maximum score is got for low n, and its value is limited to the similarity percentage.

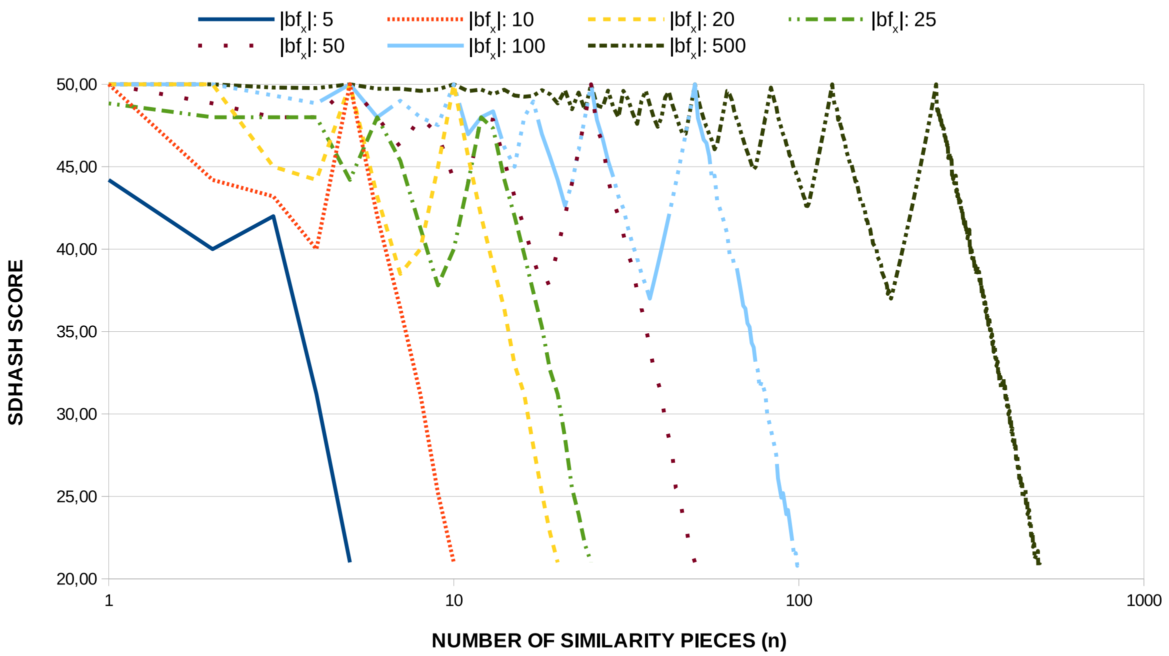


Figure 49 – Similarity fragmentation and its impact on `sdhash` score for different object sizes (50% of similarity content).

We also varied the similarity percentage value for fixed object sizes to observe how the similar content size is impacted when increasing n. Figs. 51 and 52 show some results. We can observe that, as the similar content increases, the influence of n decreases; for n limited to the number of bloom filters in the objects, we have a slower reduction in the score than expected. For 50% of similarity or more, all n parts (limited to the number of bloom filters) will have at least the minimum score of strong similarity (≥ 21); lower similarity percentages will work for

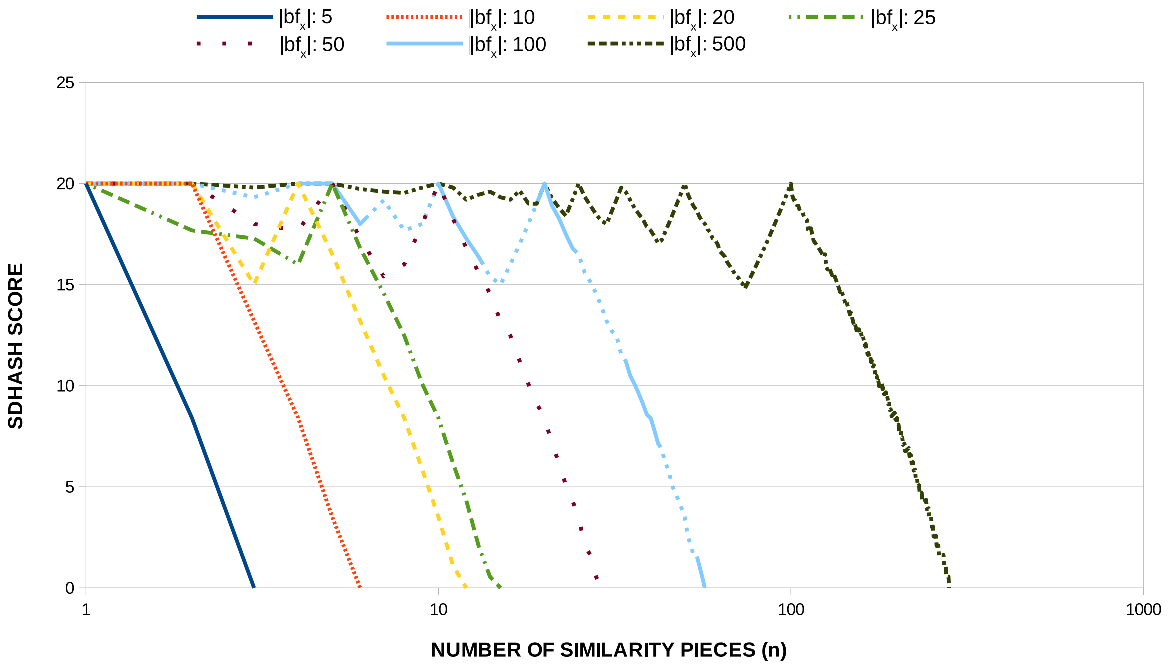


Figure 50 – Similarity fragmentation and its impact on `sdhash` score for different object sizes (20% of similarity content).

low n values, and as this value increases, the score tends to zero as there are not enough similar features.

We can estimate the maximum n to the point of having the maximum similarity score, using Eq. 6.8.

$$n = \frac{P \cdot |bf_x|}{100} \tag{6.8}$$

Here, P is the percentage of similar content. For instance, given P =10% and $|bf_x|$ =20, we see from figure 52 that n =2; from this point on, the score will only drop.

Comparing figures 51 and 52, we can see a relation between the object size and n. For instance, the object size considered in Fig. 51 is five times larger than the one in figure 52. Considering the 5% of similar content case, $score = 0$ is reached for the object of figure 51 with n =15, while the object of figure 52 reaches $score = 0$ with n =3 (about five times of difference). The same relation was observed for other percentages of similarity as well as for different object sizes.

Given the relation discussed above, we noticed that using similarity content lower than 35% (minimum value for `sdhash` detects similarity of two bloom filters, see Sec. 6.3.3.3), we can estimate the maximum n (n_{max}) in which we start having $score < 1$; higher percentage values for similarity will produce $score \geq 1$ anyway for every $n \leq |bf_x|$.

Table 34 shows some statistics obtained from our experiments in which we found a relation to get n_{max} before score drops to zero. Here, we present the following terms:

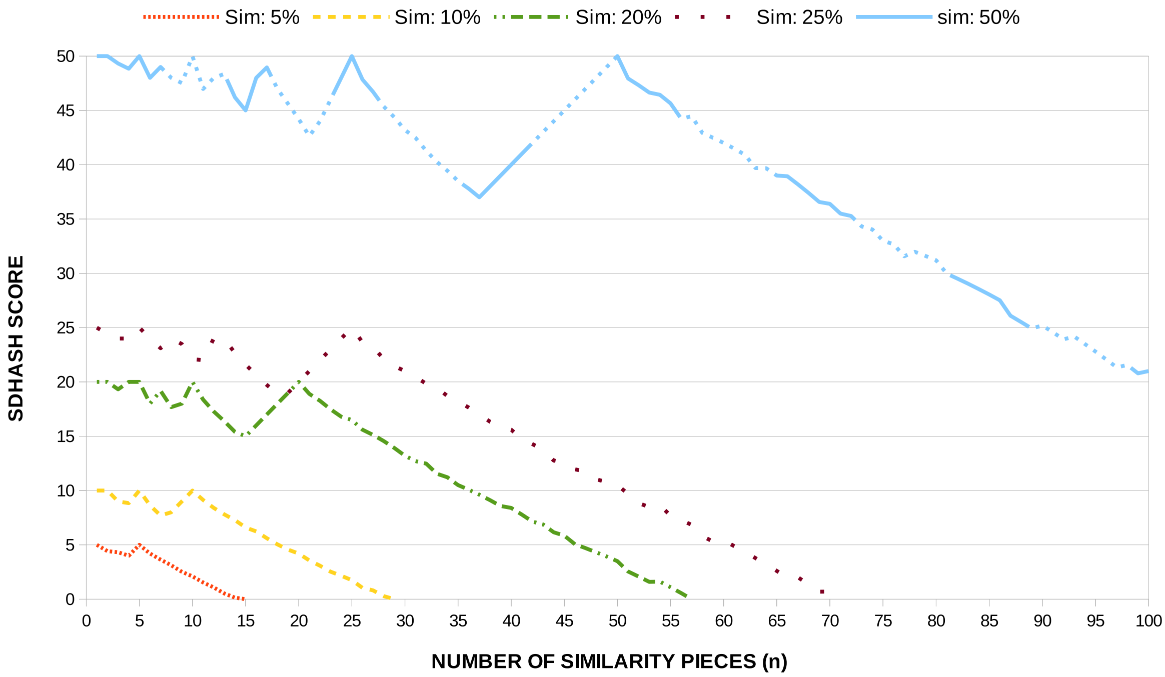


Figure 51 – Similarity fragmentation and its impact on `sdhash` score for different amounts of similar content (object size: 100 bloom filters).

R**:** Division of $|bf_x|$ of a given object x by $n_{max}+1$ by which we have $score<1$;

Q**:** Division of P by R and used universally to estimate n_{max}.

P	R	Q
1	0.03	33.33
5	0.15	33.33
10	0.15	34.48
15	0.29	34.88
20	0.43	35.09
25	0.71	35.21
30	0.85	35.29
35	0.99	35.35
Average:		34.62

Table 34 – Statistics for estimating n_{max}

By using Q_{avg} (average of all Q) from our experiments and having P and $|bf_x|$, we can estimate n_{max} to have an idea about the maximum value from which we still have $score \geq 1$ using Eq. 6.9.

$$n_{max} = \frac{P \cdot |bf_x|}{Q_{avg}} - 1 \tag{6.9}$$

Our results show that, although the `sdhash` score does not perform well for small objects, it tends to the real percentage values of similarity as the object size increases. Besides,

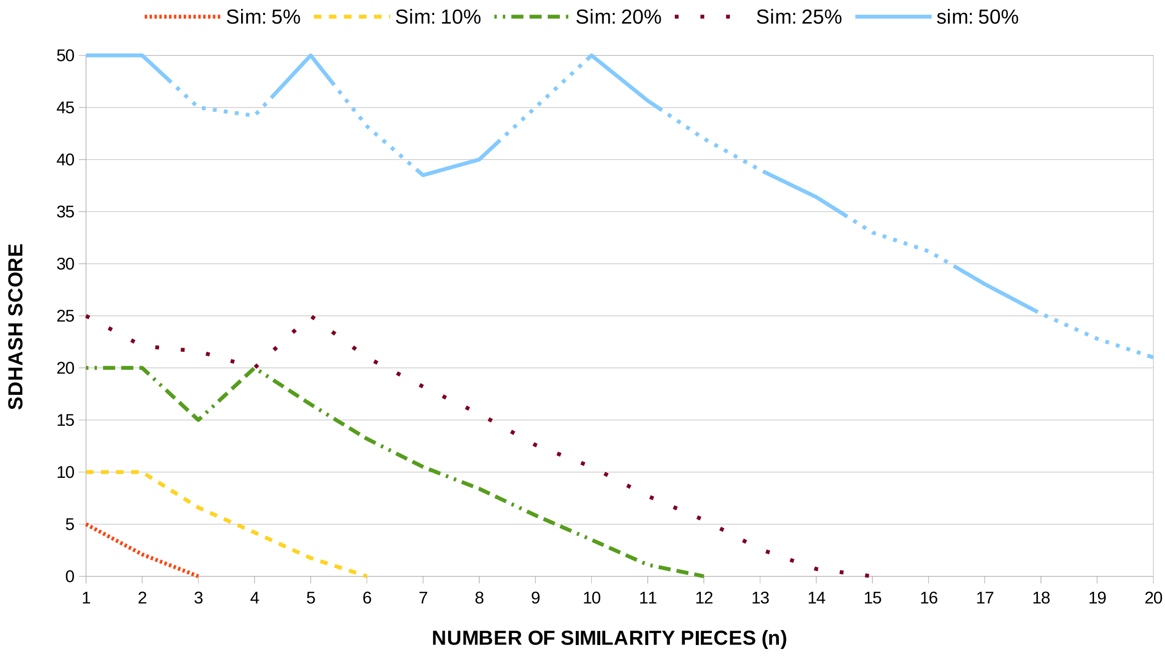


Figure 52 – Similarity fragmentation and its impact on `sdhash` score for different amounts of similar content (object size: 20 bloom filters).

the major factor contributing to the unexpected behavior of `sdhash` when dealing with small objects is α adoption; by removing its influence, the score approximates to real similarity value at the cost of an increase in the number of false positive matches (as it will be discussed and mitigated in Sec. 6.6). When the similarity is spread over many parts of the object, the same conclusions hold for small n; however, as n increases, the score tends to drop to its minimum value.

6.4 sdhash source code analysis: Limitations of current implementation

We analyzed `sdhash` source code (v.3.4[2]) for a better understanding of its working process and main functions, such as the ones related to the digest generation and comparison processes. Besides, a deeper analysis helped us to understand `sdhash` limitations.

Roussev, V. and Candice, Q. (ROUSSEV; QUATES, 2013) mention that `sdhash` only works for objects larger than 512 bytes; indeed, this holds for the current implementation. However, to compare two digests, `sdhash` requires that each has at least *min_elem_count* features (16 by default)[3]. Given that we have a feature for every 64 bytes (see Sec. 6.3.1), the minimum object size required for attending this requirement would be an object with the size

[2] GitHub page: <https://github.com/sdhash/sdhash/releases/tag/v3.4> (last accessed: Oct 23, 2019)

[3] see `sdhash` *sdbf_core.cc* file, function *sdbf_max_score*.

of $(16 \cdot 64 =)$ 1024 bytes (on average). We argue that although a digest is produced for smaller objects, it has no use if we cannot compare it and establish its similarity with others.

Another limitation of sdhash that we found out by inspecting its source code is the creation of new bloom filters when creating the similarity digest. Looking at the code[4], we discovered that an additional bloom filter (after the first one) is only created if the number of features that will be inserted into this new filter is at least $max\_elem/8$. Current sdhash implementation sets max_elem as the maximum number of features inserted into a filter (by default, 160); this way, the minimum number of features necessary for creating a bloom filter is 20. This design choice is justified in the source code by a commentary stating that it aims to reduce false positives. However, we argue that relevant information at the end of an object may be lost since many features are dropped out when the minimum value is not achieved.

6.5 Evaluation of sdhash for the different forms of similarity

In this section, we present a complementary theoretical evaluation of sdhash and discuss how this tool would perform on each form of similarity presented before.

6.5.1 Single common block correlation

Given two objects F_x and F_y sharing only a single common piece of data of $d_{x,y}$ bytes, what is the smallest value of $d_{x,y}$ for which sdhash can reliably correlate the two targets as similar?

We consider similar any match with $score \geq 1$. We also analyzed the matches having the recommended score ($score \geq 21$). To find out the minimum $d_{x,y}$ size for the two required scores, we use the analysis performed in Sec. 6.3.3.3, which shows how sdhash behaves for the single common block correlation test for different object sizes.

Our results show that the value of $d_{x,y}$ varies according to the objects' size. In general, the common piece percentage (concerning the smallest object size) varies from 0.50% to 37.50% for having $score \geq 1$ and from 20.50% to 50% for having $score \geq 21$. The significant decrease in the percentage value was obtained by increasing the object size from 10 KiB to 1.95 MiB, for the latter one having the smallest percentage requirement. The similarity requirements for larger objects tend to hold the same (minimum of 0.50% and 20.05%).

6.5.2 Fragment detection

Consider an object F_x with $s(F_x)$ bytes and a fragment of it ($Frag_x$) with $s(Frag_x)$ bytes ($s(F_x) >> s(Frag_x)$). Since sdhash considers the smallest object size for computing the

[4] see sdhash *sdbf_core.cc* file, function gen_chunk_sdbf.

similarity score, we need to know the smallest $s(Frag_x)$ for which sdhash still correlates it to its source (having $score \geq 1$).

We know from Sec. 6.4 that the current implementation of sdhash requires at least 16 features for creating a valid digest, i.e., one that can be compared to another to assess similarity. A simple conversion of the number of features to bytes gives us the smallest $Frag_x$ size. However, we first need to make some assumptions: (1) there is no change in $Frag_x$ bytes in relation to its source F_x; (2) sdhash maps 160 features to every 10 KiB of data (BREITINGER *et al.*, 2014c), which is a feature for every 64 bytes (on average). By assuming the two aforementioned points as true, the smallest size of $Frag_x$ is $(16 \cdot 64 =)$ 1024 bytes. Due to the limitations on our formulas to perform sdhash estimations for small objects (see Sec. 6.3.3.2 - Linear Regression), we will adopt as the minimum amount of features the value 20; we argue that doing so, we have only a small increase of a few bytes $(20 - 16 \cdot 64 = 256)$ in the object size in contrast to more accurate estimations. This way, the smallest object size will be $(20 \cdot 64 =)$ 1280 bytes.

Having a maximum score when comparing the fragment to its source requires that all similar features shared between the objects be stored on a single bloom filter in their corresponding bloom filter sets (or in subsequent ones in case the number of similar features exceeds the maximum allowed by the filters). However, such a scenario is unlikely to occur unless the fragment was extracted from the beginning or end of the source object; otherwise, the features will be distributed between two or more filters. In the worst case, half of the similar features are placed in one filter of F_x and the other half in another. In such a case, sdhash still produces some similarity, but with a reduced score; using equations presented in Sec. 6.3.3.3, we come up with a score for such a scenario of $score = 21$, which is still an indication of strong similarity.

Due to the file type and sdhash feature selection process, we can not show that the limits presented here will always stand in any condition. The score may have a small reduction due to the border features (Sec. 6.3.3.5). However, we believe our assumptions will stand most of the time. Besides, it gives us a general idea of sdhash limitations and what to expect from it when assessing similarity.

6.5.3 Alignment robustness

Given two equal objects F_x and F_y, how many extra bytes of any type have to be inserted at the beginning of F_y to get a $score = 0$ on sdhash? To answer this question, we need to consider three scenarios involving F_x size ($s(F_x)$):

- $s(F_x) < 10$ KiB (at most one full bloom filter)
- $s(F_x) = 10$ KiB (exactly one full bloom filter)
- $s(F_x) > 10$ KiB (object having more than one bloom filter)

In the first scenario, we started with a minimum object size of $s(F_x) = 1024$ bytes and added bytes (to the limit of 10239 bytes) to verify the minimum number of insertions before `sdhash` reduces the score to zero. Using equations presented in Sec. 6.3.3.3, we estimated that adding a sequence of 2240 bytes at the beginning of F_y (having no similarity with F_x) is the minimum value to produce a zero score (218.75% of the smallest object size).

For the second scenario, when F_x size is about 10 KiB of data (a full bloom filter), `sdhash` will always produce $score \geq 1$. Since the original object (F_x) has only one bloom filter, adding bytes at the beginning of a copy of it will at most reduce the score, but never turns it to zero. As we add bytes to F_y, we increase its size, and, at some point, a new bloom filter is created to store all of its exceeding features. For the alignment robustness test, a similar part will always be located on the last filters and will match the single filter of F_x. Since `sdhash` requires at least 20 features to create a new filter after the first one (Sec. 6.4), while the bytes addition do not produce this minimum number of features, the similarity is reduced because the last bytes of F_y will be ignored. Specifically, from the 10240 bytes of the original object, about $(10240 - 20 * 64 =)$ 8960 will still be similar to the source object before the next filter is created. In this condition, where we have about 140 similar features on a filter (worst case scenario), `sdhash` produces a $score = 78$. When a new bloom filter is created, all bytes set in this new filter will be the same as those of the F_x filter, producing a score of 100 (which is also the final score since F_x has only a single filter).

For larger objects, considered in the third scenario, the worst case is when the insertion of bytes shifts 80 features from one bloom filter (bf_i) to another (bf_{i+1}). Note that shifting more features will make bf_{i+1} the most similar to the original filters. In such a case, consider that F_x has three bloom filters. When we add bytes to F_y so that about 80 different features are increased on it, the bloom filters comparison depicted in Fig. 53 is performed.

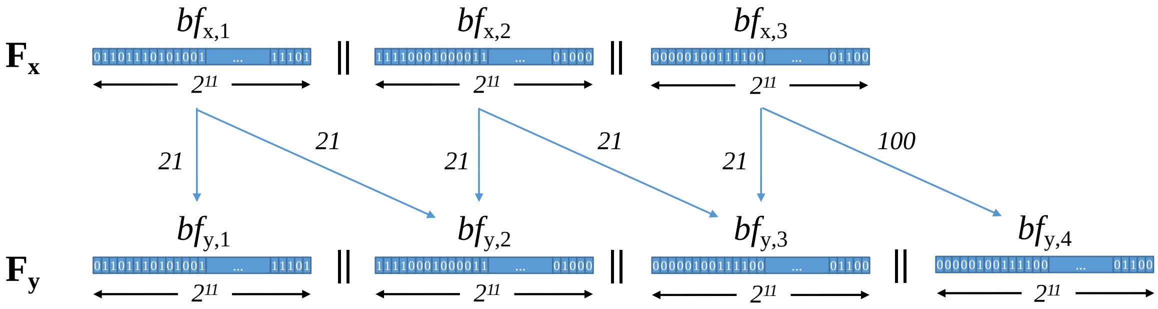


Figure 53 – Bloom filter matching in the alignment test.

We highlight that although the second object has $|bf_y| = 4$, the last filter has only 80 features mapped into it (since we inserted 80 newer and different features at the beginning of F_y, shifting all other ones). In the figure, we present only the bloom filter comparisons with a meaningful comparison result (≥ 1). We take into consideration the maximum value of each bloom filter comparison (all-against-all fashion) for computing the final similarity score of both objects. In this example, the maximum values are the following ones:

- $(bf_{x,1} \quad vs. \quad bf_{y,1})$ =21 or $(bf_{x,1} \quad vs. \quad bf_{y.2})$ =21
- $(bf_{x,2} \quad vs. \quad bf_{y,2})$ =21 or $(bf_{x.2} \quad vs. \quad bf_{y,3})$ =21
- $(bf_{x,3} \quad vs. \quad bf_{y,3})$ =21 or $(bf_{x,3} \quad vs. \quad bf_{y,4})$ =100

By averaging results considering $|bf_x|$ of the smallest object, we have $score = 47.33$. We argue that this value may change due to the border features, but we expect only a slight variation.

Our assumption for the third sceario can be corroborated by the FRASH framework test (BREITINGER *et al.*, 2013) results, where the authors found similar values as the one we got theoretically for the same test.

The `sdhash` tool worked perfectly (for object sizes larger than 10 KiB) in the alignment test. Even for the first scenario, where `sdhash` have limitations when dealing with small object sizes, it had good results, allowing the similarity detection even after a significant amount of bytes being inserted at the beginning of the object (about 218% of its original size).

6.5.4 Random noise resistance

Suppose that we have two identical objects F_x and F_y. What is the minimum number of changes one has to perform on F_y (in a random way) to get a non-match using `sdhash`?

The three basic manipulations that can be performed on objects are addition, removal, or modification of bytes. These three kinds of manipulations have slight differences with respect to the impact in the feature extraction and selection processes, as explained below. Remember that β is the feature size (bytes), W the feature selection window size, and features are hashed before inserted into bloom filters, so changing one of its bytes, will change its hash completely.

Modification Changing one byte to another alters β features in the feature extraction process.

Addition The addition of a new byte on an object creates a new feature and changes another l features in the extraction process, where l can be obtained from Eq. 6.10. It may also change the bloom filter alignment.

Removal The removal of an object byte eliminates one feature and changes another l features in the extraction in the feature extraction process, where l can be obtained from Eq. 6.10. It may also change the bloom filter alignment.

$$l = \begin{cases} F_x^{(offset)} & if(F_x^{(offset)} < \beta) \quad OR \quad (s(F_x) - F_x^{(offset)} < \beta) \\ \beta - 1 & \text{otherwise} \end{cases} \tag{6.10}$$

where $F_x^{(offset)}$ is the position of file x where the byte was manipulated (starting in zero), and $s(F)$ being the object size (bytes).

When changing objects, such as a text file, users tend to perform the three aforementioned manipulations. For this reason and simplicity, we will consider that any change can be performed, and all of them will have the same effect on the digest creation: A change of β features. We argue that, although they have different effects and some of them may increase/decrease object size, on average, we expect that adding and removing characters produce, in the end, an object with similar size to its original value (one operation will add one new byte while the other remove one byte).

The `sdhash` tool selects a feature to compose the object digest for every 64 bytes on average (Sec. 6.3.1). With this consideration in mind and that $\beta = 64$ on current `sdhash` implementation, we expect that each change on object bytes (addition, removal, or modification) alters β extracted features and at least one of these features be selected to compose the object digest (at most $\lfloor \beta/t \rfloor$ =3 features can be selected). Therefore, for every single byte changed on F_y, there will be one less feature in common when comparing F_x to F_y. Here, we are considering the worst case scenario, where the changed bytes are spaced by β bytes; changing, for example, two consecutive bytes will have a minimum impact on the feature extraction and selection processes, since only $\beta + 1$ features would change. We use equations from Sec. 6.3.3.3 to make our estimations. Table 35 shows our results on the random noise resistance test for some object sizes.

Object size (bytes)	**# random changes**	**Percentage (%) of object size**
1280	15	1.17
4096	44	1.07
4480	48	1.07
5120	55	1.07
5760	61	1.06
7680	81	1.05
9600	100	1.04
10240	104	1.02
20480	208	1.02
204800	2089	1.02
358400	3655	1.02
768000	7832	1.02
1048576	10731	1.02

Table 35 – Estimating the number of random changes supported by `sdhash` to maintain the similarity indication.

Our results show that more than 1.02% of changes (related to object size) are necessary to get a non-match between identical objects with sdhash. The results displayed here are corroborated by empirical tests performed by Breitinger, F. et al., who show that although sdhash deals well with random noise of up to 1.0% of object size, it struggles with noises of 2.0% or more, with no similarity detected (BREITINGER *et al.*, 2014a). Our theoretical results are close to the literature but presented lower values since we considered the worst case scenario: The bytes changed are separated so that there is no intersection between their feature sets.

6.6 Improving the similarity comparison function of sdhash by using Jaccard similarity

Given sdhash limitations discussed along this chapter, we propose here a new approach for computing the similarity score which mitigates current constraints. Our goal is to produce a comparison result that is: (1) Easier to interpret (a percentage value that represents the real similarity) and (2) capable of distinguishing the kind of similarity detected (containment or resemblance).

To achieve our goals, we will: (1) Use Jaccard similarity as a new form of estimating similarity between bloom filters; (2) produce two scores related to the kind of similarity detected; (3) show that, by removing the influence of α from the similarity computation, the score becomes closer to the real similarity (see Sec. 6.3.3.4) and (4) show other practices that can be employed to prevent the increase of false positive matches.

To accomplish our goals and show the feasibility of our proposal, we modified sdhash original implementation and replaced Eqs. 2.8 and 2.7 by a new approach, where we estimate the number of features in common between two filters (using Eq. 6.4) and then compute the Jaccard similarity, as defined next.

6.6.1 Jaccard similarity

Given two sets A and B, we can establish their Jaccard similarity (i.e., number of elements in common) by using one of the following equations:

Jaccard resemblance (BRODER, 1997):

$$J_r(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{6.11}$$

Jaccard containment (AGRAWAL *et al.*, 2010):

$$J_c(A,B) = \frac{|A \cap B|}{|A|} \tag{6.12}$$

where $|\cdot|$ denotes the cardinality of a set. If A is a subset of B, then $|B| \geq |A|$. The $J_c(A,B)$ definition considers that the set A is the smaller one.

The new `sdhash` version that computes the Jaccard similarity is called `J-sdhash`[5]. The new tool was developed from `sdhash` source code; it computes resemblance and containment similarity, as explained below.

Consider two digests SD_x and SD_y having $u = |bf_x|$ and $v = |bf_y|$ bloom filters each. The basic idea of our similarity measure is to estimate the number of features in common between the bloom filters sets of both digests and apply Jaccard equations to come up with a similarity score. More specifically, we estimate the number of features in common $G(bf_i, bf_j)$ (Eq. 6.4) for the first bloom filter of SD_x with all other filters of SD_y and select the maximum value; next, we proceed to the second bloom filter of SD_x and perform the same process with all other filters of SD_y; after comparing all filters from one digest to all of the other, we sum up all maximum values to get g_{common} as described in Eq. 6.13. Finally, we apply Eqs. 6.14 and 6.15 (based on the definition of Jaccard similarity) to compute the resemblance and containment scores. We also need to provide the total number of features mapped in each digest ($|F_x|$ and $|F_y|$).

$$g_{common}(SD_x, SD_y) = \sum_{i=1}^{u} max_{1 \leq j \leq v}(G(bf_{x,i}\ ,\ bf_{y,j})) \tag{6.13}$$

$$J_r(SD_x, SD_y) = 100 \cdot \frac{g_{common}}{|F_x| + |F_y| - g_{common}} \tag{6.14}$$

$$J_c(SD_x, SD_y) = 100 \cdot \frac{g_{common}}{min(|F_x|, |F_y|)} \tag{6.15}$$

Here, $min(\cdot,\cdot)$ returns the size of the smallest set.

Note that J_r and J_c scores are decimal values varying from 0.00 to 100.00.

6.6.2 Evaluation

The adoption of the Jaccard as a similarity measure makes the interpretation of the score easier by practitioners as well as produces results closer to the real similarity found on objects. The first hypothesis is corroborated by the fact that the Jaccard can behave as a percentage value, which is easily understood and a well-established metric; besides, it does not suffer from the object size variation as the `sdhash` score does. The second hypothesis is demonstrated next, where, for different variations on the object size and type of similarity, we show how `J-sdhash` behaves and how it is closer to the real similarity value. In the following, we introduce the metrics used to evaluate our tool and then show results.

[5] Source code is available at our GitHub page: <https://github.com/regras/J-sdhash>

6.6.2.1 Metrics used for evaluation

On chapter 5, we presented and made use of two well-known metrics of the Information Retrieval field: recall and precision. In this section, we adopt the same metrics and, since recall and precision only provide a binary answer considering whether or not a match is similar, we complement our analysis with another metric that considers the quality of the score instead. Here, we propose a new metric (*M*) that considers how the score of AM tools are close to the real similarity shared between objects. Eq. 6.16 defines this metric.

$$M = |score - sim| \tag{6.16}$$

where *score* is the AM tool score and *sim* the known similarity of a match (percentage of bytes that are the same). The metric *M* measures how distant the score reported by the AM tool is from the real similarity; the shortest the distance, the better. Using this equation, we can measure the quality of the score of each tool, analyzing how far from reality the tools' comparison results are. Note that *M* ranges from 0 (perfect value) to 100 (worst case).

To have a better idea of a tool's performance under *M*, we can use Eq. 6.17, which is basically an average of *M* for several comparisons.

$$diff = \frac{1}{q} \cdot \sum_{i=1}^{q} M_i \tag{6.17}$$

where q is the number of comparisons performed in the data set with $score \geq 1$.

Since metric *M* requires the knowledge of the ground truth of the data set used for evaluation (the real number of bytes similar among all objects), we use here a synthetic data set (Sec. 6.2.3), where the exact percentage of similarity among the objects is known (and controlled). Next, we present the evaluation of `sdhash` and `J-sdhash` under the single common block and random noise resistance tests, varying the amount of similarity on objects, their sizes, and position where the similar content is placed.

6.6.2.2 Tests, setup and application

The tests performed in this chapter used a machine running Elementary OS 0.4.1 Loki 64-bit (built on Ubuntu 16.04.2 LTS), with an i7-5500U CPU @2.40 GHz processor, 8 GB of memory, 1 TB SATA 3GB/s hard disk drive (5,400 rpm), and NVIDIA GeForce 920M. The `J-sdhash` was implemented by modifying the `sdhash` source code (written in C++ language) in only some specific parts related to the digest comparison function. The developed tool (piece of code changed) and all scripts used to create the synthetic data set used for the evaluation can be found in GitHub[6].

[6] GitHub page: <https://github.com/regras/J-sdhash>

6.6.2.3 Results

We used `sdhash` and `J-sdhash` to create and compare digests of our synthetic data set (Sec. 6.2.3) and then we evaluate both tools using the metrics presented previously: Recall, precision, and the M metric. Results are presented and discussed in this section. We highlight that although `J-sdhash` produces two scores for each object (resemblance and containment), the evaluation and comparisons performed here will only take into consideration the containment one. We argue that containment measure can better express the similarity of objects with the same or different sizes; furthermore, we expect the same results for resemblance since we estimate both in the same way. Besides, both metrics have different purposes and consider similarity in different ways; for instance, two objects having $|F_x| = |F_y|$=100 features and 50% of similarity, will have a containment score of 50%; for resemblance, the score will be of 33.33%. Since `sdhash` uses the size of the smallest object, we believe it is closer to containment and hence this is the metric used for comparisons here.

6.6.2.3.1 `sdhash` vs. `J-sdhash` with no α influence

The first comparison performed considered the `sdhash` comparison function in its pure form, with no influence of the α variable on the score computation; this means that $C = e_{min}$. We have shown in the previous sections that the addition of α makes the score behaves inconsistently concerning the percentage of real similarity shared between small objects. We removed this variable from the calculation of the `sdhash` score and compared both tools (containment score). The comparison here is limited to the synthetic data set in the single common block scenario, where the similar piece is placed at the beginning or middle of objects. Results are presented in Table 36 for precision, recall, and $diff$ (average of M, Eq. 6.17).

Metric	**Sim. at beginning**		**Sim. at the middle**	
	`sdhash`	`J-sdhash`	`sdhash`	`J-sdhash`
Recall	97.73%	97.73%	97.73%	97.73%
Precision	1.16%	1.16%	1.16%	1.15%
diff (all matches)	6.42	7.81	6.66	8.10
diff (only true matches)	3.32	4.26	6.18	6.71

Table 36 – Comparison of `sdhash` (with no α influence) and `J-sdhash`. Analysis of the number of matches and quality of the score for the synthetic data set (44 similar matches).

We can see by our results that `sdhash` performed better than `J-sdhash` when we removed α from the similarity score calculation. The reason for `sdhash` overcoming the proposed tool is due to the way the score is produced. By removing α from consideration, `sdhash` computed the score based on the proportion of overlapping bits set on the bloom filters under comparison in relation to e_{min} (Eq. 2.6) and e_{max} (Eq. 2.5). On the other hand, `J-sdhash` performed an estimate of the number of features in common between the filters based on the number of overlapping bits, producing an approximation of the real value. Since the result is based on probabilities, there will be a small difference between the real and estimated values in some cases, causing J-sdhash to perform worst.

Our results also show that both tools produced small values for $diff$ (considering only similar objects); besides, the tools found almost all matches; only a single match between two 10 KiB objects sharing 1% of similarity (only 102 bytes) was missing. Even though we removed α, we still need 192 bytes of equal bytes between objects so that sdhash can produce *score* ≥ 1 (see Sec. 6.3.3.4). The down side of both tools was the poor results for precision, which was expected due to the removal of α from sdhash, as noted by Roussev, V. (ROUSSEV, 2010), and a similar mechanism for dealing with false positives to J-sdhash.

6.6.2.3.2 sdhash vs. J-sdhash with α influence

In this scenario, we use the original version of sdhash. We also adapted the α parameter as a mechanism to reduce false positives in J-sdhash, where the tool ignores matches of two bloom filters with less than C (Eq. 2.7) bits in common. By using α, we had a better performance, especially for J-sdhash. Table 37 summarizes the results for the synthetic data set (single common block scenario).

Metric	**Sim. at beginning**		**Sim. at the middle**	
	sdhash	J-sdhash	sdhash	J-sdhash
Recall	88.64%	88.64%	88.64%	90.91%
Precision	100.0%	100.0%	100.0%	100.0%
diff (all matches)	4.16	2.17	8.43	6.71
diff (only true matches)	4.16	2.17	8.43	6.71

Table 37 – Comparison of sdhash and J-sdhash considering α in the score computation of both tools. Analysis of the number of matches and quality of the score for the synthetic data set (44 similar matches).

The $diff$ metric for the whole set was about 48% smaller for J-sdhash than sdhash in the scenario having the similarity placed at the beginning of objects and 20% smaller considering the similarity in the middle. Besides, we got maximum precision for both tools, although recall decreased a little; we missed five matches involving the 10 KiB object with small similarity percentages (30, 20, 10, 5, and 1%) in most scenarios, except for J-sdhash that found the 30% match (similarity placed at the middle). Both tools perform better in the case of similarity at the beginning because of the impact of border features. In the first scenario, we have the influence of only one border in comparison to the two borders of the second scenario, which makes detection harder.

As we can see, J-sdhash had a better performance than sdhash when we removed false positives by increasing the minimum number of overlapping bits between two filters (use of α). Such an approach is required when comparing many objects due to the huge number of false positives, which decreases precision to levels not allowed in practice. Given the same conditions as sdhash, we showed that J-sdhash performed better.

We highlight that using the default cutoff point C set to 0.3 requires at least 56 features in common (about 35% of the total number of features inserted in a filter) in two bloom filters for $score \geq 1$. We showed along this work that using α is inconsistent with small object sizes, and it is not straightforward to change the default value and understand its impact. Given the proposed method to compute similarity and to help practitioners to control the amount of similar content two objects must share to produce a match, we present below a different form of filtering false positives more adequate to J-sdhash.

6.6.2.3.3 Calibrating J-sdhash to remove false positives

Given that many false positive matches occur when using J-sdhash without a mechanism like α, we need to establish a method for filtering the most relevant results. Here, we propose a technique to help to filtering results that is easier to understand and modify by practitioners, which will be able to adapt the tool according to the investigation needs.

We propose using a predefined percentage of features in common necessary to consider two bloom filters similar. To this end, the practitioner must define γ, the minimal percentage of features in common to J-sdhash consider two filters as similar. More specifically, we convert γ to number of features and apply Eq. 2.3 to estimate the corresponding number of bits set; two bloom filters must have at least the estimated amount of bits set to one in their intersection for having a valid match; otherwise, we consider them as different.

Finding the best γ depends on the purpose of the investigation and the minimum amount of content a practitioner is interested in. Note that using low values for γ may result in many false positives, while high values filter out most unwanted matches (along with some similar cases having small similarities). Tables 38 and 39 show the same parameters as before for some γ settings for the cases with a single common block placed at the beginning and middle of objects, respectively. The use of α has approximately the same impact as adopting γ=35%.

Metric	sdhash	J-sdhash			
	(γ=35%)	**γ=30%**	**γ=20%**	**γ=15%**	**γ=10%**
Recall	89.0%	90.91	93.18%	93.18%	95.45%
Precision	100.0%	100.00	36.94%	10.87%	1.93%
diff (all matches)	4.16	1.55	1.35	0.86	1.65
diff (only true matches)	4.16	1.55	1.19	1.20	1.31

Table 38 – J-sdhash settings varying the minimal percentage of features in common for similarity in the beginning of objects (44 similar matches).

Metric	`sdhash`	`J-sdhash`			
	(γ=35%)	**γ=30%**	**γ=20%**	**γ=15%**	**γ=10%**
Recall	88.64%	93.18%	93.18%	95.45%	97.73
Precision	100.0%	100.00%	22.04%	8.57%	1.86
diff (all matches)	8.43	6.50	1.54	0.78	2.45
diff (only true matches)	8.43	6.50	6.46	6.45	6.36

Table 39 – `J-sdhash` settings varying the minimal percentage of features in common for similarity in the middle of objects (44 similar matches).

From these results, we can see that as we decrease γ for `J-sdhash`, the better the recall but the worst the precision. Requiring fewer features in common between filters allows us to find matches with a small content in common, which was reflected in an increase in recall when decreasing γ from 30% to 10%. However, by reducing γ, we got poor results for precision due to many matches with an insignificant amount of similarity, such as 0.09%, for example; we believe such values have no use and could be excluded from consideration.

All versions of `J-sdhash` presented better results than `sdhash` concerning the quality of the score; the $diff$ value was smaller for the cases considering similarity at beginning and middle: For γ=30%, that presented the best relation recall and precision, we had a reduction of 62.74% and 22.89% in $diff$ in comparison to `sdhash` for the same value of precision; `J-sdhash` had a better recall in all presented cases.

6.6.2.3.4 Evaluating `J-sdhash` for other scenarios

Two more scenarios are presented here, comparing `sdhash` to `J-sdhash` regarding the quality of the score. The first one is for the same single common block presented before, with the difference that the similar block is placed at the end of the objects. Table 40 presents the results, where the same behavior of the two previous scenarios is observed: as we increase γ, the precision gets better, but recall gets slighted worst. The γ=30 setting has the same precision and recall as `sdhash`, with a significant reduction of 43.35% on the average of the M metric. Other settings, although recall and $diff$ were better, had poor results of precision.

Another scenario is presented in Table 41, where we compare a synthetic data set under the random noise resistance test. In this new form of similarity, a given object is duplicated and its new version is randomly modified (with the change of r% bytes); then, the original and modified objects are compared under the AM tools.

The data set used for this experiment is described in Sec. 6.2.3. For each object of this set, we created 11 new versions of it and compared only the given object to its modified versions so we still have control of the level of similarity shared between the objects; comparing all objects in an all-against-all manner would produce many additional matches since all 11 modified versions have bytes in common among themselves on different levels that we did not know in advance. In the end, we had 2560 similar matches with no false positive.

The results for the random noise resistance test show that `J-sdhash` had a better performance than `sdhash` for both recall and the M average for all of its settings. As we decrease γ, the better the recall, and the closer the score gets to the real known similarity. For γ=30%, we had $\approx$ 38% reduction in the distance between score and real similarity. For other settings, the reductions achieved $\approx$ 40%. Again, `J-sdhash` was superior in all aspects considered.

Metric	**sdhash**	`J-sdhash`			
	γ=35%	**γ=30%**	**γ=20%**	**γ=15%**	**γ=10%**
Recall	90.91%	90.91%	93.18%	93.18%	93.18
Precision	100.0%	100.0%	27.52%	7.79%	1.78
diff (all matches)	13.34	7.29	3.23	1.13	1.92
diff (only true matches)	13.34	7.29	6.92	6.90	6.52

Table 40 – `J-sdhash` settings varying the minimal percentage of features in common for similarity it the end of objects (44 similar matches).

Metric	**sdhash**	`J-sdhash`			
	γ=35%	**γ=30%**	**γ=20%**	**γ=15%**	**γ=10%**
Recall	99.65%	99.69%	99.88%	99.96%	100.00%
diff (similar matches)	30.00	18.67	18.04	20.35	17.91

Table 41 – `J-sdhash` settings varying the minimal percentage of features in common for the random noise resistance test (2560 similar matches not taking into consideration false positive results: Precision = 100%).

For the experiments performed here with some `J-sdhash` settings, where we varied the γ parameter, we had the best recall and precision relation for $\gamma = 30\%$, which is the value that we suggest for being used on digital forensic investigations. However, such value can be easily changed and adapted to work according to the investigation's needs.

6.7 Discussion

Along this chapter, we presented a study of several aspects of `sdhash` and reached some conclusions. In this section, we focus on providing answers to our research questions presented in Sec. 6.2.1.

Our work focused on a theoretical analysis of the `sdhash` tool and towards solutions to improve it. By understanding the inner processes of `sdhash`, we manage to estimate the minimum amount of similarity it can detect, as shown in Sec. 6.3.3.3. With such an analysis, we answer RQ.1 *"What is the minimum amount of similarity that `sdhash` can detect?"* We also showed the different scenarios in which the tool works and the minimal conditions for its operation, where the object size plays an important rule (RQ.2 *"What to expect from `sdhash` when assessing similarity? In which cases/scenarios it will work?"*). We showed that for small objects, `sdhash` tends to produce score values different from the real similarity found in objects. Besides, such objects require many similar bytes to produce valid scores, while larger objects require lesser (in relation to a percentage of the object size); about 35% of similar data is required for small objects while only 0.50% is needed for larger ones to produce a score ≥ 1.

In Sec. 6.3.3.4, we answered RQ.3 *"What is the impact of the α variable (and its default value) on the similarity assessment?"* by showing the impacts of the α parameter (with its default value) on the similarity assessment. It is one of the main causes of the inconsistency found between the similarity score and the real similarity values of small objects.

Sec. 6.6 focused on improving `sdhash` by changing its comparison function to the Jaccard Similarity. We say *yes* to RQ.4 *"Can we improve the similarity assessment process of `sdhash` to make it easier to understand and accurate?"* by showing how `J-sdhash` presents an easier to interpret, more adaptable, and accurate similarity value (based on the difference between a score and real similarity).

6.8 Conclusions

In this chapter, we presented a theoretical analysis of one of the most popular approximate matching tools, `sdhash`. In detail, we showed the internal process of creating and comparing digests with `sdhash`, a deep analysis of its detection capabilities, and limitations. We also proposed a new version of `sdhash`, where we replaced its comparison function for the Jaccard similarity. The new application, called `J-sdhash`, has a more easy to interpret score, can be adjusted to work in different scenarios, and produces results closer to the real similarity found in objects. Given all details provided here about `sdhash` performance on the detection of similarity and with the newly developed version of it, we hope that practitioners can perform a similarity search in digital forensic investigations more efficiently according to their needs.

7 New applications for Approximate Matching functions

7.1 Introduction

In this chapter, we present two new applications for AM. We start by providing an approach for fast file identification on forensics investigations, using sampling techniques; then, we move to the fingerprint realm and propose a new strategy to perform fingerprint identification in large data sets efficiently. We finish this chapter by presenting conclusions and possible new applications for AM as future work.

7.2 Fast file identification with AM

A new application for AM is fast file identification using sampling techniques to reduce the amount of data analyzed in an investigation. Instead of examining all objects from a seized media, we extract random samples of it to make a statement about the entire population. We aim to implement the same idea of sampling discussed in Garfinkel's work (GARFINKEL *et al.*, 2010) but using AM instead of block hashes to increase the effectiveness of the search for shreds of evidence. We also use a sector level approach in order to detect even small parts of the target data. This way, we expect to reduce the amount of time to pursue a triage process and to increase the accuracy of the search, where shreds of evidence with small changes will not confuse the digital forensic practitioner on an investigation.

7.2.1 Approximate Matching and Digital Forensics

AM tools have great potential to be used in digital forensic investigations to identify similarity and embedded objects. However, besides all advantages and flexibility achieved with AM, its high cost when compared to cryptographic hash functions is a major problem. For example, one of the most known and used tools of AM is `sdhash`. For a certain 1 MiB object, we have about 16.384 features produced by `sdhash`; as each feature needs to be hashed before inserted in the bloom filter, a total of 16.384 hashes need to be performed (for more details about `sdhash` working process, see Chapter 2). In this case, this tool will be at least 16.384 times more expensive than the traditional SHA-1 hash function used in such context. As the object size grows, so the number of hashes and the costs (in time) of investigations. This example shows how `sdhash` can be expensive, although it presents desirable detection capabilities; therefore, ways for improving somehow its efficiency are necessary.

Another problem with sdhash is the size of the similarity representation. Hash functions produce fixed-size digests that are independent of object size. On the other hand, sdhash outputs a digest that varies between 2.6% (ROUSSEV, 2010) to 3.3% (BREITINGER; BAIER, 2013) of object size. Therefore, the object size is another problem in the use of such a tool, as it can be a bottleneck in the triage process. Given the constraints discussed so far, we propose a new way to deal with these problems by using statistical methods, as presented in the next section.

7.2.2 Sampling in digital forensics

To reduce the time taken for an examiner to classify a seized media as worth or not for a further and more in-depth analysis, we could reduce the amount of data processed. The task can be done following the ideas presented by Garfinkel et al. (GARFINKEL *et al.*, 2010), where the authors show how to adapt the classical *"Urn problem without replacement"* in the forensic context. They say that taking enough random samples from a set of objects, there is a good chance that these samples represent the entire data set satisfactorily. The amount of samples n required can be obtained from the following equation, which calculates the probability of missing one of the objects of interest (p).

$$p = \prod_{i=1}^{n} \frac{((N-(i-1))-M)}{(N-(i-1))} \tag{7.1}$$

Here, N represents the total number of objects in the set, M the number of targets (objects of interest), and n the number of objects required to be sampled. The value of p is set by the examiner and usually corresponds to less than 1%. We will use the same idea combined with AM to reduce the time for triage in investigations and yet have an efficient method for detecting objects that are the same/similar from a reference object. Our goal is to verify whether a media contains at least one object from a database of interest objects. If so, we can take this media for further analysis.

In the calculus, the variable N is related to the seized media size. We will work at the disk sector level; for this reason, N will be the number of media sectors. For determining the M value, also in sectors, we can use different approaches. The first one considers the case where we want to find a particular file, which is just using its size (in sectors) as the M value. Another scenario consists of figuring out whether a seized drive contains any file of a database or not. In such a case, we need to adopt the *"Urn Problem"* in this context. Since we do not know whether the media contains evidence or not, we have to consider that at least one file of a particular size will be present in the media in order to comply with one of the basic assumptions of the problem: The presence of an object inside the "*urn*", or in our case, the media. Then, we convert the chosen file size to sectors and use it as M (estimated value). Next, we calculate the formula using M to obtain n for a chosen probability rate. This means that we need to take n random samples from the media, and with probability $1 - p$, we will find files with the chosen size or larger (we can adjust this value as needed, but the n value will change). We highlight that the practitioner can control the M value according to the investigation, using an average file size from the database or a common size for the type of searched files for that particular scenario. Therefore, prior knowledge about the search can potentially help in determining the M value.

Using the *Urn Problem* equation and controlling its error rate, practitioners can look for objects using random samples with a high probability of success. For example, considering a 2TB seized disk (which contains approximately 500 million 4 KB sectors) and that we want to find a 100 MB file (25,000 sectors) on it. Using the Eq. 7.1 with N = 500,000,000, M = 25,000, and fixing a success rate of 99%, we need to take around 92,500 random samples from the hard disk drive in order to find at least one of those 25,000 sectors. If we see one such sector of interest, we can presume that the disk may contain the 100 MB file, and we can examine it more carefully.

7.2.3 Using sampling to reduce sdhash cost

To minimize the time to analyze a large volume of data, we propose an approach to combine the use of `sdhash` tool with sampling techniques. We will also work with object fragments (disk sectors) instead of whole objects since we want to encompass scenarios where the file system is corrupted, and no metadata about its structure is available. Besides, taking sectors directly from a hard drive gives us the possibility of finding files that were deleted, but whose fragments were not yet overwritten.

However, using sampling with object fragments can also be costly; the number of digests and comparisons between the data sets increases as their sizes grow, even though we use samples. With this in mind, we propose the use of clusters to mitigate such problems.

7.2.3.1 Clustering approach

We propose the use of clusters to reduce `sdhash` cost. We take a determined number of sectors from a seized media under analysis and gather them in a single object, which will have the similarity digest generated and used for comparisons. The fragments can (and most of the time will) be from different objects, and yet the `sdhash` tool will be able to identify similarity due to its characteristics.

Our objective is to figure out whether a seized media contains objects of interest based on a comparison to a database, in the shortest possible time. To this end, we randomly sample sectors from the disk, gather them into clusters of j sectors and generate their digests with `sdhash`. Then, we compare these digests to a database to find interest objects in common.

One question that stands is how to take the samples. We propose two different ways to perform this operation: diffuse or contiguous. In the first one (Fig. 54), we select objects (sectors) randomly around the seized media and group them into clusters, which will have their digest created and compared to a database of interest objects. The second method consists in taking contiguous fragments around some randomly chosen sectors to build the clusters (Fig. 55). It is important to highlight that for fragmented disks, both methods seem to work fine, while for a defragmented one, the second method will not get fragments from many different files and will break the assumption of a random selection of objects partially. In this work, we will only use the diffuse mode for sampling. The other method will be covered in future studies.

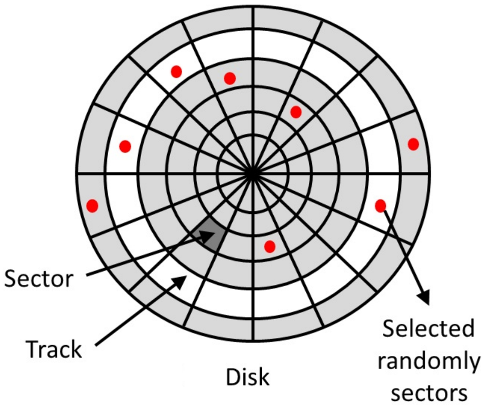


Figure 54 – Diffuse method of selecting sectors for a cluster.

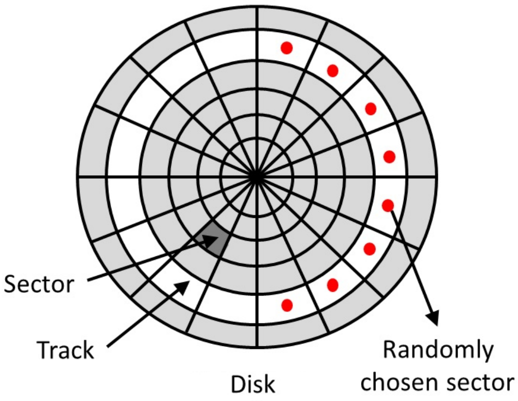


Figure 55 – Contiguous method for clustering sectors.

7.2.3.2 Experiments

We conducted experiments to validate our ideas using `sdhash` and sampling. To this end, we simulated specific scenarios to measure the efficiency of this approach in each one in order to find evidence in a triage process. Our goal is to find at least one fragment from a database of interest objects in the seized media being analyzed. If the search is successful, this media will be selected for further and more in-depth analysis.

We first compared our approach of using similarity digests and sampling to the common one where a similarity representation is generated for every object in seized media. Then, we did experiments with the first approach exploring different scenarios and evaluating the impact of using the clustering method.

In our experiments, we selected two data sets and labeled them as DBIO (Database of Interest Objects) and SM (Seized Media). The DBIO is the largest one, with 2,204 different files (218,458 sectors), while the second has 549 files (74,929 sectors). The data sets were taken from the *t*5 corpus (ROUSSEV, 2011) and encompass different file formats: `text`, `pdf`, `html`, `doc`, `ppt`, `jpg`, `xls`, and `gif`. For our experiments, the DBIO and SM data sets have only four files in common, with the following format and size: F_1: `doc` (1,039 sectors), F_2: `ppt` (469 sectors), F_3: `text` (87 sectors) and F_4: `pdf` (273 sectors). Our goal is to find at least one object in common in the shortest time possible. All files in the SM data set were fragmented (broken into 4 KB objects) to simulate disk sectors. For the DBIO data set, we disposed of the objects in three ways to simulate different scenarios: fragmented (4 KB), whole file and image file. The latter format is of a single and huge file having all objects together, one appended to another.

We first evaluated the approach of using `sdhash` without sampling. To this end, we followed the steps below.

1. Creation of a similarity digest for every object in the DBIO.
2. Creation of a similarity digest for every object of the SM.
3. Comparison of similarity digests between DBIO and SM objects.
4. Evaluation of the results.

The next experiment evaluated the inclusion of sampling in the process. The first step is to make a sample of the SM data set. To this end, we used the approach described in sec. 7.2.2. We first estimated the total number of objects we want to search and applied Eq. 7.1 using the SM number of 74,929 sectors. Next, we chose the number of 1,039 sectors (the number of sectors in the largest common file available in both data sets) and a 99,9% success rate. Using Eq. 7.1, we got a result of 550, which is the required number of samples we have to take from the SM.

Then, we did several other experiments based on the described approach with different scenarios to evaluate the impact of clustering and determine the best conditions for using it. In general, we did our search following the steps below.

1. Creation of a similarity digest for every object in the DBIO.
2. Sampling the SM data set.
3. Creation of a similarity digest for the samples and clusters of SM.
4. Comparison of similarity digests between DBIO objects and SM samples and clusters.
5. Evaluation of the results.

Our first experiment in this second round involved comparing the objects from the SM with those in DBIO data set in the fragment level, to simulate disk sectors (4 KB). Next, we set the cluster size c to 100 sectors (a reasonable value got from empirical tests). For larger values of c, we identified an increasing number of false positives (similarity between unrelated files); for smaller values, no significant reduction was achieved, only unnecessary comparisons had to be made. In the next experiment, we evaluated the use of whole files in the database instead of their fragments. Again, we did it with and without clusters. The last experiment compared the SM fragments to a DBIO image file. To this end, we created a similarity hash using the entire data set as if it was a single object to create a single representation of it. By default, `sdhash` breaks large files in 128 MB pieces and creates a similarity hash for each one of them (ROUSSEV; QUATES, 2013). This way, we ended up with a few digests that represented the whole image file.

7.2.4 Discussion

The results of our experiments are presented in Tables 42 and 43. In the first one, we present a comparison of `sdhash` with and without sampling, using DBIO and SM in the fragmented level. Notice that there is a difference in the maximum number of comparisons in both techniques for the expected value (16,368,839,482 without sampling and 120,151,900 with it). This is because `sdhash` does not work with objects smaller than 512 bytes, and the data sets have a few files which do not reach this minimum size.

When we analyze the impact of adding sampling, a significant reduction in the number of comparisons is noticed, requiring about 136 times fewer operations. Also, we present the number of objects of interest found by each technique, where we show that both found more objects than it supposed to. This exceeding value represents the false positives, and it is possible to see that the use of sampling reduces such value.

The cost of each technique is presented in the form of the number of hash functions (SHA-1) required to generate the digests, shown in the last three columns of Table 42. For the traditional approach using a hash-based method, the cost is equal to the number of objects: each one only requires a single hash to create its representation, resulting in $(218.458 + 74.929 =)$ 293.387 hashes. However, a high cost is observed using `sdhash` (about $(65.738.212/293.387 =)$ 224 times more expensive than the traditional technique), which is minimized by the adoption of sampling (reduced to $(49.071.984/293.387 =)$ 167 times). This way, it is evident that using `sdhash` is very expensive, and its combination with statistical methods becomes essential towards a practical approach. To this end, we focus on this combination and propose the use of clusters to reduce even more the costs, evaluating different scenarios to find the best ones in which we have the largest reduction in costs.

Table 42 – Experiments comparing `sdhash` with and without sampling on the fragment level, measuring the efficiency and the cost.

Technique	#comparisons (max.)	#Objs of interest found	#Hash functions (SHA-1)		
			DBIO	SM	Total
sdhash only	16.343.700.471	9114 / 1868	48.947.005	16.791.207	65.738.212
sdhash with sampling	120.031.450	62 / 22	48.947.005	124.979	49.071.984

We present in Table 43 the results of `sdhash` and sampling for different scenarios, evaluating the use of our clustering approach, where the maximum number of comparisons is reduced by a factor of c (size of the cluster). However, the clustering approach was ineffective in some of the experiments, as shown in cases #3 and #5. In such cases, no matches were found when comparing fragments with whole files and fragments with the image file on the DBIO set. We believe that this happened because fragments of different files were selected to be part of the same cluster due to the random selection process; when creating and comparing the digest of these clusters, the similarity with the whole object or image file was too small and could not be identified by sdhash. Only case #1, when the comparison is made considering the DBIO fragmented, we had significant results, as shown in Table 43, where almost all fragments of objects F_1 and F_2 could be found. Also, in the scenarios where we did not use cluster and consider the DBIO data set in whole file and image file formats (#4 and #6), the number of comparisons decreased, but the number of false positives (number of matches between scores 20 and 100) is elevated in relation to the other scenarios (#1 and #2).

Table 43 – Experiments with sdhash using clustering

#exp.	**#comparisons (max.)**	**#matches (score: 20-100)**	**Clusters**	**Database format**	**Objs of interest** F_1	F_2	**#Features**
1	1.309.434	107	Yes	Fragments	11(4)/4	20(15)/18	48.985.337
2	120.031.450	238	No	Fragments	17(4)/4	45(15)/18	49.071.984
3	12.870	0	Yes	Whole file	0/4	0/18	14.363.578
4	1.179.750	149	No	Whole file	5(4)/4	15/18	14.450.225
5	42	0	Yes	Image file	0/4	0/18	10.587.068
6	3850	288	No	Image file	4/4	17(15)/18	10.673.715

Table 43 also presents the total number of matches between the data sets according to a range of scores got from sdhash, where the results express the similarity between objects in the SM sample to those in the DBIO. We highlight that only scores of similarity ≥ 20 were counted since they are the ones significant (reliable) (ROUSSEV; QUATES, 2012) in the original version of sdhash.

We also show in Table 43 the number of matches restricted to the objects selected, where we present the total number of matches and the number of true positives (tp) matches (parentheses). We knew beforehand that there are only 22 objects in common between the data sets after the sampling, where 18 belongs to file F_1 and 4 to F_2. The other two objects (F_3 and F_4) had no fragments selected in the sampling. For example, in experiment #2, object F_2 had 45 matches for only 18 TP possible. Between parentheses, we show the number 15, which is the number of tp matches. Object F_1 was 100% covered in experiments #1, #2, #4 and #6, while F_2 was 83.34% covered for the same cases.

Comparing the expected number of matches to the one got from the total matches (third column), we can see a high number of false positives results (considering results with score ≥ 20). The high number can be justified by the presence of common blocks (see chapter 5), but future work is necessary to find out how to remove common blocks when sampling with AM and its impacts.

7.2.5 Partial conclusions

Digital forensics is becoming a critical field as its techniques need to scale to follow the fast increase in media storage capacity. In the first part of this section, we presented a proposal for using approximate matching tools (sdhash) combined with sampling techniques to reduce the time of a triage process. We showed how the expensive sdhash could be used and possible ways to reduce the overall cost of investigation, using sampling techniques and clustering disk sectors. Besides the reduction in the number of comparisons and features generated, the new approach showed effectiveness in finding similar objects of interest and presented a smaller number of false positives. Also, the proposed technique gives practitioners more flexibility and allow them to find even objects that were deleted while some of their fragments remain on disk. We also showed limitations of clustering by evaluating different scenarios, which proved to be ineffective when comparing clusters to database image files and whole files. On the other hand, we had good results comparing them to a fragmented database. Although our experiments have shown that sdhash could identify objects of interest with a small set of data taken randomly from a seized media, future work is necessary to corroborate our hypothesis that the false positives generated were indeed due to common blocks.

7.3 Approximate Matching for fingerprint identification

Another application for AM is for the identification of fingerprints. One challenging problem on the fingerprint realm is the identification of individuals over large databases, where the most similar template must be found. We believe AM can be used to identify similar fingerprints and become a promising technique to speed up searches. Here, we explore this hypothesis and present MCC-HBFT, a new fingerprint identification strategy based on the approximate matching technique HBFT and the state-of-the-art fingerprint representation model MCC. We show how MCC-HBFT identify fingerprints and outperforms a commonly used indexing strategy in some public databases.

7.3.1 Introduction

Fingerprint identification is one of the most well-known and publicized biometric traits due to its interesting characteristics: uniqueness, consistency over time, easy acquisition, and low cost. However, one problem that remains is how to search an unknown fingerprint over large repositories, which poses challenging obstacles regarding accuracy and efficiency. Identifying individuals in such a case requires the comparison of the input fingerprint templates to every other template in the database, in an all-against-all fashion. This process, often called brute-force, is ineffective and becomes impractical for large sets.

One trivial solution to overcome the problem mentioned above is to reduce the total number of comparisons by prefiltering techniques, such as exclusive classification. The issues with this approach are the fixed and small number of classes and the uneven distribution among them. A more efficient solution could be the use of indexing schemes, where the fingerprint features represent the indexes. Features can be classified as global or local. The first category gives macro-level details of the ridge flow, such as the fingerprint class, a pattern of ridges, and valleys on the surface of a finger. Local features can be minutia points, which are discriminative enough for the recognition task, and are composed of local ridge discontinuities. Minutiae can be of two types: terminations (ridge endings) and bifurcations.

The minutiae-based fingerprint representation method has been proposed by ANSI-NIST and includes the minutiae location and orientation (direction of the underlying ridge at the minutia location). The state-of-the-art fingerprint representation technique used to code minutiae is the Minutiae Cylinder-Code (`MCC`) (CAPPELLI *et al.*, 2010), which represents each minutia and the ones around it into a single cylinder. The MCC is invariant for translation and rotation and robust to skin distortion and small feature extraction errors. These characteristics make `MCC` a good representation model for a minutiae-based fingerprint. Besides, comparing `MCC` features (cylinders) is very efficient since they can be represented as a bit vector and compared using XOR operations.

Even though comparing two `MCC` templates is very efficient due to its bit-based representation, the comparison of large data sets is not. Given that two templates that were obtained from the same finger could have considerable variability due to numerous reasons, such as rotation, pressure, noise etc., the problem at hand becomes finding the most similar template to the queried one in a broad set in the shortest time possible. A comparable situation is found in the digital forensics field, where approximate matching techniques are used efficiently to find similar data (BREITINGER *et al.*, 2014b).

As a second application proposal for AM, we explore its use in the fingerprint identification problem. Our contribution is towards an efficient way of finding similar fingerprints templates over large databases. To this end, we present `MCC-HBFT`, a new fingerprint identification strategy based on the combination of the `HBFT` approximate matching technique and the state-of-the-art `MCC` representation model. Our results indicate that the merge of AM and fingerprint techniques can be beneficial to the problem at hand. Furthermore, we show that `MCC-HBFT` outperforms a commonly used indexing scheme on some public databases in the field.

7.3.2 The Minutia Cylinder-Code (MCC) representation model

The state-of-the-art fingerprint representation model is the `MCC` technique. According to Cappelli, R. et al. (CAPPELLI *et al.*, 2010), `MCC` encodes each minutia extracted from a fingerprint into a cylinder structure, corresponding to the spatial (cylinder base) and directional (cylinder height) relationships between a given minutia and the ones around it (ISO/IEC-19794-2:2005, 2005). Each minutia m is a triplet $\{m = x_m, y_m, \theta_m\}$, where x_m and y_m are the minutia location and θ_m the minutia direction (in the range [0, 2π]).

`MCC` cylinders can be divided into sections, corresponding to a directional difference in the range $[-\pi, \pi]$; sections are discretized into cells, and each cell receives a value related to the accumulating contributions from the minutiae around it, which depends on both spatial and directional information (CAPPELLI *et al.*, 2010).

One of the most interesting characteristics of `MCC` is its bit-based representation with a fixed length. With a negligible loss of accuracy, the value of each cell can be represented as a bit. This way, a cylinder with n cells becomes an n bit vector by linearizing its cells; a fingerprint template becomes a set of binary vectors. Another characteristic of `MCC` is that its cylinder structure is invariant for translation and rotation and robust against skin distortion and small feature extraction errors. These singularities make `MCC` extremely simple, reliable and fast for matching, being also suitable for indexing with AM techniques.

7.3.3 Related work

The basic concept of fingerprints and techniques for recognition, matching and identification can be found in surveys of the field (PERALTA *et al.*, 2015; SONI; GOYANI, 2018). Here, we will present only indexing schemes. There are several approaches to deal with the fingerprint identification problem when searching for a query fingerprint template against an extensive database. Among all schemes, we will focus on the minutia-based ones due to their better accuracy. Furthermore, we focus on `MCC` because of its bit-based representation since it is less computationally expensive and consumes less memory compared to other methods. We highlight that it is not in the scope of this chapter to present a detailed analysis of fingerprint indexing approaches. For this matter, a review is presented by Parmar, P. A. and Degadwala, S. D. (PARMAR; DEGADWALA, 2015).

The `MCC-LSH` (CAPPELLI *et al.*, 2011) indexing approach has outperformed most indexing algorithms (minutia-based) on several public databases. It is based on the Locality Sensitive Hashing (LSH) technique and `MCC` representation model. The index structure corresponds to several hash tables, each having a particular hash function. To populate the index, each cylinder (bit-based implementation) from each fingerprint in the database is hashed by several hash functions and stored in their corresponding hash tables. When searching for a fingerprint, the process repeats, but instead of saving the cylinders in the table, we check for collisions with other cylinders. The more collisions we get, the more similar the templates are.

Other methods in the literature are variations of Cappelli's approach (CAPPELLI *et al.*, 2011). Wang, Y. et al. (WANG *et al.*, 2014b) use MCC and geometric hashing (Geo-MCC). Their work is extended in (WANG *et al.*, 2015), where more compact binary hash codes are learned from MCC binary representations and used again with geometric hashing, but now combined with LSH (Geo-LSH). The authors reduced the cylinder size from 384 bits to only 24 bits, decreasing the number of hash functions and hash tables used by the system to store fingerprint templates. The benefit of their scheme comes with a reduction in accuracy, but it is still better than MCC-LSH according to their experiments. A similar approach is presented by Bai, C. et al. (BAI *et al.*, 2018), where a learning-based algorithm is used to create shorter codes from MCC. The authors use these codes to create substrings and store them into multiple hash tables.

Finally, Su, Y. et al. (SU *et al.*, 2016) presented another indexing scheme to speed up the search. They combine an improved LSH technique (using MCC) with a learning-based fingerprint pose estimation algorithm to register fingerprints into a unified finger coordinate system and avoid unnecessary comparisons.

7.3.4 The MCC-HBFT fingerprint identification strategy

The MCC-HBFT proposed here is a fingerprint identification strategy that leverages the benefits of MCC fingerprint representation and the efficiency of the AM field in finding similar fingerprint templates. We adapted the similarity digest search strategy HBFT (BREITINGER *et al.*, 2014c; LILLIS *et al.*, 2017) to operate with MCC fingerprint templates. In the next subsections, we will describe how our strategy works and the necessary steps to insert and query fingerprint templates efficiently. The algorithms describing our approach, as well as extra material with all tests performed, can be found in our GitHub page: <https://github.com/regras/mcc-hbft>. More details about the working process of HBFT can be found in sec. 2.5.4.2.

7.3.4.1 The proposed MCC-HBFT scheme

MCC-HBFT has the same working principle as HBFT, except by some modifications and additional resources necessary to operate with the MCC representation. Here, we explain the singularities of our approach.

7.3.4.1.1 Features and multiple trees

Just like HBFT, instead of inserting the template itself into the data structure, we insert small pieces of it (*features*). In an MCC template, there is one cylinder (fixed-size bit vector) representing each minutia. One possible candidate to *feature* is the cylinder itself, but

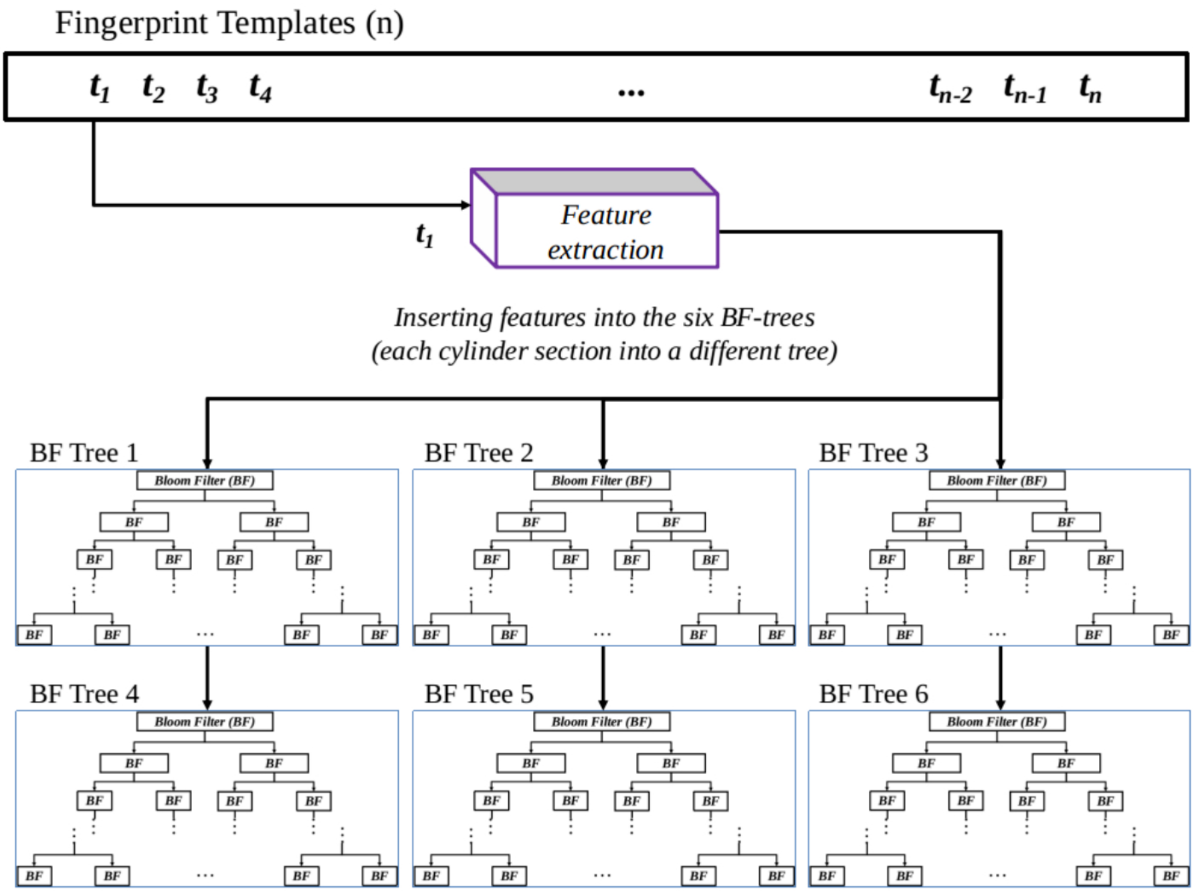

Fingerprint Templates (n)
t1 t2 t3 t4 ... tn-2 tn-1 tn
t1
Feature extraction
Inserting features into the six BF-trees
(each cylinder section into a different tree)
BF Tree 1
BF Tree 2
BF Tree 3
BF Tree 4
BF Tree 5
BF Tree 6
Bloom Filter (BF)
BF

Figure 56 – Inserting fingerprint templates into the MCC-HBFT strategy for MCC representations with six-cylinder sections.

for a more accurate version, we choose to work in the cylinder section level. Since we have s sections per cylinder, we will have s features for each cylinder.

The choice of working in the section level from the cylinder perspective has some advantages. We can reduce the *feature* size (s times) and the number of hash functions used to insert the *features* into the bloom filter structure (for the same accuracy). The downside is that the number of *features* is multiplied by s, which will demand an increase of the bloom filter size. Since bloom filters are space-efficient structures and MCC are compact representations, the increment is smoothed. Another possible issue is how to distinguish a *feature* from one section to another to avoid misleading collisions. The solution adopted in this work for this problem is based on the same paradigm used by the HBFT approach: divide and conquer. We choose to use different HBFT data structures to store the features of each section separately, i.e., one bloom filter tree for each cylinder section, as shown in Fig. 56.

To get the number of features (z) for each bloom filter tree structure, we use Eq. 7.2:

$$z = n \cdot c, \tag{7.2}$$

where n is the number of templates and c the average number of cylinders in a template.

7.3.4.1.2 Hash functions

One efficient way to create different hash functions for setting bits on bloom filters is to hash the given element with a cryptographic hash function (e.g., MD5, SHA-1 etc.) and split the result into k parts, where each piece corresponds to a different hash. However, in biometric systems, we have considerable variability in different fingerprint templates even though they belong to the same finger. Numerous factors can contribute to this fact, including displacement (same finger placed at different locations on the acquisition sensor), rotation, pressure, skin condition, noise etc. A common practice is to establish a threshold of the acceptable difference between two templates to consider them a match. This way, our problem here is to find *“similar"* templates, not precisely identical ones.

Due to the characteristics of the problem at hand, using cryptographic hash functions on fingerprint *features* will not produce good results since they can not stand minor changes in the input. Changing a single bit in data will create an entirely different output (avalanche effect). For this reason, we need a different kind of function. Here, we will use a new version of the functions presented by Cappelli, R. et al. (CAPPELLI *et al.*, 2011). Given a fingerprint feature (cylinder section), we will use k_1 functions where each of them will randomly choose b bits from each feature; next, a cryptographic hash function is used to create the hash value from the selected bits. In the end, each feature will have k_1 hash values. Since we expect that some bits may differ from one template to another due to biometric variations, we only require that k_2 hash functions ($k_2 < k_1$) match against the bloom filter to consider the feature as similar. Using many hashes per feature and establishing a minimum number of functions (k_2) to have a match, allow our approach to detect similar fingerprint templates.

7.3.4.1.3 Bloom filters sizes and number of bits in the hash functions

The bloom filter size depends on three factors: the number of elements inserted in the filter, false positive rates, and the number of hash functions. Since our structure is arranged in a tree fashion, we can create fixed-size or variable-size filters. Considering we want to keep the same number of hash functions and false positive rates for all filters, the latter option is more space-efficient because at each level, we have half the elements of the previous one inserted in the filter, which decreases the bloom filter size. The root filter (level 1) is the largest one, containing all elements of the data set. Its children and all other filters from other levels have half of their parent size. Given that we established a maximum and a minimum number of hash functions (sec. 7.3.4.1.2), we had to change the root bloom filter size formula slightly (BREITINGER *et al.*, 2014a). The adapted formula is shown in Eq. 7.3.

$$m_{root} = \frac{-k_1 \cdot z}{ln(1 - fp^{1/k_2})} \; (bits) \tag{7.3}$$

where m_{root} is the root bloom filter size, z the number of features, k_1 and k_2 are the maximum and a minimum number of hash functions, respectively, and fp the false positive rate.

Given the size of the root bloom filter (largest filter in the entire structure), we can compute the number of bits (b) necessary for the hash functions to address the bits into bloom filters using Eq. 7.4. For efficiency purposes, we compute the feature hashes once and discard one bit per level when working with filters from other levels, since each bloom filter has half of its parent size.

$$b = \lceil log_2(m_{root}) \rceil \tag{7.4}$$

7.3.4.1.4 Final score

When searching for a template T_i, we count for each feature the number of hashes that matched against the queried filter and, in case we have at least k_2 matches, that feature is said to be part of the filter; otherwise, we drop it. Then, we move on and search for the next feature in the same filter. We stop the search in this particular bloom filter when a predefined number ($hits_{min}$) of features is found, or we are out of features. In the first case, we assume a similar template lies in this bloom filter, and we can continue searching in the next levels of the tree. In the second case, we understand there is no similar template in this filter nor the subsequent levels of this node.

Once we have reached the last level of the tree (single fingerprint template per filter), we count the number of features found in that filter. Additionally, we determine a match score for the queried template and the one belonging to the bloom filter. Eq. 7.5 shows our score formula, which seeks to normalize the number of matching features by the average number of features found in the queried template T_i ($|T_i|$) and bloom filter template BF_j ($|BF_j|$).

$$score(T_i, BF_j) = \frac{H_M}{(k_1 \cdot (|T_i| + |BF_j|))/2}. \tag{7.5}$$

It is important to highlight that H_M corresponds to the number of matching hashes of all features from T_i that had at least k_2 hashes values matched.

7.3.4.1.5 Additional resources

Additional resources were integrated into `MCC-HBFT` to improve its efficiency:

Fingerprint classes: To reduce the number of template comparisons, we added a new component to our approach: fingerprint classes. Each bloom filter has a flag indicating the classes of fingerprints that lies on it. We adopted the six-class model used by NIST: Arch (A), Tented Arch (T), Left Loop (L), Right Loop (R), Scar (S), and Whorl (W) (KO, 2007). One can use the NIST PCASYS (Fingerprint Pattern Classification) (KO, 2007) system to predict the class of a fingerprint or any other method, including establishing it manually.

The classes help to decrease the number of unnecessary comparisons. When creating the bloom filter tree structure, we group the fingerprints by classes and insert them in the same or near filters. In the search process, when the queried fingerprint template belongs to a different class from the ones of a particular filter, we stop the search in that filter and all subsequent levels. Since some fingerprints may have a pattern that classifies it in more than one class, we allow an assignment of at most two classes per fingerprint.

Compatible function: Like other indexing approaches (CAPPELLI *et al.*, 2011; BAI *et al.*, 2018), a compatible function is used to narrow down the search. Two minutiae m_1 $(x_{m1}, y_{m1}, \theta_{m1})$ and m_2 $(x_{m2}, y_{m2}, \theta_{m2})$, are only considered a match if their angular difference $d_\theta(\theta_{m1}, \theta_{m2}) < \sigma_\theta$ and euclidean distance $d_{xy}((x_{m1}, y_{m1}); (x_{m2}, y_{m2})) < \sigma_{xy}$. According to Cappelli, R. et al. (CAPPELLI *et al.*, 2011), this is done to ensure a minimum rotation and displacement between them. In the proposed strategy, the minutiae attributes (x_m, y_m, θ_m) are stored into a hash table along with a minutia identification. Each fingerprint template has an exclusive table for keeping its attributes.

As mentioned before, the problem handled here is to deal with similarity cases, where fingerprint templates are not identical. For this reason, we use k_1 and k_2 as a maximum and a minimum number of hash functions, respectively, to set and query bits in our bloom filter tree structure. This allows us to have different bits between the database template and the queried one. The same problem applies to create indexes for storing the minutiae attributes (represented by k_1 hash values) into the hash table. Since each feature has several hash values and we allow the match of only part of them, we can not create a single index for a feature to insert it into the table. If we use all hashes to derive an index value, any similar feature having at least one different bit will probably have a different index and will not be correlated to their similar ones.

The solution proposed here follows the idea adopted in sec. 7.3.4.1.2. We use all k_1 hash values from a feature to derive many indexes and, for each one, we insert the feature in the corresponding hash table bucket (we only store the feature identification, while its attributes (x_m, y_m, θ_m) are stored elsewhere to avoid redundancy and save memory). Upon a query request, we check the hash table for the features that collide at least k_2 times with the queried one; in a positive case, we perform the compatible function between the features. Only if these two conditions are true, we count a match for the queried feature.

7.3.4.2 Creating the MCC-HBFT data structure: The preparation phase

The preparation phase, often called the offline stage, consists of creating the `MCC-HBFT` data structure and inserting all database fingerprint templates on it. First, we create s (cylinder sections) empty bloom filter trees and a set of hash functions for each tree according to the cylinder section it lies on. The next step is grouping the fingerprint templates according to their classes and insert them into `MCC-HBFT`. Since we could have more than one class per fingerprint, we classify the fingerprints prioritizing the first class.

The insertion process goes as follow: The first template of the first group is inserted into BF_x (where $x = 2^{L-1}$ is the number of the bloom filter BF in the tree at level L), the first filter

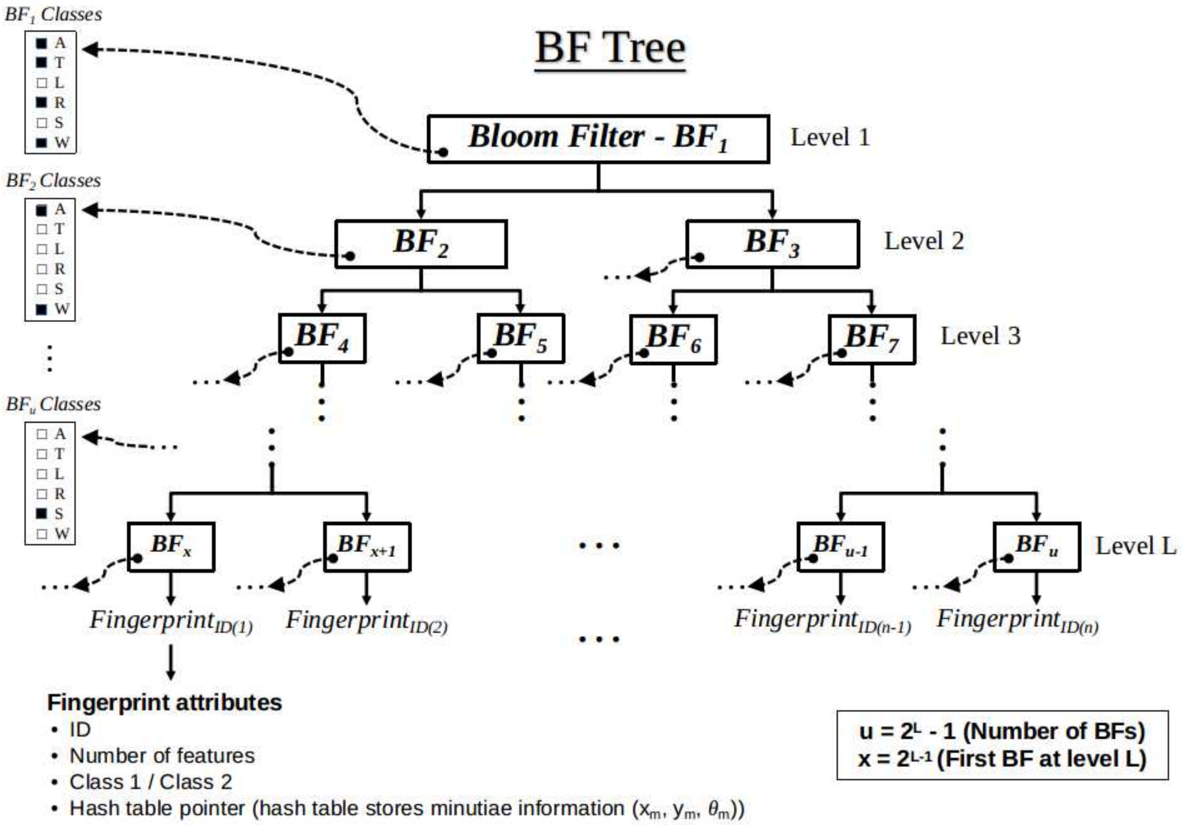


Figure 57 – MCC-HBFT: Single bloom filter tree structure

of the last level of the tree, and all its parents' nodes. In BF_x, we insert the template features, some attributes, and create a new hash table for keeping the minutiae information. Besides, we set the class tag of the current filter and all its parents according to the given template classes. Then, we proceed to the next template and insert it into BF_{x+1}, the second filter of the last level, and all its parents. Again, we set the classes of the bloom filters accordingly and proceed until we have inserted all templates. Fig. 57 shows the MCC-HBFT structure of a single bloom filter tree.

We highlight that the lowest level of each tree stores only one template per filter (which has several cylinders, thus having several features). Furthermore, we keep some fingerprint attributes for later identification and comparison purposes, such as an identification (ID), fingerprint classes (classes 1 and 2), and the number of features. A hash table for each template is also stored to keep the minutiae information.

7.3.4.3 Searching fingerprint templates: The operational phase

The operational phase, also known as the online stage, follows the preparation and consists of performing searches on the MCC-HBFT structure. Given the MCC template and its class, the proposed strategy performs the feature extraction process and looks for each feature according to their cylinder section in the bloom filter trees. A match is found when at least $hits_{min}$ features are located in a filter. In the case of a non-match in the root filter, we discard that feature and move to the next one. When matching, we go further and look for matches in the bloom filter children of that node. We proceed with the search until we reach a filter on the last level of the tree, or we have a non-match in the current node and all other ones.

In the last level of the tree, we have one template per filter. Besides looking for all features, we also compute a match score for the queried template (sec. 7.3.4.1.4), and after searching into all *s* trees, we summed up the results and present an ordered list (by match score) with all possible candidates to the similarity to the queried template.

7.3.5 Assessment

7.3.5.1 Evaluation setup, databases, and parameters

The tests described here used a machine running a dual boot of Elementary OS 0.4.1 Loki 64-bit (built on Ubuntu 16.04.2 LTS) and Microsoft Windows 10 64-bit, with an i7-5500U CPU @2.40 GHz processor, 8 GB of memory, 1 TB SATA 3Gb/s hard disk drive (5,400 rpm), and NVIDIA GeForce 920M. The proof of concept `MCC-HBFT` was developed using C language.

The performance of `MCC-HBFT` was measured using public domain fingerprint databases, such as the NIST Special Databases 4 (NIST, 2018) and some FVC databases ((FVC2002, 2018) and (FVC2004, 2018)). All database details are shown in Table 44.

Table 44 – Detailed information on public databases used in the experiments

Databases	**Size**	**Resolution**	**Subject**	**Impressions**	**Sensor**	**Format**
NIST DB4	512x512	500dpi	2000	2	Ink-rolled	PNG
FVC2002 DB1a	388x374	500dpi	100	8	Optical	TIF
FVC2002 DB3a	300x300	500dpi	100	8	Capacitive	TIF
FVC2004 DB1a	640x480	500dpi	100	8	Optical	TIF

The tests presented here followed the strategy used in the literature (BAI *et al.*, 2018; CAPPELLI *et al.*, 2011). For the NIST database, the first fingerprint impression of DB4 is used for indexing and the second one for querying. On the FVC database, the first impression was used for index and remaining seven for querying. To estimate some parameters of `MCC-HBFT`, we used 600 fingerprints from NIST DB4 (500 first impressions for index and 100 second impressions for query) and 200 fingerprints from FVC2002 DB1a (25 first impressions for index and 175 impressions for query).

The fingerprint minutiae extraction was performed in two different ways. For the NIST fingerprints, we used the open source NBIS[1] software. For all FVC sets, we used a set of manually extracted minutiae (FM3) (KAYAOGLU *et al.*, 2013). Next, the minutia information of both databases is used to create the `MCC` representation using `MCC SDK v2.0`[2], creating a

[1] NIST Biometric Image Software (NBIS) v5.0.0, http://www.nist.gov/itl/iad/ig/nbis.cfm

[2] MCC SDK v2.0, http://www.biolab.csr.unibo.it/mccsdk.html

384-bit-based template for each fingerprint. We adopted the same parameters as reported in (CAPPELLI *et al.*, 2011) for the cylinder creation.

We limited the comparison of our strategy to only the state-of-the-art index structure `MCC-LSH` (CAPPELLI *et al.*, 2011) since it has a free implementation available by `MCC SDK v2.0`. All other indexing schemes that work with `MCC` did not have their source code available for comparison at the time we were performing our experiments (from the best of our knowledge). Most approaches only compare their proposals to `MCC-LSH`, the only readily available one in the literature. The tests performed here used the same `MCC` and LSH parameters available in (CAPPELLI *et al.*, 2011). A C# routine was developed in the Windows operating system to create the index structure, perform the queries, and then consolidate the results.

`MCC-HBFT` makes use of fingerprint classes. Here, we adopted the free NIST PCASYS (KO, 2007) software to perform the class assignment for first class and a manual adjustment to insert the second class, when necessary. Since the focus of this research is based on the use of classes to reduce the number of comparisons and not in the classification process itself, we are not interested whether PCASYS assigns a right class to a fingerprint or not. We are only concerned about the fact that two mate fingerprints have the same class.

Other parameters of `MCC-HBFT` include the number of hash functions k_1 and k_2, established after experiments over the test databases (shown in the next section). We also defined $hits_{min}$=20%, which is the number of feature matches to conclude that a similar feature is inserted in a bloom filter. This value is a proportion of features found in the filter by the total number of queried features, also found experimentally. Other parameters are $s = 6$ (the same value as the one adopted by (CAPPELLI *et al.*, 2010)) and SHA-1 as a cryptographic hash function. The values used in the compatible function are: $\sigma_\theta = \pi/4$ and $\sigma_{xy} = 256$.

7.3.5.2 Results

The evaluation of the accuracy and efficiency of fingerprint indexing schemes is measured by the trade-off between Error Rate (ER) and Penetration Rate (PR). The first metric is based on the number of queried fingerprints not found in a search, while the latter one corresponds to the proportion of the database explored by the indexing approach in a query. The best scenario is the smallest possible error to the lowest penetration rate.

In our experiments, we have randomly generated the hash functions (bits selected in the cylinders) each time we ran a full trial with `MCC-HBFT`. Even though we require fixed hash functions to always produce the same results, our tests changed it because *(i)* the size of the databases is different, requiring more or fewer bits for each function; *(ii)* we wanted to verify the impact of “good" and “bad" bit selections. We also have chosen three versions of `MCC-HBFT` using different numbers of hash functions to find the best cost/benefit setting. We highlight that the more hash functions used, the higher the costs (time) is. The settings include low, mid, and high-cost versions.

Determining the values of k_1 and k_2 required the two test databases. First, we set $k_1 = x \cdot k_2$, where x is a variable controlling the level of changes accepted over two templates. The best results for x were different for FVC and NIST. In the former, the best values varied between $x = 5$ and $x = 6$, while the later had $x = 3$ and $x = 4$. These results helped us to find the different settings of our approach.

We ran each experiment 10 times and selected the worst, average, and best-case scenarios, and compared them to `MCC-LSH`. To be fair, we have compared our approach under its average-case scenario. Fig. 58 shows the results for the FVC2002-DB1a set with the average results obtained from `MCC-HBFT` using the three different settings. We can see that all `MCC-HBFT` versions had better outcomes compared with `MCC-LSH` on average. Even though `MCC-LSH` had better results for a low penetration rate ($PR \leq 7.0\%$), `MCC-HBFT` settings presented better results from this point on. Besides, our low-cost version reached $ER = 0\%$ with $PR = 39\%$, while `MCC-LSH` had $PR = 48\%$.

Fig. 59 shows the three scenarios of the low-cost setting ($k_1 = 20/k_2 = 4$) and the baseline `MCC-LSH`. The worst-case had $ER = 0.28\%$ ($PR = 100\%$). However, the average scenario had $ER = 0.0\%$ and $PR = 39\%$. We chose to show this particular setting because it had the worst performance with respect accuracy of all three and because it is a low-cost version. Adopting a more proper hash function means increasing the number of hashes, as shown in Fig. 58, where the high-cost version ($k_1 = 72/k_2 = 12$) performed better than the others ($ER = 0.0\%$ and $PR = 28\%$).

Fig. 60 shows the experiments under FVC2002-DB3 database. The low-cost version ($k_1 = 20/k_2 = 4$) performed worst, while the others competed with `MCC-LSH` but had a higher error rate. It is important to highlight that none of the approaches reached $ER = 0\%$, and `MCC-LSH` had $ER = 1\%$. On the other hand, over FVC2004-DB1 (Fig. 61), `MCC-HBFT` beats the state-of-the-art proposal significantly, except for the mid-cost version with $PR \geq 82\%$, when it ends with a superior error rate.

The tests using the NIST BD4 database have adopted different parameters for the hash functions, but we still use the idea of the three settings: low, mid, and high-cost versions. Fig. 62 shows the results of three settings and the baseline `MCC-LSH`. Even with the low-cost version ($k_1 = 18/k_2 = 6$), we had better results than `MCC-LSH` from $PR \geq 7.95\%$ and reached $ER = 0.0\%$ before it (with $PR = 41.3\%$ in comparison to $PR = 100.0\%$ of `MCC-LSH`). `MCC-HBFT` presented similar results over the NIST database in all of its executions. The worst-case scenario of all experiments always had results close to the best one, except in one case, where the search could not find all templates in the search, as illustrated in Fig. 63. In this setting, we had $ER = 0.05\%$, which corresponds to a single fingerprint not found. The other two experiments were successful in finding all candidates with $PR = 42.05\%$ (on average) for their worst-case scenario.

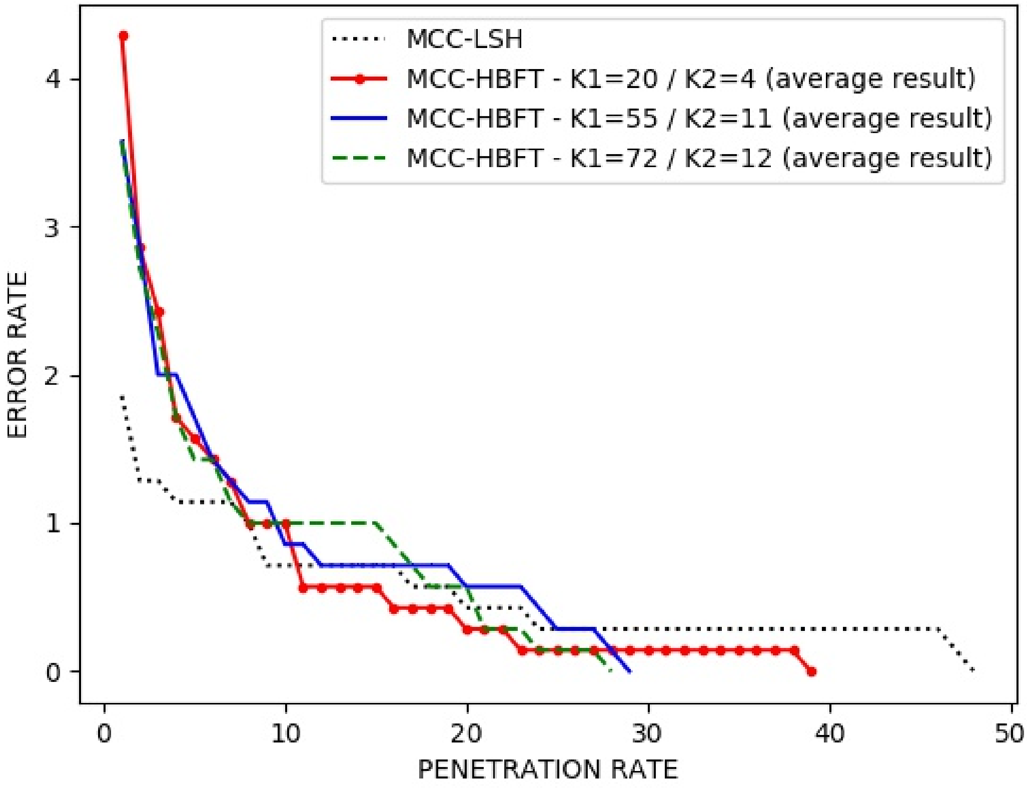


Figure 58 – Performance evaluation on FVC2002 DB1: Average-case scenario of three different MCC-HBFT versions

7.3.5.3 Discussion

`MCC-HBFT` can be used in different settings. We expected that the more hash functions we use, the higher the accuracy. However, the benefits were quite small, given high k_1 values. In general, the results for the different settings were quite similar. In opposition, the time costs increased significantly since we had to perform many more hashes per feature. The low-cost version appeared as the most cost/benefit combination. Future work encompasses a cost/benefit analysis of different MCC-HBFT settings.

To understand the hash function parameters changing from FVC to the NIST database, we analyzed the details of each set and summarized it in Table 45. The number of minutiae per template in each database varied significantly. NIST database has more minutiae per template, allowing a smaller proportion of k_1 and k_2 since it has many minutiae to compare between two templates (more chances to get matches). The FVC sets have fewer minutiae, decreasing its chances for matching. For this reason, the difference between k_1 and k_2 must be higher.

One difference to call attention between NIST and FVC databases results is the error rate of small values of database penetration. We had significantly lower errors for FVC than NIST set. We attribute that to the extraction minutia process, which was manually and carefully

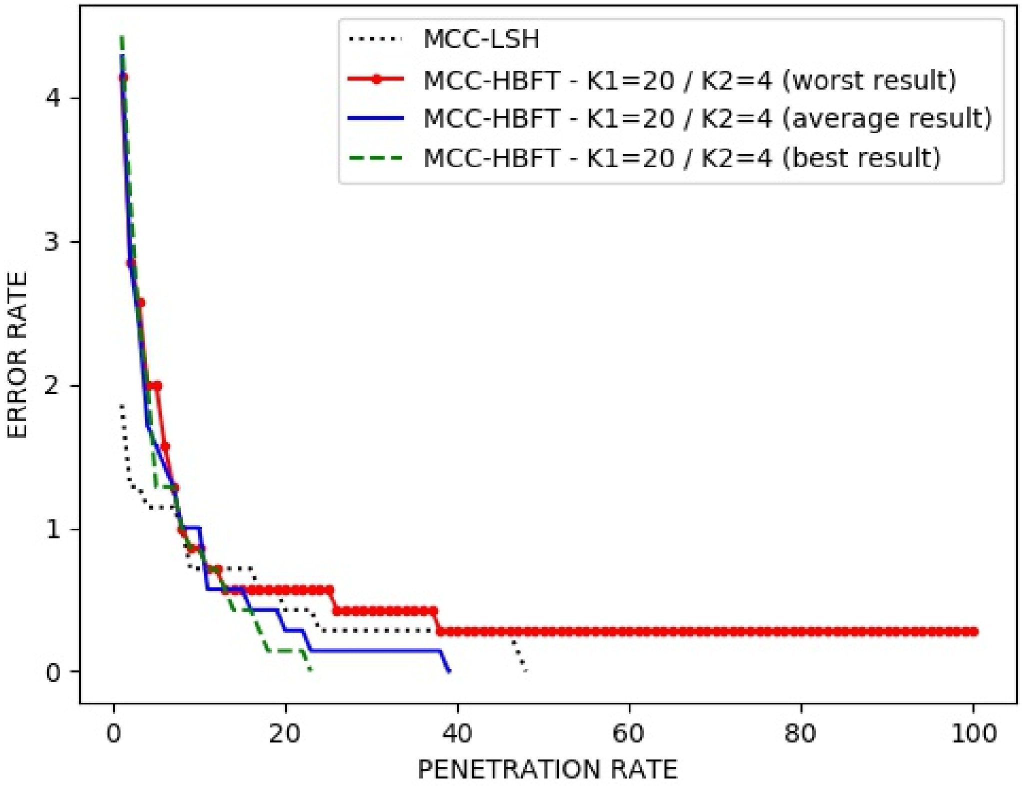


Figure 59 – Performance evaluation on FVC2002 DB1: The worst, average, and best-case scenarios (low-cost)

Table 45 – Detailed information on public databases used in the experiments

Database **Parameters**	**2002DB1**	**FVC** **2002DB3**	**2004DB1**	**NIST** **DB4**
Number of minutiae (avg) per template	35.02	21.49	38.86	120.0
Max. number of minutiae per template	81	46	77	237
Min. number of minutiae per template	5	2	8	13
Number of bit 1 in each minutia (avg)	11.0681	9.1413	10.5342	12.7070

done for the FVC database and automatic for NIST set. The quality of the minutiae possibly influenced the results, but here we are more concerned with the performance of our approach against the state-of-the-art scheme. Since we used the same minutiae for both approaches, we believe this will not affect the relative results.

7.3.6 Partial conclusions

Identifying individuals over large databases using fingerprints is a challenging problem, mainly due to efficiency reasons. The approximate matching techniques are efficient solutions in digital forensics to find similar content, and they can do the same in the fingerprint

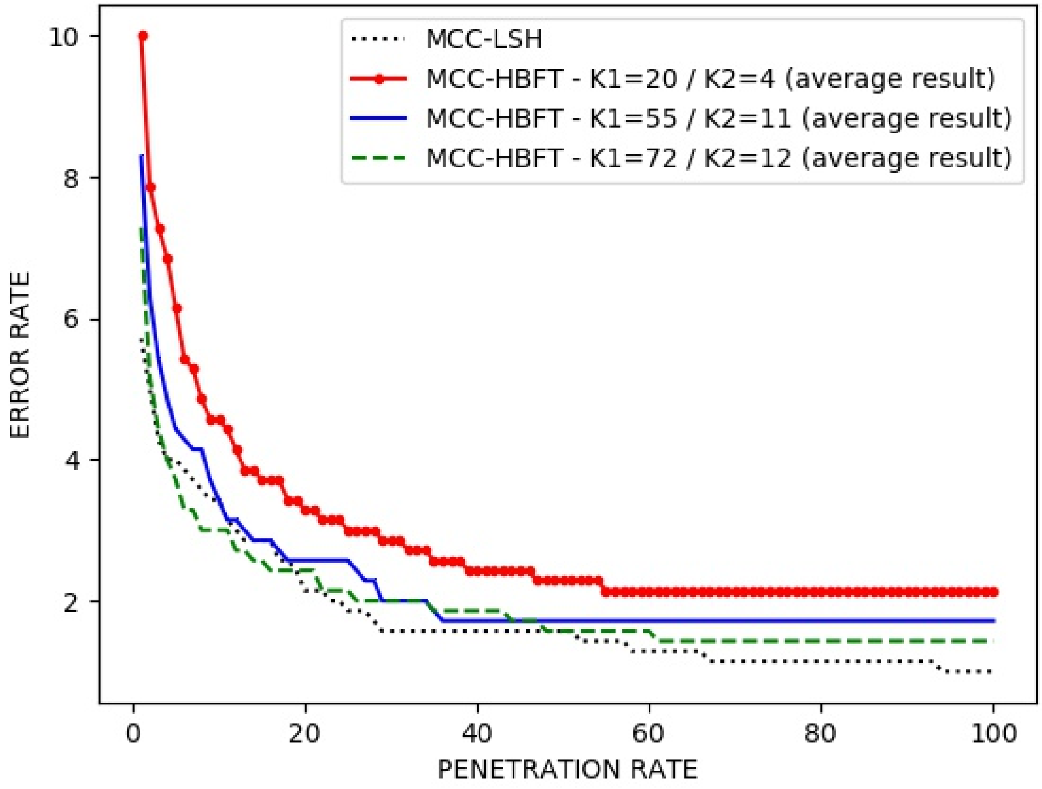


Figure 60 – Performance evaluation on FVC2002 DB3: Average-case scenario of three different MCC-HBFT versions

field. In this chapter, we have presented `MCC-HBFT`, a new fingerprint identification strategy that leverages the efficiency of the AM field and the accuracy of the state-of-the-art fingerprint representation `MCC`. We showed how our strategy works and outperforms a commonly used fingerprint indexing approach on public domain databases. Future work encompasses a cost/benefit analysis of different `MCC-HBFT` settings. We also plan to analyze the use of more efficient hash functions to decrease the overall costs without impacting accuracy.

7.4 Other applications for AM

Other than the two approaches presented in this chapter, we can mention another application for AM: File type identification. By following the concepts explained in Chapter 5, we believe we could use a database of common features to track some of the most common structures among several file types. Then, by counting the features in common between the target file (or just a piece of it) and our database, we could establish the type that file belongs to. However, future work is necessary to demonstrate the feasibility of our approach.

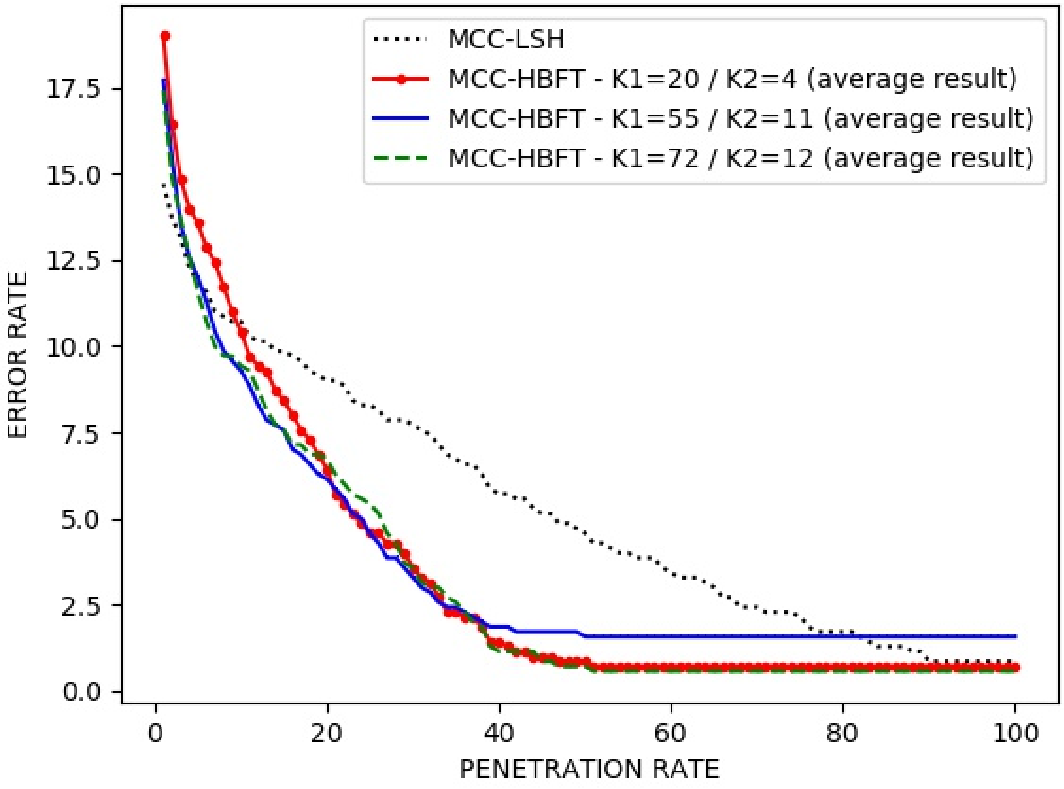


Figure 61 – Performance evaluation on FVC2004 DB1: Average-case scenario of three different MCC-HBFT versions

7.5 Conclusions

In this chapter, we presented two new applications for approximate matching: Fast file identification and searching fingerprints with AM. We discussed each approach, presented its results, and a discussion about their effectiveness and limitations. We showed that besides the high cost associated with AM tools compared to traditional hash-based approaches, the effectiveness achieved in identifying similar objects, makes AM a good choice for file identification, especially when combined with sampling as proposed in this chapter. The two techniques together achieved significant results related to the reduction in the number of comparisons and false positive matches compared to AM alone. Besides, using AM for searching for fingerprints was also a good choice, where the solution proposed in this chapter outperformed a commonly used fingerprint indexing approach. By our results, we can see that AM can also be applied for mitigating problems of other areas successfully.

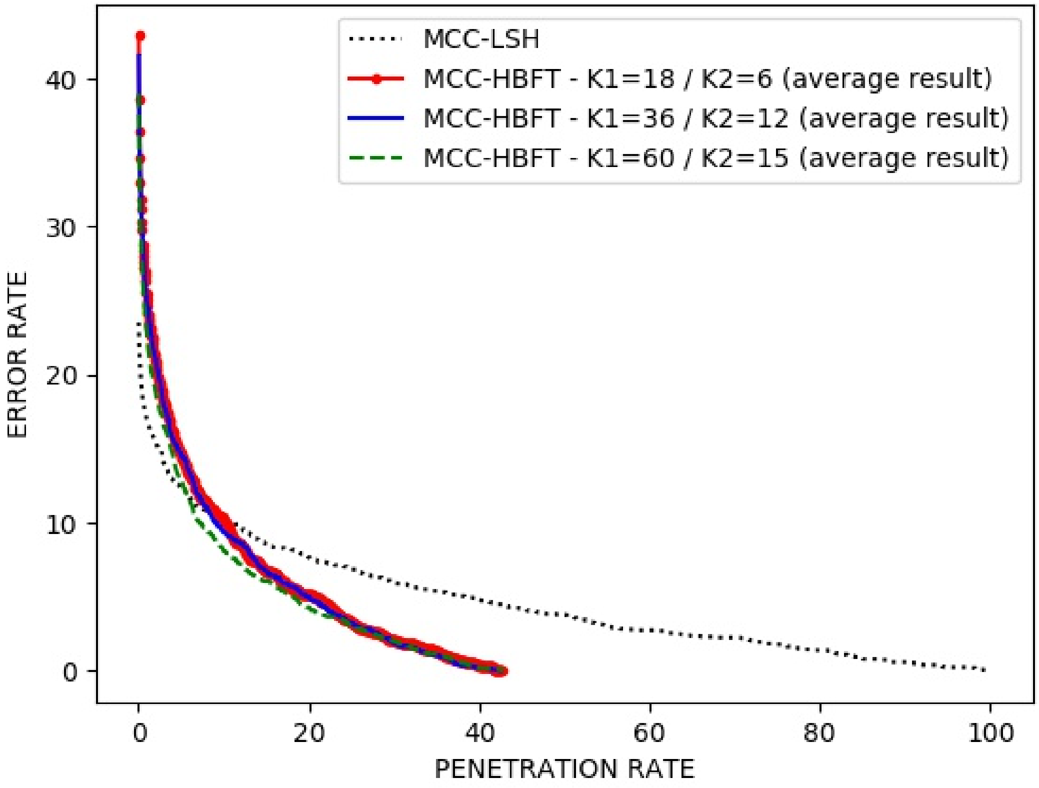


Figure 62 – Performance evaluation on NIST DB4: Average-case scenario of three different MCC-HBFT versions

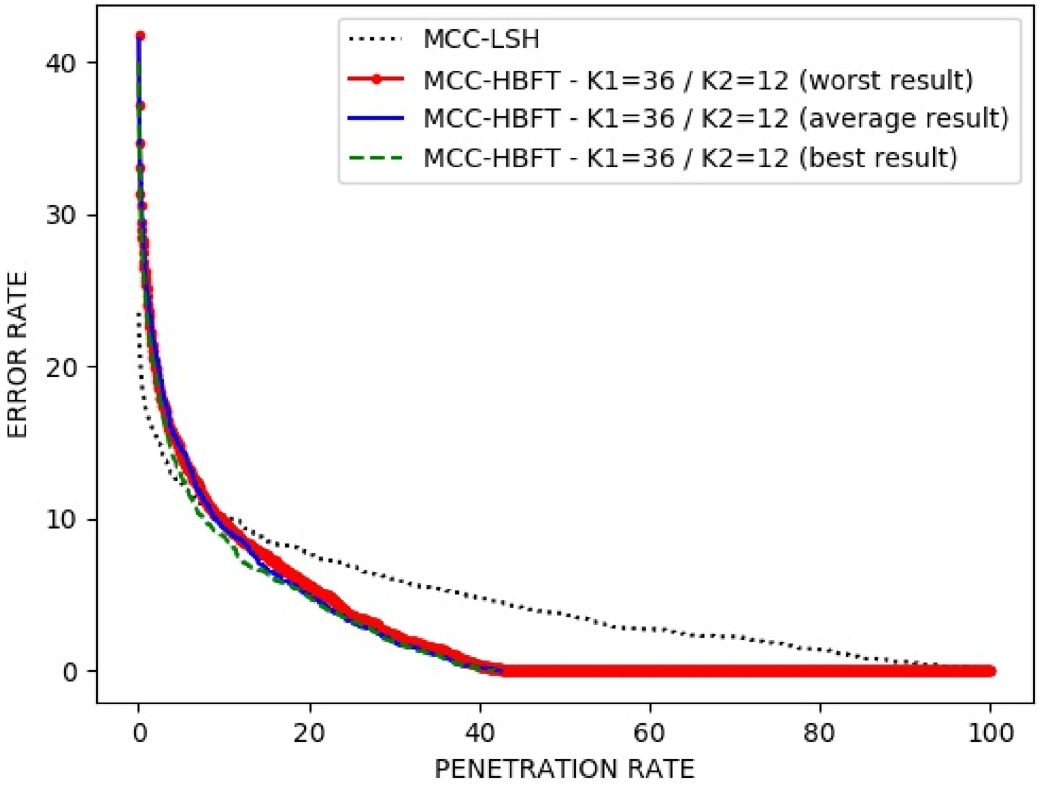


Figure 63 – Performance evaluation on NIST DB4: The worst, average, and best-case scenarios (mid-cost)

8 Conclusions

8.1 Contributions of this thesis

Digital forensic investigations suffer from the massive amount of data available nowadays, where time is a scarce resource. In this work, we presented Approximate Matching (AM) functions as a candidate to deal with such a problem when performing the similarity search. We presented the main concepts of AM and the limitations of current solutions. Our contributions to the field were focused on new solutions to mitigate some of the current limitations. In the following paragraphs, we present some of the main conclusions one can obtain from this study. A list of the publications derived from this work can be found in appendix C.

In Chap. 3, we showed that the Similarity Digest Search Strategies (SDSS) are efficient approaches for comparing large data sets of objects. Our results demonstrated that, although the strategies outperform the naive brute force method, there is no suitable approach that satisfies all the most relevant requirements of forensic investigations, such as low memory requirement, high detection capabilities (for both resemblance and containment), and efficient lookup procedure.

Another problem with SDSS is that they are specific for a particular AM tool, being `ssdeep` and `sdhash` the most predominant choices. Other tools with interesting characteristics for digital forensic investigations do not have better forms to perform the similarity search other than the brute force. For this reason, Chap. 4 presents Fast Similarity Digest Search (FSDS), a new strategy aiming to perform efficiently over large amounts of data. FSDS is based on `TLSH` approximate matching tool and shows a reduction of about 95% in time concerning brute force with a minimum impact on precision.

We also analyzed current AM tools with a focus on improving the similarity detection process and hence their effectiveness in producing valid results on investigations. By simulating real world investigations, we figured out that many matches pointed out as similar by current solutions were not similar when we visually inspected them and look for similar content, such as paragraphs, figures, tables, among other elements created by users. We found out that the matches were indeed a result of common blocks (Chap. 5). Then, we came up with a classification of the matches according to the kind of similarity detected (e.g., user-generated content, application-generated content, and template content) and removed the common data among objects from the similarity digest. Our results showed a significant reduction in the number of matches; in some cases, we obtained approximately 87% fewer matches compared to the traditional tool.This practice also benefited precision and recall rates, where different settings aided each metric differently based on the goal of the investigation. Besides, the impact on the threshold score of AM after removing common blocks was also noticed, where all scores produced relevant matches, and no threshold was necessary.

Chapter 6 improved our knowledge about one of the most popular AM tools. It provided a theoretical analysis of the detection capabilities of sdhash, showing the conditions for which it works efficiently and the ones where it struggles. In general, the larger the object, the better the performance of sdhash; it requires 0.50% of similar content between two objects (with the smallest object size) to detect similarity in the best case. For small objects, on the other hand, it requires about 35% of similar data to produce valid scores. By identifying sdhash's limitation (inconsistency when dealing with small objects), we proposed changing its comparison function. We developed an improved version of it where we replaced the comparison function for a new one using Jaccard Similarity. Our results showed that the new version generated an easier to interpret score (that can be adjusted to work in different scenarios) and produces results more reliable and closer to the real similarity found in objects.

Our last contribution was towards new applications for AM functions. Chap. 7 showed how to use AM combined with sampling techniques for fast file identification.Besides, it also showed how to apply AM for fingerprint identification where individuals can be identified over large data sets.

8.2 Research Questions

All of the contributions presented so far led us to answer satisfactorily our research questions, presented in Chapter 1.

Leading RQ. How can we perform a similarity search (from a digital forensic practitioner perspective) over large data sets in a (time/space) efficient manner?

To this end, more specific research questions were elaborated, covering many topics over the AM field. Here, we will provide a brief answer to the proposed questions, according to the results presented in previous chapters.

RQ1. Can AM functions deal with huge data sets efficiently? How would digital forensic investigations benefit from the use of such functions?

Our answer to the first part is yes. AM functions can be used to perform similarity searches in huge data sets using Similarity Digest Search Strategies (SDSS). In Chapter 3, we presented the SDSS and provided a detailed comparison among these strategies, showing the characteristics and how they would scale for data set size increase. We also showed how some SDSS perform in real investigations. Finally, we proposed in Chapter 4, a new strategy to perform queries efficiently using the TLSH approximate matching tool.

RQ.2 How can we estimate the (theoretical) minimum similarity detected by AM functions?

Most AM tool proposals show their benefits by empirical tests over random and real-world data sets. We believe that such analysis, although extremely important, is not enough for making AM a reliable choice in real-life investigations. A more formal study showing the boundaries of the proposed function is necessary. In this work, we sought to demonstrate theoretically how `sdhash` would behave for detecting similarity. We meant to answer questions like: *What is the minimal similarity between two objects detected by `sdhash`? In which cases/scenarios will it work? What is the impact of the α variable (on `sdhash` score computation) and the chosen default value on similarity assessment?*

We answered these and other questions in Chapter 6 in order to provide a detailed analysis of the `sdhash` similarity assessment process. In short, `sdhash` tends to produce score values different from the real similarity found in small objects and requires many similar bytes to produce valid scores (about 35% of the same content). For larger objects, on the other hand, it requires less similar bytes (only 0.50% to produce a score > 0). We believe that such understanding is important to forensic practitioners when performing investigations so they can maximize their results according to the tools' capabilities and investigation goals.

RQ3. How can we improve current AM tools to perform better over large data sets and produce more reliable results?

We proposed solutions to improve the similarity assessment process, by removing common blocks from the similarity digest (Chap. 5) and introducing a new similarity comparison function for one of the most well-known and used AM tools: `sdhash` (Chap. 6). In the first case, we removed common structures that repeated over many different objects. The goal was to avoid this information to interfere with the similarity digest of objects and influence the similarity assessment process. By removing these blocks, we reduced significantly the number of matches returned by AM tools, such as `sdhash`; consequently, the precision improved. This was achieved with a small cost on the tool's performance.

The second contribution was based on the improvement of the `sdhash` comparison function. By changing the way `sdhash` computes the similarity score, we could get more reliable results that express the real similarity shared by two objects. Besides, our new function (using Jaccard similarity) can also be more easily interpreted regarding the score produced and the real similarity of objects, since it behaves like a percentage score. This way, forensic practitioners can adjust the tool more precisely according to their needs and tool capabilities.

RQ.4 Is there any other application for AM functions?

Chap. 7 presented two new applications for AM: Fast file identification (using sampling techniques) and fingerprint identification. Considering the quality of our results in both applications, we can conclude that AM can be used in other scenarios successfully. Besides, other applications for AM can also be proposed, such as the file type identification, a possible topic for future work.

By answering all these questions, we addressed our leading research question by providing enough details on how AM can and should be used in digital forensic investigations; besides, we showed how to use it when dealing with huge amounts of data.

8.3 Limitations

This thesis has the following limitations:

It does not deal with encrypted data. All approximate matching functions presented here are unable to detect the similarity of encrypted data. Investigations with encrypted seized devices require the decryption of the data before the application of the methods discussed in this thesis.

Plagiarism. Another application of AM could be for plagiarism identification. However, one should be aware that only the copy and paste approach can be detected for AM approaches discussed in this thesis that operate in the bytewise level. The similarity of content in the semantic level is a limitation of AM. An example of such a case is changing a word for its synonym (e.g., substitute *method* for *technique*) or changing the name of variables in source code.

Image files identification. AM does not work equally well for all file types, such as images. Any change in the illumination, scale, or other simple operation performed in images, may change the whole byte structure of the object, making the similarity detection harder. Algorithms that work at the semantic level should be the ones used for better results, although they require more resources to operate.

Manual testing on experiments. For not having a proper data set to evaluate AM effectively concerning recall and precision rates as our thesis did, we had to perform a lot of manual testing and random sampling in existing databases. Our goal was to find out the ground truth, and to this end, we selected a subset of the *t5-corpus* to perform our analysis. We argue that it is possible that we misclassified some of the matches or that we ended up with poor samples drawn from the data set. We provide in appendix B a list with our results so one can validate and complement it.

8.4 Future works

Along the way of working towards solutions to current limitations of the field, we also identified opportunities for future work. Next, we summarize the main open issues on the AM field discussed along with this work.

First, measuring precision and recall rates of the SDSS with/without the influence of common blocks will show how current approaches would perform in investigations regarding their detection capabilities of similar content. Besides, including an analysis of resemblance/ containment of the strategies will be desirable.

Another future work is to study the possibility of applying machine learning algorithms to the identification of common blocks. Given enough data, we think it is possible to train an algorithm for detecting and separating the kind of similarity we want (user-generated content, application-generated content, or template content).

Last, future work is necessary to find out the viability of applying common blocks concept and AM to the file type identification problem. Given the byte sequence (e.g., disk sector, object fragment), we will be able to use such techniques to discover the type of the object by comparing the provided sequence to known blocks of certain file types.

Bibliography

ADOBE-SYSTEMS. *ISO 32000 - Document management - Portable Document 493 Format. Part 1: PDF 1.7*. First. [S.l.], 2008.

AGRAWAL, P.; ARASU, A.; KAUSHIK, R. On indexing error-tolerant set containment. In: ACM. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. [S.l.], 2010. p. 927–938.

ALVAREZ, P. Using extended file information (exif) file headers in digital evidence analysis. *International Journal of Digital Evidence*, v. 2, n. 3, p. 1–5, 2004.

ANDONI, A. *Nearest neighbor search: the old, the new, and the impossible*. 2009. <https://dspace.mit.edu/handle/1721.1/55090>. Doctoral dissertation, Massachusetts Institute of Technology. Accessed 2019 Oct 21.

AUTOPSY. *Documentation*. 2015. <http://wiki.sleuthkit.org/index.php?title=Autopsy>. Accessed 2018 Jan 12.

AUTOPSY. *Hash Database Lookup Module*. 2016. <http://sleuthkit.org/autopsy/docs/user-docs/4.1/hash_db_page.html>. Accessed 2018 Jan 12.

BAI, C.; WANG, W.; ZHAO, T.; LI, M. Fast exact fingerprint indexing based on compact binary minutia cylinder codes. *Neurocomputing*, Elsevier, v. 275, p. 1711–1724, 2018.

BAIER, H.; BREITINGER, F. Security aspects of piecewise hashing in computer forensics. In: IEEE. *IT Security Incident Management and IT Forensics (IMF), 2011 Sixth International Conference on*. [S.l.], 2011. p. 21–36.

BJELLAND, P. C.; FRANKE, K.; ÅRNES, A. Practical use of approximate hash based matching in digital investigations. *Digital Investigation*, Elsevier, v. 11, p. S18–S26, 2014.

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, ACM, New York, NY, USA, v. 13, n. 7, p. 422–426, jul. 1970. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/362686.362692>.

BREITINGER, F.; ASTEBØL, K. P.; BAIER, H.; BUSCH, C. mvhash-b-a new approach for similarity preserving hashing. In: IEEE. *IT security incident management and it forensics (IMF), 2013 seventh international conference on*. [S.l.], 2013. p. 33–44.

BREITINGER, F.; BAIER, H. A fuzzy hashing approach based on random sequences and hamming distance. In: ASSOCIATION OF DIGITAL FORENSICS, SECURITY AND LAW. *Proceedings of the conference on digital forensics, security and law*. [S.l.], 2012. p. 89.

BREITINGER, F.; BAIER, H. Performance issues about context-triggered piecewise hashing. In: *Digital Forensics and Cyber Crime: Third International ICST Conference, ICDF2C 2011, Dublin, Ireland, October 26-28, 2011, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 141–155. ISBN 978-3-642-35515-8. Disponível em: <http://dx.doi.org/10.1007/978-3-642-35515-8_12>.

BREITINGER, F.; BAIER, H. Properties of a similarity preserving hash function and their realization in sdhash. In: *2012 Information Security for South Africa*. [S.l.: s.n.], 2012. p. 1–8. ISSN 2330-9881.

BREITINGER, F.; BAIER, H. 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*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 167–182. ISBN 978-3-642-39891-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-39891-9_11>.

BREITINGER, F.; BAIER, H.; BECKINGHAM, J. Security and implementation analysis of the similarity digest sdhash. In: *First international baltic conference on network security & forensics (nesefo)*. [S.l.: s.n.], 2012.

BREITINGER, F.; BAIER, H.; WHITE, D. On the database lookup problem of approximate matching. *Digital Investigation*, Elsevier, v. 11, p. S1–S9, 2014.

BREITINGER, F.; GUTTMAN, B.; MCCARRIN, M.; ROUSSEV, V.; WHITE, D. Approximate matching: definition and terminology. *NIST Special Publication*, v. 800, p. 168, 2014.

BREITINGER, F.; RATHGEB, C.; BAIER, H. An efficient similarity digests database lookup-a logarithmic divide & conquer approach. *The Journal of Digital Forensics, Security and Law: JDFSL*, Association of Digital Forensics, Security and Law, v. 9, n. 2, p. 155, 2014.

BREITINGER, F.; ROUSSEV, V. Automated evaluation of approximate matching algorithms on real data. *Digital Investigation*, Elsevier, v. 11, p. S10–S17, 2014.

BREITINGER, F.; STIVAKTAKIS, G.; BAIER, H. Frash: A framework to test algorithms of similarity hashing. *Digital Investigation*, Elsevier, v. 10, p. S50–S58, 2013.

BREITINGER, F.; STIVAKTAKIS, G.; ROUSSEV, V. Evaluating detection error trade-offs for bytewise approximate matching algorithms. *Digital Investigation*, Elsevier, v. 11, n. 2, p. 81–89, 2014.

BREITINGER, F.; WINTER, C.; YANNIKOS, Y.; FINK, T.; SEEFRIED, M. Using approximate matching to reduce the volume of digital data. In: *Advances in Digital Forensics X: 10th IFIP WG 11.9 International Conference, Vienna, Austria, January 8-10, 2014, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 149–163. ISBN 978-3-662-44952-3. Disponível em: <http://dx.doi.org/10.1007/978-3-662-44952-3_11>.

BREITINGER, F.; ZIROFF, G.; LANGE, S.; BAIER, H. Similarity hashing based on levenshtein distances. In: *Advances in Digital Forensics X: 10th IFIP WG 11.9 International Conference, Vienna, Austria, January 8-10, 2014, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 133–147. ISBN 978-3-662-44952-3. Disponível em: <http://dx.doi.org/10.1007/978-3-662-44952-3_10>.

BRODER, A. Z. On the resemblance and containment of documents. In: IEEE. *Compression and Complexity of Sequences 1997. Proceedings*. [S.l.], 1997. p. 21–29.

CAPPELLI, R.; FERRARA, M.; MALTONI, D. Minutia cylinder-code: A new representation and matching technique for fingerprint recognition. *IEEE TPAMI*, IEEE, v. 32, n. 12, p. 2128–2141, 2010.

CAPPELLI, R.; FERRARA, M.; MALTONI, D. Fingerprint indexing based on minutia cylinder-code. *IEEE TPAMI*, IEEE, v. 33, n. 5, p. 1051–1057, 2011.

CHANG, D.; SANADHYA, S. K.; SINGH, M.; VERMA, R. A collision attack on sdhash similarity hashing. In: *Proceedings of 10th intl. conference on systematic approaches to digital forensic engineering*. [S.l.: s.n.], 2015. p. 36–46.

CHAWATHE, S. S. Effective whitelisting for filesystem forensics. In: *2009 IEEE International Conference on Intelligence and Security Informatics*. [S.l.: s.n.], 2009. p. 131–136.

CHAWATHE, S. S. Fast fingerprinting for file-system forensics. In: *2012 IEEE Conference on Technologies for Homeland Security (HST)*. [S.l.: s.n.], 2012. p. 591–596.

CHEN, L.; WANG, G. An efficient piecewise hashing method for computer forensics. In: *First International Workshop on Knowledge Discovery and Data Mining (WKDD 2008)*. [S.l.: s.n.], 2008. p. 635–638.

DAMIANI, E.; VIMERCATI, S. D. C. di; PARABOSCHI, S.; SAMARATI, P. An open digest-based technique for spam detection. *ISCA PDCS*, v. 2004, p. 559–564, 2004.

DAVIS, J.; GOADRICH, M. The relationship between precision-recall and roc curves. In: ACM. *Proceedings of the 23rd international conference on Machine learning*. [S.l.], 2006. p. 233–240.

DORNELES, C. F.; GONÇALVES, R.; MELLO, R. dos S. Approximate data instance matching: a survey. *Knowledge and Information Systems*, Springer, v. 27, n. 1, p. 1–21, 2011.

FAN, B.; ANDERSEN, D. G.; KAMINSKY, M.; MITZENMACHER, M. D. Cuckoo filter: Practically better than bloom. In: ACM. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. [S.l.], 2014. p. 75–88.

FLOWERS, M.; MCGUIRE, R.; BIRNBAUM, L. Adversary arguments and the logic of personal attacks. *Strategies for natural language processing*, Erlbaum, Hillsdale, NJ, p. 275–294, 1982.

FOSTER, K. *Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus*. 2012. <https://apps.dtic.mil/docs/citations/ADA570831>. Master's thesis, Naval Postgraduate School Monterey (CA). Accessed 2019 Oct 21.

FVC2002. *The Second Fingerprint Verification Competition*. 2018. <http://bias.csr.unibo.it/fvc2002/>. Accessed 2018 Jun 20.

FVC2004. *The Third International Fingerprint Verification Competition*. 2018. <http://bias.csr.unibo.it/fvc2004/>. Accessed 2018 Jun 20.

GARCIA, J. Duplications and misattributions of file fragment hashes in image and compressed files. In: IEEE. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. [S.l.], 2018. p. 1–5.

GARFINKEL, S.; NELSON, A.; WHITE, D.; ROUSSEV, V. Using purpose-built functions and block hashes to enable small block and sub-file forensics. *Digital investigation*, Elsevier, v. 7, p. S13–S23, 2010.

GARFINKEL, S. L. Digital forensics research: The next 10 years. *digital investigation*, Elsevier, v. 7, p. S64–S73, 2010.

GARFINKEL, S. L.; MCCARRIN, M. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigation*, Elsevier, v. 14, p. S95–S105, 2015.

GUPTA, V.; BREITINGER, F. How cuckoo filter can improve existing approximate matching techniques. In: SPRINGER. *International Conference on Digital Forensics and Cyber Crime*. [S.l.], 2015. p. 39–52.

GUTIERREZ-VILLARREAL, F. J. *Improving sector hash carving with rule-based and entropy-based non-probative block filters*. 2015. <https://apps.dtic.mil/docs/citations/ADA620778>. Master's thesis, Naval Postgraduate School Monterey (CA). Accessed 2019 Oct 21.

HARBOUR, N. Dcfldd. *Defense Computer Forensics Lab. http:/dcfldd. sourceforge. net*, v. 5, n. 5.2, p. 1, 2002.

HARICHANDRAN, V. S.; BREITINGER, F.; BAGGILI, I. Bytewise approximate matching: The good, the bad, and the unknown. *The Journal of Digital Forensics, Security and Law: JDFSL*, Association of Digital Forensics, Security and Law, v. 11, n. 2, p. 59, 2016.

INTEL. *Intel SHA Extensions*. 2013. <https://software.intel.com/en-us/articles/intel-sha-extensions>. Accessed 2019 Nov 16.

ISO/IEC-19794-2:2005. Information technology - biometric data interchange formats - part 2: Finger minutiae data. In: . [S.l.: s.n.], 2005.

JAGADISH, H. V.; OOI, B. C.; TAN, K.-L.; YU, C.; ZHANG, R. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, ACM, v. 30, n. 2, p. 364–397, 2005.

JIANZHONG, Z.; KAI, P.; YUNTAO, Y.; JINGDONG, X. ictph: An approach to publish and lookup ctph digests in chord. In: *Algorithms and Architectures for Parallel Processing: 10th International Conference, ICA3PP 2010, Busan, Korea, May 21-23, 2010. Workshops, Part II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 244–253. ISBN 978-3-642-13136-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-13136-3_25>.

KAMEYAMA, A. S.; MOIA, V. H. G.; HENRIQUES, M. A. A. Aperfeiçoamento da ferramenta sdhash para identificação de artefatos similares em investigacoes forenses. In: *Extended Anais of XVIII Brazilian Symposium on information and computational systems security*. Natal-RN, Brasil: SBC, 2018. p. 223–232. Disponível em: <http://portaldeconteudo.sbc.org.br/index.php/sbseg_estendido/article/view/4161>.

KAYAOGLU, M.; TOPCU, B.; ULUDAG, U. Standard fingerprint databases: Manual minutiae labeling and matcher performance analyses. *arXiv preprint arXiv:1305.1443*, 2013.

KIM, H.; AGRAWAL, N.; UNGUREANU, C. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, ACM, v. 8, n. 4, p. 14, 2012.

KO, K. *User's guide to NIST biometric image software (NBIS)*. [S.l.], 2007.

KORNBLUM, J. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, Elsevier, v. 3, p. 91–97, 2006.

LEE, A.; ATKISON, T. A comparison of fuzzy hashes: Evaluation, guidelines, and future suggestions. In: ACM. *Proceedings of the SouthEast Conference*. [S.l.], 2017. p. 18–25.

LI, Y.; SUNDARAMURTHY, S. C.; BARDAS, A. G.; OU, X.; CARAGEA, D.; HU, X.; JANG, J. Experimental study of fuzzy hashing in malware clustering analysis. In: USENIX ASSOCIATION. *8th workshop on cyber security experimentation and test (cset 15)*. [S.l.], 2015. v. 5, n. 1, p. 52.

LILLIS, D.; BECKER, B.; O'SULLIVAN, T.; SCANLON, M. *Current Challenges and Future Research Areas for Digital Forensic Investigation*. 2016. <https://arxiv.org/abs/1604.03850>. Accessed 2019 Oct 21.

LILLIS, D.; BREITINGER, F.; SCANLON, M. Expediting mrsh-v2 approximate matching with hierarchical bloom filter trees. In: SPRINGER. *International Conference on Digital Forensics and Cyber Crime*. [S.l.], 2017. p. 144–157.

LIN, D. An information-theoretic definition of similarity. In: *International Conference on Machine Learning (ICML)*. [S.l.: s.n.], 1998. v. 98, n. 1998, p. 296–304.

NIST. *National Software Reference Library*. 2016. <http://www.nsrl.nist.gov/>. Accessed 2016 Set 13.

NIST. *Special Database 4*. 2018. <https://www.nist.gov/srd/nist-special-database-4>. Accessed 2018 Jun 20.

NOLL, L. C. *Fowler/Noll/Vo (FNV) hash*. 2012. <http://www.isthe.com/chongo/tech/comp/fnv/index.html>. Accessed 2019 Mar 20.

OLIVER, J.; CHENG, C.; CHEN, Y. TLSH–a locality sensitive hash. In: IEEE. *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth*. [S.l.], 2013. p. 7–13.

OLIVER, J.; FORMAN, S.; CHENG, C. Using randomization to attack similarity digests. In: SPRINGER. *International Conference on Applications and Techniques in Information Security*. [S.l.], 2014. p. 199–210.

OLSON, D. L.; DELEN, D. *Advanced data mining techniques*. [S.l.]: Springer Science & Business Media, 2008.

PAGH, R.; RODLER, F. F. Cuckoo hashing. *Journal of Algorithms*, Elsevier, v. 51, n. 2, p. 122–144, 2004.

PARMAR, P. A.; DEGADWALA, S. D. Fingerprint indexing approaches for biometric database: A review. *IJCA*, Citeseer, v. 130, n. 13, p. 0975–8887, 2015.

PEARSON, P. K. Fast hashing of variable-length text strings. *Communications of the ACM*, ACM, v. 33, n. 6, p. 677–680, 1990.

PERALTA, D.; GALAR, M.; TRIGUERO, I.; PATERNAIN, D.; GARCÍA, S.; BARRENECHEA, E.; BENÍTEZ, J. M.; BUSTINCE, H.; HERRERA, F. A survey on fingerprint minutiae-based local matching for verification and identification: Taxonomy and experimental evaluation. *Information Sciences*, Elsevier, v. 315, p. 67–87, 2015.

PUROHITH, D.; MOHAN, J.; CHIDAMBARAM, V. The dangers and complexities of sqlite benchmarking. In: ACM. *Proceedings of the 8th Asia-Pacific Workshop on Systems*. [S.l.], 2017. p. 3.

QUICK, D.; CHOO, K.-K. R. Impacts of increasing volume of digital forensic data: A survey and future research challenges. *Digital Investigation*, Elsevier, v. 11, n. 4, p. 273–294, 2014.

RAFF, E.; NICHOLAS, C. Lempel-ziv jaccard distance, an effective alternative to ssdeep and sdhash. *Digital Investigation*, Elsevier, v. 24, p. 34–49, 2018.

RAGHAVAN, S. Digital forensic research: current state of the art. *CSI Transactions on ICT*, Springer, v. 1, n. 1, p. 91–114, 2013.

RENTZ, D. *Microsoft Compound Document File Format*. [S.l.], 2007.

ROUSSEV, V. Data fingerprinting with similarity digests. In: SPRINGER. *IFIP International Conf. on Digital Forensics*. [S.l.], 2010. p. 207–226.

ROUSSEV, V. An evaluation of forensic similarity hashes. *Digital investigation*, Elsevier, v. 8, p. 34–41, 2011.

ROUSSEV, V.; QUATES, C. Content triage with similarity digests: The m57 case study. *Digital Investigation*, Elsevier, v. 9, p. S60–S68, 2012.

ROUSSEV, V.; QUATES, C. *sdhash tutorial: Release 0.8*. 2013. <http://roussev.net/sdhash/tutorial/sdhash-tutorial.pdf>. Accessed 2016 Set 13.

ROUSSEV, V.; RICHARD III, G. G.; MARZIALE, L. Multi-resolution similarity hashing. *Digit. Investig.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 4, p. 105–113, set. 2007. ISSN 1742-2876. Disponível em: <http://dx.doi.org/10.1016/j.diin.2007.06.011>.

SONI, U. A.; GOYANI, M. M. A survey on state of the art methods of fingerprint recognition. *IJSRSET*, v. 4, 2018.

SQLITE. *SQLite Index*. 2019. <http://www.sqlitetutorial.net/sqlite-index/>. Accessed 2019 Feb 21.

SQLITE. *The SQLite Query Optimizer Overview*. 2019. <https://www.sqlite.org/optoverview.html>. Accessed 2019 Oct 21.

STALLINGS, W. *Cryptography and network security: principles and practice*. [S.l.]: Pearson, 2014. v. 6.

SU, Y.; FENG, J.; ZHOU, J. Fingerprint indexing with pose constraint. *Pattern Recognition*, Elsevier, v. 54, p. 1–13, 2016.

TRIDGELL, A. Spamsum. *URL http://samba.org/ftp/unpacked/junkcode/spamsum/README*, 2002.

UKKONEN, E. On approximate string matching. In: SPRINGER. *International Conference on Fundamentals of Computation Theory*. [S.l.], 1983. p. 487–495.

WANG, J.; SHEN, H. T.; SONG, J.; JI, J. *Hashing for similarity search: A survey*. 2014. <https://arxiv.org/abs/1408.2927>. Accessed 2019 Oct 21.

WANG, Y.; WANG, L.; CHEUNG, Y.-m.; YUEN, P. C. Fingerprint geometric hashing based on binary minutiae cylinder codes. In: IEEE. *Pattern Recognition (ICPR), 2014 22nd International Conference on*. [S.l.], 2014. p. 690–695.

WANG, Y.; WANG, L.; CHEUNG, Y.-M.; YUEN, P. C. Learning compact binary codes for hash-based fingerprint indexing. *IEEE Transactions on Information Forensics and Security*, IEEE, v. 10, n. 8, p. 1603–1616, 2015.

WINTER, C.; SCHNEIDER, M.; YANNIKOS, Y. F2s2: Fast forensic similarity search through indexing piecewise hash signatures. *Digital Investigation*, Elsevier, v. 10, n. 4, p. 361–371, 2013.

ZHANG, J.; LU, H.; LAN, X.; DONG, D. Dhtnil: An approach to publish and lookup nilsimsa digests in dht. In: IEEE. *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*. [S.l.], 2008. p. 213–218.

Appendix

APPENDIX A – Similarity Digest Search Strategies operational costs

In this appendix, we complement Chapter 3, by showing how to estimate the amount of space required by each SDSS when comparing them and analyzing how they would scale with the data set increase. To this end, we developed and adapted some formulas. In our calculus, we had to estimate the number of objects a data set of a given size would have, so we know the number of digests we would have to create for this set, the number of entries in hash tables, the size of bloom filters, and any other required parameter. We estimated an average object size and divide it by the data set size. The chosen object size was 512 KiB (approximation of the size found in some known forensics data sets, as the *govdoc-corpus*[1], for instance). With this information in hand, we proceed our calculus. For the rest of this section, we will consider *n* as the number of objects in the reference list.

A.1 Brute force

The costs associated with the memory requirement for brute force will be calculated for the two most used and known tools nowadays: `ssdeep` and `sdhash`. We will also consider the `TLSH` tool in some scenarios despite none of the strategies found in the literature use it as AM tool. We will show how `TLSH` would perform in an investigation as a brute force strategy due to its interesting characteristics (precision and recall rates and low digest length) and because it is a recently developed technique. Even though `saHash` (BREITINGER *et al.*, 2014c) is newer, it was not chosen because it only works for objects of similar sizes and produces a minimum digests length of 769 bytes (≈ 22 times greater than `TLSH`).

A.1.1 ssdeep and TLSH

The memory consumption of `ssdeep` and `TLSH` are calculated by Eq. A.1.

$$m_{ss} = n \cdot s_{ss} \tag{A.1}$$

where n is the number of digests in the target system and s_{ss} the size of the digest created by `ssdeep/TLSH`.

[1] http://digitalcorpora.org/corpora/files (last accessed Nov 20th, 2019).

A.1.2 sdhash

To calculate the memory requirement for `sdhash`, we need to estimate the number of features extracted from a set of object (on average). According to Breitinger, F. et al. (BREITINGER *et al.*, 2014c), *"`sdhash` maps 160 features into a bloom filter for every approximately 10 KiB of input file"*. This way, we can calculate z (number of features) in the following way:

$$z = (\mu \cdot 2^{20} \cdot 160)/(10 \cdot 2^{10}) = 2^{14} \cdot \mu \tag{A.2}$$

where μ is the reference list size (MiB) and 2^{20} and 2^{10} are factors to change, from MiB and KiB to bytes, respectively. Once we have calculated the number of features on the reference list, we can figure out how many bloom filters will be needed to represent all these features, and hence the memory requirement. To this end, we use Eq. A.3.

$$m_{sd} = (z \cdot s_{bf})/f_{max}\ (bits) \tag{A.3}$$

where s_{bf} is the size of each bloom filter (bits) and f_{max} the maximum number of features allowed to be inserted by sdhash in each filter.

A.2 DHTnil

To calculate the memory requirement for `DHTnil`, we can use Eq. A.4.

$$m_{Dnil} = (n \cdot s_{nil}) + (n_{nodes} \cdot (L_{ft} + L_{rp}))\ (bits) \tag{A.4}$$

where s_{nil} is the size of `Nilsimsa` digests (256 bits), n_{nodes} is the number of nodes in the Chord network, L_{ft} the size of the routing table of each node, and L_{rp} the size of the list of reference points stored in each node. We choose the number of entries in each finger table as $m = log(N)$, where N is the number of nodes. The finger table size is:

$$L_{ft} = (2 \cdot L_{id} \cdot m) + L_{idP} + L_{idS}\ (bits) \tag{A.5}$$

where L_{id} is the ID length of each one of the m entries (Key and value), L_{idP} is the ID length of the predecessor node, and L_{idS} the ID length of the successor node. These values are necessary information to the management of the Chord nodes. Also, L_{rp} refers to a list of digests which represent the reference nodes which is kept by each node to manage the search. Its length is:

$$L_{rp} = n_{rf} \cdot s_{nil}\ (bits) \tag{A.6}$$

where n_{rf} is the reference points number chosen.

A.3 iCPTH

The same formula used by `DHTnil` is applied to `iCPTH` (Eq. A.4). The difference here is the digest size, where `ssdeep` is used instead of `Nilsimsa`.

A.4 F2S2

Eq. A.7 estimates the amount of memory required for this strategy,

$$m_{F2S2} = n \cdot s_{ss} \cdot (1 + p_{fac}) + s_{name}\ (bytes) \tag{A.7}$$

where s_{ss} is the size of ssdeep digests, p_{fac} the payload factor added for the index (between 7-8) and s_{name} the length of each object name in the reference list.

A.5 MRSH-NET

To calculate the amount of memory required for the MRSH-NET approach, we can use the equation from Breitinger's work (BREITINGER *et al.*, 2014a):

$$m_{MRSH-NET} = \frac{k \cdot s \cdot 2^{14}}{ln(1 - \sqrt[k \cdot r_{min}]{p_f})}\ (bits) \tag{A.8}$$

where k denotes the number of sub-hashes, s the object set size in MiB, 2^{14} the number of features in the set s, r_{min} the number of following features required to produce a match, and p_f the probability of false positive for a fragment of an object.

A.6 HBFT

Estimating the amount of memory required for this approach can be done using Breitinger's equations (BREITINGER *et al.*, 2014a). We first need to determine the size of the root bloom filter, using Eq. A.9:

$$m_{BFroot} = \left\lceil -z \cdot \frac{ln(p)}{ln(2)^2} \right\rceil\ (bits) \tag{A.9}$$

where z is the number of features in the set (Eq. A.2) and p the false positive probability for a single feature, calculated by $p = \sqrt[r_{min}]{p_f}$. The parameter r_{min} is the number of consecutive features needed to be found in the filter and p_f the false positive probability for a fragment of an object.

The next step involves calculating the level of the tree, using Eq. A.10.

$$h = log_x(n), \tag{A.10}$$

where x is the degree of the tree (e.g. $x = 2$ for a binary tree).

Then we calculate the memory required for the HBFT structure using Eq. A.11:

$$m_{BFtree} = m_{BFroot} \cdot h\ (bits) \tag{A.11}$$

where m_{BFroot} is root bloom filter size (Eq. A.9) and h the level of the tree (Eq. A.10).

A.7 MRSH-CF

To estimate the amount of memory required for MRSH-CF, we first need to compute the tag size for each item, which can be done using Eq. A.12.

$$f = log_2(1/p) + log_2(2 \cdot b)\ (bits) \tag{A.12}$$

Here, p is the false positive probability for a single feature and b the number of entries of each bucket in the hash table.

Then, we need to estimate the average of bits per item C. According to Fan, B. et al. (FAN *et al.*, 2014), each entry in the hash table stores one fingerprint, but not all of them are occupied. This way, there must be some slack in the table to the avoid failures when inserting new items, making each item cost more than a fingerprint. This value can be calculate by Eq. A.13,

$$C = \frac{f}{\alpha}\ (bits/item) \tag{A.13}$$

where α (load factor) is used to express the percentage of the filter currently used ($0 \leq \alpha \leq 1$).

Finally, we can estimate the amount of memory required by MRSH-CF using Eq. A.14:

$$m_{MRSH-CF} = z \cdot C\ (bits) \tag{A.14}$$

where z is the number of features extracted from the reference list (Eq. A.2) and C the average bit per item (Eq. A.13).

A.8 Parameters

For our experiments, we have adopted the parameters described in Table A.1.

Table A.1 – Similarity digest search strategies experiments - Parameters

Parameter	Value
$s_{ss}(ssdeep)$	96 (bytes)
$s_{ss}(TLSH)$	35 (bytes)
s_{nil}	256 (bits)
s_{bf}	2048 (bits)
f_{max}	160 (features)
k	5 (sub-hashes)
r_{min}	6 (features)
p_f	10^{-6}
p	0.1
x	2 (binary)
α	0.95
b	4 (items/bucket)
p_{fac}	8
s_{name}	10 (bytes)
$L_{id}/L_{idP}/L_{idS}$	160 (bits)
m	10 (nodes)

APPENDIX B – Analysis results of the t5-corpus data set

This appendix presents information and results about one of the data sets used along this thesis: The *t5-corpus*. These information are pertinent to Chapter 5, where we use all these data to perform our experiments.

In Table B.1, we present the objects that compose the *Target* data set, extracted from the *t5-corpus*. The remaining objects of *t5-corpus* compose the *Known* data set. Both sets were used to simulate real-world investigations, where we compared one set to the other to find similarity among objects.

We also present in this appendix the results of a manual analysis performed in some comparisons between *t5-corpus* objects that had some level of similarity pointed out by the AM tools. We classified these matches according to their similarity classes, as defined in Sec. 5.3. Table B.2 summarizes our results. Note that some matches compare objects of different types (e.g., doc vs. html), which is due to similar content or common features.

Table B.1 – List of objects that compose the Target data set (extracted from *t5-corpus*)

000004.doc	000114.doc	000251.doc	000268.doc	000698.doc
000968.doc	001466.doc	001645.doc	001647.doc	002375.doc
002394.doc	002403.doc	002687.doc	003646.doc	004420.doc
004862.doc	004863.doc	002661.doc	003317.doc	003345.doc
000047.xls	000048.xls	000050.xls	000397.xls	000777.xls
001093.xls	001110.xls	001978.xls	002879.xls	004705.xls
000314.ppt	000558.ppt	000712.ppt	000985.ppt	000986.ppt
001891.ppt	001911.ppt	004113.ppt	004610.ppt	004968.ppt
000197.html	000189.html	000199.html	000214.html	000816.html
001329.html	002107.html	002120.html	002123.html	002245.html
002933.html	002950.html	003041.html	003485.html	003497.html
003751.html	003892.html	004338.html	004515.html	004915.html
000020.pdf	000159.pdf	000158.pdf	000168.pdf	000592.pdf
000738.pdf	001278.pdf	001301.pdf	001672.pdf	001675.pdf
002852.pdf	003047.pdf	003049.pdf	003189.pdf	003299.pdf
003668.pdf	003693.pdf	004336.pdf	004448.pdf	004682.pdf

Continued on next page

Table B.1 – *Continued from previous page*

000863.text	000835.text	001818.text	002816.text	002817.text
003112.text	003547.text	003548.text	004222.text	004229.text
000543.gif	000545.gif	000534.gif	004270.gif	004542.gif
001239.jpg	002635.jpg	000906.jpg	004292.jpg	000505.jpg

Table B.2 – List of objects that compose the Known data set (extracted from *t5-corpus*)

Object 1	**Object 2**	**Similarity class**
000003.doc	000002.doc	UGC
004317.doc	004357.html	UGC
004410.doc	004499.html	UGC
001466.doc	001467.doc	UGC
004610.ppt	004558.doc	UGC
004863.doc	004862.doc	UGC
004863.doc	004944.html	UGC
001639.doc	001649.doc	UGC
001647.doc	001630.doc	UGC
001637.doc	001638.doc	UGC
001645.doc	001646.doc	UGC
003043.doc	003044.doc	UGC
002687.doc	002682.doc	UGC
003358.ppt	003344.doc	UGC
002154.doc	002145.doc	UGC
000671.doc	000788.html	UGC
000692.doc	000838.text	UGC
003646.doc	003637.doc	UGC
004420.doc	004413.doc	TC
001619.doc	001621.doc	TC
000274.doc	003617.doc	TC
001264.doc	001263.doc	TC
000699.doc	000697.doc	TC
000005.doc	000004.doc	TC
000005.doc	000696.doc	TC
000005.doc	003345.doc	TC
000005.doc	003331.doc	TC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	Object 2	Similarity class
000004.doc	000696.doc	TC
000004.doc	003345.doc	TC
000004.doc	003331.doc	TC
003345.doc	003331.doc	TC
000696.doc	003345.doc	TC
000696.doc	003331.doc	TC
000968.doc	002154.doc	TC
000968.doc	002145.doc	TC
000968.doc	003619.doc	TC
000251.doc	000968.doc	TC
000251.doc	002154.doc	TC
000251.doc	002145.doc	TC
000251.doc	003619.doc	TC
002145.doc	003619.doc	TC
002154.doc	003619.doc	TC
001873.doc	003646.doc	TC
001873.doc	003637.doc	TC
000277.doc	003646.doc	TC
000277.doc	003637.doc	TC
001631.doc	001643.doc	AGC
001871.doc	004053.doc	AGC
000252.doc	001631.doc	AGC
002661.doc	003634.doc	AGC
002661.doc	004548.doc	AGC
002381.doc	004066.doc	AGC
004077.doc	004066.doc	AGC
004081.doc	004066.doc	AGC
004063.doc	004963.doc	AGC
004419.doc	004411.doc	AGC
001466.doc	002404.doc	AGC
001466.doc	004066.doc	AGC
004419.doc	004963.doc	AGC
004548.doc	004561.doc	AGC
004548.doc	004572.doc	AGC
001465.doc	003634.doc	AGC
001465.doc	003616.doc	AGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	**Object 2**	**Similarity class**
001465.doc	004066.doc	AGC
001465.doc	004548.doc	AGC
003157.doc	003351.doc	AGC
001463.doc	004548.doc	AGC
001628.doc	001618.doc	AGC
001467.doc	002404.doc	AGC
001614.doc	004548.doc	AGC
003320.doc	004063.doc	AGC
003320.doc	004419.doc	AGC
003325.doc	003351.doc	AGC
003325.doc	004066.doc	AGC
003325.doc	004063.doc	AGC
003325.doc	004419.doc	AGC
003338.doc	004066.doc	AGC
003158.doc	003634.doc	AGC
003158.doc	004548.doc	AGC
003318.doc	004063.doc	AGC
003310.doc	004063.doc	AGC
003310.doc	004419.doc	AGC
003319.doc	004063.doc	AGC
003319.doc	004419.doc	AGC
003329.doc	003351.doc	AGC
003329.doc	004063.doc	AGC
003329.doc	004419.doc	AGC
003337.doc	004063.doc	AGC
003337.doc	003351.doc	AGC
003337.doc	004419.doc	AGC
001882.doc	003325.doc	AGC
001882.doc	003310.doc	AGC
001882.doc	003319.doc	AGC
001882.doc	003329.doc	AGC
001882.doc	003337.doc	AGC
001882.doc	003335.doc	AGC
001882.doc	003618.doc	AGC
001882.doc	004411.doc	AGC
001882.doc	004963.doc	AGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	Object 2	Similarity class
003335.doc	003351.doc	AGC
003335.doc	004419.doc	AGC
003618.doc	003351.doc	AGC
003351.doc	004411.doc	AGC
003351.doc	004963.doc	AGC
000698.doc	000693.doc	AGC
000698.doc	001882.doc	AGC
000698.doc	003351.doc	AGC
000698.doc	004063.doc	AGC
000698.doc	004419.doc	AGC
000256.doc	001853.doc	AGC
000256.doc	001882.doc	AGC
000256.doc	004419.doc	AGC
000268.doc	000693.doc	AGC
000268.doc	003351.doc	AGC
000268.doc	004063.doc	AGC
000268.doc	004419.doc	AGC
000272.doc	000691.doc	AGC
000272.doc	001882.doc	AGC
000272.doc	003351.doc	AGC
000272.doc	004419.doc	AGC
003618.doc	004063.doc	AGC
003618.doc	004419.doc	AGC
003634.doc	004073.doc	AGC
003634.doc	004561.doc	AGC
003634.doc	004572.doc	AGC
001618.doc	003634.doc	AGC
001618.doc	004548.doc	AGC
001853.doc	003320.doc	AGC
001853.doc	003325.doc	AGC
001626.doc	003634.doc	AGC
000706.doc	004419.doc	AGC
000694.doc	003634.doc	AGC
000704.doc	004548.doc	AGC
000691.doc	003335.doc	AGC
000693.doc	003319.doc	AGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	Object 2	Similarity class
000703.doc	001628.doc	AGC
000703.doc	003629.doc	AGC
000703.doc	003634.doc	AGC
000703.doc	003616.doc	AGC
000703.doc	004548.doc	AGC
000955.doc	003623.doc	AGC
000955.doc	003634.doc	AGC
000955.doc	004548.doc	AGC
000952.doc	004963.doc	AGC
000978.doc	003634.doc	AGC
000978.doc	004548.doc	AGC
002142.doc	002404.doc	AGC
002142.doc	004066.doc	AGC
002142.doc	004578.doc	AGC
000250.doc	003634.doc	AGC
000263.doc	003634.doc	AGC
000263.doc	004548.doc	AGC
000266.doc	003634.doc	AGC
000266.doc	004548.doc	AGC
000279.doc	003634.doc	AGC
000285.doc	003351.doc	AGC
000285.doc	004419.doc	AGC
002394.doc	002404.doc	AGC
002394.doc	003632.doc	AGC
002394.doc	004066.doc	AGC
002375.doc	003632.doc	AGC
002375.doc	004066.doc	AGC
002387.doc	004066.doc	AGC
002404.doc	003161.doc	AGC
002404.doc	003338.doc	AGC
000701.doc	004066.doc	AGC
000289.ppt	001643.doc	AGC
001281.pdf	001297.pdf	UGC
003975.pdf	003979.pdf	UGC
003049.pdf	003046.pdf	UGC
004968.ppt	004974.pdf	UGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	**Object 2**	**Similarity class**
004967.ppt	004974.pdf	UGC
003300.pdf	003297.pdf	UGC
003299.pdf	003296.pdf	UGC
000148.pdf	000158.pdf	UGC
002463.pdf	002453.pdf	UGC
000348.pdf	000370.pdf	UGC
000743.pdf	000740.pdf	UGC
001363.pdf	001368.pdf	UGC
001054.pdf	003693.pdf	AGC
001301.pdf	004682.pdf	AGC
001689.pdf	003972.pdf	AGC
001939.pdf	003047.pdf	AGC
001939.pdf	004118.pdf	AGC
000167.pdf	001689.pdf	AGC
000592.pdf	001301.pdf	AGC
000592.pdf	003189.pdf	AGC
000765.pdf	003173.pdf	AGC
001675.pdf	001672.pdf	AGC
001675.pdf	002203.pdf	AGC
004683.pdf	004682.pdf	AGC
002852.pdf	003189.pdf	AGC
002852.pdf	004682.pdf	AGC
003189.pdf	004683.pdf	AGC
003189.pdf	004682.pdf	AGC
001301.pdf	002852.pdf	AGC
001301.pdf	003189.pdf	AGC
001301.pdf	004683.pdf	AGC
001672.pdf	002203.pdf	AGC
000152.pdf	001672.pdf	AGC
000152.pdf	002203.pdf	AGC
000592.pdf	002852.pdf	AGC
000592.pdf	004683.pdf	AGC
000746.pdf	001675.pdf	AGC
000746.pdf	001672.pdf	AGC
000746.pdf	002203.pdf	AGC
000152.pdf	000746.pdf	AGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	Object 2	Similarity class
000152.pdf	001675.pdf	AGC
002816.text	002817.text	UGC
003925.text	003932.text	UGC
004222.text	004229.text	UGC
004760.text	004743.html	UGC
004784.text	004805.text	UGC
004790.text	004791.text	UGC
002927.text	002918.html	UGC
003027.text	003026.text	UGC
001817.text	001818.text	UGC
003264.text	003262.text	UGC
000228.text	000227.text	UGC
000228.text	000556.text	UGC
003931.text	003929.text	UGC
004026.text	004027.text	UGC
003926.text	003930.text	UGC
002285.text	002283.text	UGC
000227.text	000556.text	UGC
000835.text	000846.text	UGC
000692.doc	000838.text	UGC
000837.text	000836.text	UGC
001160.text	001163.text	UGC
003548.text	003547.text	UGC
003092.html	003112.text	UGC
003112.text	003111.text	UGC
003012.text	003013.text	UGC
000863.text	004957.text	TC
000497.text	000496.text	TC
004264.text	004419.doc	AGC
004642.pdf	004219.text	AGC
001399.html	001396.html	UGC
003122.html	003123.html	UGC
001795.html	001798.html	UGC
000671.doc	000788.html	UGC
001317.html	001316.html	UGC
002927.text	002918.html	UGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	**Object 2**	**Similarity class**
003745.html	003742.html	UGC
002558.html	003602.html	TC
002558.html	003600.html	TC
001402.html	004341.html	TC
001525.html	001529.html	TC
003494.html	004198.html	TC
003224.html	004198.html	TC
002261.html	002226.html	TC
001405.html	002343.html	TC
001405.html	003095.html	TC
004020.html	004514.html	TC
001574.html	001573.html	TC
001574.html	002023.html	TC
001574.html	003745.html	TC
001574.html	003742.html	TC
001579.html	002955.html	TC
001579.html	002954.html	TC
004482.html	004480.html	TC
001538.html	003993.html	TC
001538.html	004020.html	TC
001538.html	004514.html	TC
001713.html	004340.html	TC
001733.html	003249.html	TC
000450.html	003095.html	TC
001531.html	001771.html	TC
001531.html	003506.html	TC
001531.html	003901.html	TC
001531.html	004920.html	TC
003506.html	003901.html	TC
003506.html	004920.html	TC
001573.html	002023.html	TC
001573.html	003745.html	TC
001258.html	004181.html	TC
000791.html	001708.html	TC
000608.html	003254.html	TC
000428.html	002360.html	TC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	Object 2	Similarity class
002536.html	002531.html	TC
002030.html	003909.html	TC
002034.html	002549.html	TC
002119.html	002934.html	TC
002008.html	003716.html	TC
002107.html	002120.html	TC
003715.html	004934.html	TC
003732.html	003508.html	TC
003474.html	003998.html	TC
003467.html	003219.html	TC
003106.html	004174.html	TC
001797.html	002360.html	TC
001743.html	004347.html	TC
001402.html	003466.html	TC
002757.html	004934.html	TC
002773.html	003254.html	TC
001419.html	003124.html	TC
004934.html	004922.html	TC
001527.html	002549.html	TC
004757.html	004339.html	TC
003901.html	004920.html	TC
003716.html	004757.html	TC
001317.html	001797.html	TC
004347.html	003508.html	TC
002535.html	002561.html	TC
004210.html	003733.html	TC
004343.html	004352.html	TC
001718.html	001703.html	TC
001728.html	002915.html	TC
003257.html	003904.html	TC
000059.html	000060.html	TC
001725.html	001729.html	TC
003466.html	003487.html	TC
004021.html	004019.html	TC
001740.html	004947.html	TC
000412.html	000875.html	TC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	**Object 2**	**Similarity class**
000447.html	001334.html	TC
000416.html	003237.html	TC
000416.html	003902.html	TC
000457.html	003088.html	TC
001116.html	003992.html	TC
001152.html	001155.html	TC
001117.html	003992.html	TC
004014.html	004495.html	TC
004014.html	004503.html	TC
003057.html	003055.html	TC
002263.html	001712.html	TC
002557.html	003487.html	TC
003483.html	004003.html	AGC
001410.html	004003.html	AGC
004014.html	004910.html	AGC
004495.html	004910.html	AGC
004503.html	004910.html	AGC
002627.jpg	004088.ppt	UGC
000132.ppt	002627.jpg	UGC
004296.jpg	004293.jpg	UGC
000521.jpg	000510.jpg	UGC
002627.jpg	002407.doc	AGC
002627.jpg	002834.doc	AGC
002627.jpg	003369.ppt	AGC
002627.jpg	004105.ppt	AGC
002627.jpg	004627.ppt	AGC
002627.jpg	004967.ppt	AGC
002627.jpg	004971.ppt	AGC
002131.jpg	002627.jpg	AGC
002313.ppt	002627.jpg	AGC
002434.ppt	002627.jpg	AGC
000316.ppt	001239.jpg	AGC
000316.ppt	002627.jpg	AGC
002303.jpg	002627.jpg	AGC
002441.ppt	002627.jpg	AGC
000309.ppt	001239.jpg	AGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	Object 2	Similarity class
000309.ppt	002627.jpg	AGC
001239.jpg	002407.doc	AGC
001239.jpg	002627.jpg	AGC
001239.jpg	004088.ppt	AGC
001239.jpg	004105.ppt	AGC
001239.jpg	004967.ppt	AGC
001239.jpg	004971.ppt	AGC
000050.xls	002638.jpg	AGC
000034.xls	002635.jpg	AGC
000883.jpg	001989.xls	AGC
001239.jpg	002356.pdf	AGC
000534.gif	002804.text	AGC
000545.gif	002015.html	AGC
000543.gif	001100.xls	AGC
002407.doc	004292.jpg	AGC
003278.jpg	003652.ppt	AGC
000505.jpg	004292.jpg	AGC
001239.jpg	002131.jpg	AGC
000627.jpg	002836.doc	AGC
002421.ppt	003285.jpg	AGC
000132.ppt	002303.jpg	AGC
003278.jpg	003286.jpg	AGC
001354.jpg	003606.jpg	AGC
003608.jpg	004040.jpg	AGC
003286.jpg	003360.ppt	AGC
000124.doc	002647.jpg	AGC
002441.ppt	003278.jpg	AGC
001239.jpg	002313.ppt	AGC
000541.jpg	004595.ppt	AGC
002092.jpg	003286.jpg	AGC
003278.jpg	003383.ppt	AGC
003583.jpg	004042.jpg	AGC
002092.jpg	002117.jpg	AGC
002647.jpg	003606.jpg	AGC
002134.jpg	002659.jpg	AGC
003360.ppt	003285.jpg	AGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	Object 2	Similarity class
002647.jpg	004623.ppt	AGC
003583.jpg	004432.ppt	AGC
002655.jpg	002659.jpg	AGC
003278.jpg	003584.jpg	AGC
002421.ppt	002659.jpg	AGC
001239.jpg	002441.ppt	AGC
000872.jpg	002117.jpg	AGC
004862.doc	004944.html	UGC
002683.doc	002686.doc	UGC
003646.doc	003637.doc	UGC
001849.doc	001858.doc	UGC
004360.html	004317.doc	UGC
004424.doc	004422.doc	UGC
004548.doc	004547.doc	UGC
001877.doc	001875.doc	UGC
004863.doc	004944.html	UGC
001642.doc	004317.doc	TC
000001.doc	000006.doc	TC
000123.doc	000129.doc	TC
000260.doc	000276.doc	TC
000983.ppt	000966.doc	TC
000254.doc	003612.doc	TC
000957.doc	003612.doc	TC
000984.ppt	000966.doc	TC
000980.doc	003044.doc	AGC
000952.doc	004566.doc	AGC
000967.doc	004066.doc	AGC
000693.doc	003325.doc	AGC
000685.doc	001882.doc	AGC
000691.doc	004553.doc	AGC
000691.doc	000676.doc	AGC
000690.doc	002404.doc	AGC
000706.doc	001886.doc	AGC
000675.doc	000706.doc	AGC
002382.doc	003634.doc	AGC
000265.doc	003632.doc	AGC

Continued on next page

Table B.2 – *Continued from previous page*

Object 1	**Object 2**	**Similarity class**
000254.doc	004046.doc	AGC
002144.doc	003351.doc	AGC
000130.doc	001882.doc	AGC
001628.doc	003158.doc	AGC
002985.doc	003351.doc	AGC
002683.doc	004063.doc	AGC
003391.ppt	004962.doc	AGC
003361.ppt	004962.doc	AGC
004504.html	004583.doc	AGC
002963.text	004054.doc	AGC
001869.doc	002985.doc	AGC
003867.pdf	004569.doc	AGC
000289.ppt	001631.doc	AGC
002403.doc	003662.ppt	AGC

APPENDIX C – Publications derived from this work

- MOIA, V. H. G.; HENRIQUES, M. A. A. Sampling and similarity hashes in digital forensics: An efficient approach to find needles in a haystack. In: XVI Brazilian Symposium on information and computational systems security. Niteroi-RJ, Brazil: Brazilian Computer Society (SBC), 2016. p.693–702.
- MOIA, V. H. G.; HENRIQUES, M. A. A. Similarity Digest Search: A Survey and Comparative Analysis of Strategies to Perform Known File Filtering Using Approximate Matching. Security and Communication Networks, Hindawi-Wiley, v.2017, 2017. p.1-17. Available at: <https://doi.org/10.1155/2017/1306802>.
- MOIA, V. H. G.; HENRIQUES, M. A. A. A comparative analysis about similarity search strategies for digital forensics investigations. In: XXXV Simpósio Brasileiro de Telecomunicações e Processamento de Sinais (SBrT 2017). São Pedro, Brazil: [s.n.], 2017. p.462-466.
- MOIA, V. H. G.; HENRIQUES, M. A. A. An operational cost analysis of similarity digest search strategies using approximate matching tools. In: XVII Brazilian Symposium on information and computational systems security. Brasilia-DF, Brazil: Brazilian Computer Society (SBC), 2017. p.154-167.
- MOIA, V. H. G.; HENRIQUES, M. A. A. Fast Similarity Digest Search: A new strategy for performing queries efficiently with approximate matching. In: XVII Brazilian Symposium on information and computational systems security. Brasilia-DF, Brazil: Brazilian Computer Society (SBC), 2017. p.566-575.
- KAMEYAMA, A. S.; MOIA, V. H. G.; HENRIQUES, M. A. A. Aperfeiçoamento da ferramenta sdhash para identificação de artefatos similares em investigacoes forenses. In: Extended Anais of XVIII Brazilian Symposium on information and computational systems security. Natal-RN, Brazil: SBC, 2018. p.223–232. Available at: <http://portaldeconteudo.sbc.org.br/index.php/sbseg_estendido/article/view/4161>.
- MOIA, V. H. G.; HENRIQUES, M. A. A. A new similarity digest search strategy applied to Minutia Cylinder-Codes for fingerprint identification. In: XVIII Brazilian Symposium on information and computational systems security. Natal-RN, Brazil: Brazilian Computer Society (SBC), 2018. p.99-112.

- MOIA, V. H. G.; BREITINGER, F.; HENRIQUES, M. A. A. Understanding the effects of removing common blocks on Approximate Matching scores under different scenarios for digital forensic investigations. In: XIX Brazilian Symposium on information and computational systems security. Sao Paulo-SP, Brazil: Brazilian Computer Society (SBC), 2019. p.1-14.

- MOIA, V. H. G.; BREITINGER, F.; HENRIQUES, M. A. A. The impact of excluding common blocks for approximate matching. Computers & Security, Elsevier, v.89, p.1-11, 2020. ISSN 0167-4048. Available at: <https://doi.org/10.1016/j.cose.2019.101676>.

- MOIA, V. H. G.; HENRIQUES, M. A. Understanding the capabilities of the sdhash Approximate Matching function: How can we make it better? To be published.