On the utility of bytewise approximate matching in computer science with a special focus on digital forensics investigations

DISSERTATION

approved by

Computer Science Department,
Technische Universität Darmstadt

for the degree of
Doktor-Ingenieurs (Dr.-Ing.)

by

Frank Breitinger, MSc.
born in Erbach (Odw.), Germany

Advisors: Prof. Dr. Stefan Katzenbeisser
Technische Universität Darmstadt

Prof. Dr. Harald Baier
Hochschule Darmstadt

Prof. Dr. Felix Freiling
Friedrich-Alexander-Universität Erlangen-Nürnberg

Date of submission: May 6th, 2014
Date of disputation: June 30th, 2014
University index: D 17

Darmstadt 2014

To my parents.

Declaration of Authorship

English. I, Frank Breitinger, hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated. I am aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any uses of the works of any other author, in any form, are properly acknowledged at their point of use.

Deutsch. Ich, Frank Breitinger, versichere hiermit, die vorliegende Dissertation selbständig und nur mit den angegebenen Quellen / Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Signature: ______________________________
(Frank Breitinger)

Date of submission: May 6th, 2014

Wissenschaftlicher Werdegang

März 2011 - Juni 2014 (PhD Student)

Promotionsstudent an der Technischen Universität Darmstadt.
Okt. 12 - Feb. 14: Wissenschaftlicher Mitarbeiter an der Hochschule Darmstadt und CASED.
Okt. 13 - Dez. 13: Forschungsaufenthalt am National Institute of Standards and Technology (Gaithersburg, MD, USA).
Mrz 11 - Okt. 12: Stipendiat an der Hochschule Darmstadt und CASED.

März 2009 - Februar 2011 (Master-Studium)

Informatikstudium an der Hochschule Darmstadt (Vertiefung IT-Sicherheit).
Sept. 10 - Feb. 11: Masterarbeit mit dem Schwerpunkt Digitale Forensik am CASED (Darmstadt).

Oktober 2005 - Februar 2009 (Bachelor-Studium)

Informatikstudium an der Hochschule Mannheim.
Aug. 08 - Feb. 09: Bachelorarbeit bei der Sobedi GmbH (Mannheim).
Sept. 07 - Feb. 08: Praxissemester an der University of Maryland (MD, USA).

List of Publications

Publications (chronological order)

1. F. Breitinger, G. Stivaktakis, V. Roussev: Evaluating Detection Error Trade-offs for Bytewise Approximate Matching Algorithms. *Digital Investigation*. Mai 2014.
2. Frank Breitinger, Barbara Guttman, Michael McCarrin, and Vassil Roussev: Approximate matching: Definition and terminology. *NIST Special Publication 800-168*. DRAFT. 2014.
3. F. Breitinger and V. Roussev: Automated evaluation of approximate matching algorithms on real data. *Digital Forensics Research Conference Europe (DFRWS EU'14)*. Amsterdam (Netherlands). May 2014.
4. F. Breitinger, H. Baier and D. White: On the database lookup problem of approximate matching. *Digital Forensics Research Conference Europe (DFRWS EU'14)*. Amsterdam (Netherlands). May 2014.
5. F. Breitinger, G. Ziroff, S. Lange and H. Baier: Similarity Hashing Based on Levenshtein Distance. *Tenth Annual IFIP WG 11.9 International Conference on Digital Forensics (IFIP WG11.9)*, Vienna (Austria). January 2014.
6. F. Breitinger, C. Winter, Y. Yannikos, T. Fink and M. Seefried: Reducing Data for Forensic Investigations Using Approximate Matching. *Tenth Annual IFIP WG 11.9 International Conference on Digital Forensics (IFIP WG11.9)*, Vienna (Austria). January 2014.
7. C. Rathgeb, F. Breitinger, C. Busch, H. Baier: On the Application of Bloom Filters to Iris Biometrics. In *IET Biometrics*, 2013.
8. F. Breitinger, G. Stivaktakis, V. Roussev: Evaluating Detection Error Trade-offs for Bytewise Approximate Matching Algorithms. *5th International Conference on*

Digital Forensics & Cyber Crime (ICDF2C). Moscow (Russia). September 2013. (Best paper award)[1].

9. F. Breitinger, H. Liu, C. Winter, H. Baier, A. Rybalchenko and M. Steinebach. Towards a process model for hash functions in digital forensics. *5th International Conference on Digital Forensics & Cyber Crime (ICDF2C)*. Moscow (Russia). September 2013.

10. F. Breitinger, G. Stivaktakis and H. Baier. FRASH: A framework to test algorithms of similarity hashing, *In Proceedings of the 13th Digital Forensics Research Conference (DFRWS)*, Monterey (Californien, US). August 2013.

11. C. Rathgeb, F. Breitinger and C. Busch. Alignment-Free Cancelable Iris Biometric Templates based on Adaptive Bloom Filters. *In Proceedings of the 6th IAPR International Conference on Biometrics (ICB)*. Madrid (Spain). June 2013.

12. F. Breitinger, K. Åstebøl, H. Baier and C. Busch. mvHash-B - a new approach for similarity preserving hash function. *7th International Conference on IT Security Incident Management & IT Forensics (IMF)*. Nuernberg (Germany). March 2013.

13. F. Breitinger and K. Petrov. Reducing time cost in hashing operations. *Ninth Annual IFIP WG 11.9 International Conference on Digital Forensics (IFIP WG11.9)*. Orlando (Florida, US). January 2013.

14. F. Breitinger and H. Baier. Similarity Preserving Hashing: Properties and a new Algorithm MRSH-v2. *4th International Conference on Digital Forensics & Cyber Crime (ICDF2C)*. Lafayette (Indiana, US). October 2012.

15. F. Breitinger and H. Baier. Properties of a Similarity Preserving Hash Function and their Realization in sdhash. *Information Security South Africa (ISSA)*. Johannesburg (South Africa). August 2012.

16. F. Breitinger, H. Baier and J. Beckingham. Security and Implementation Analysis of the Similarity Digest sdhash. *1st International Baltic Conference on Network Security & Forensics (NeSeFo)*. Tartu (Estonia). August 2012.

17. F. Breitinger and H. Baier. A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance. *7th annual Conference on Digital Forensics, Security and Law (ADFSL)*. Richmond (Virgina, US). pp. 89-101. May 2012.

18. F. Breitinger and H. Baier. Performance Issues about Context Triggered Piecewise Hashing. *3rd International ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*. Dublin (Ireland). October 2011.

---

[1] Due to the best paper award, this paper was published in *Digital Investigation* (see Bib-item number 1). Therefore, the proceedings of this conference only include a 2-page abstract.

19. H. Baier and F. Breitinger. Security Aspects of Piecewise Hashing in Computer Forensics. *6th International Conference on IT Security Incident Management & IT Forensics (IMF)*. pp. 21-36. Stuttgart (Germany). May 2011.
20. F. Breitinger and C. Nickel. User Survey on Phone Security and Usage. *Biometrie und elektronische Signaturen (BIOSIG 2010)*. pp. 139-144. Darmstadt (Germany). September 2010.

Other Publications / Talks

1. F. Breitinger. Reducing data for forensic investigations using approximate matching. *Invited talk at University New Haven.* New Haven (Connecticut, USA), 18 February 2014. Presentation.
2. Barbara Guttman, F. Breitinger, S. Garfinkel, J. Kornblum and C. Shields: Approximate Matching of Digital Artifacts. *13th Digital Forensics Research Conference (DFRWS'13)*, Monterey (Californien, US). August 2013. Panel Discussion.
3. F. Breitinger. Similarity Preserving Hashing. *8. GI SIG SIDAR Graduate Workshop on Reactive Security (SPRING)*. Muenchen (Germany), 18-19 February 2013. Presentation.
4. F. Breitinger. Similarity Preserving Hashing. *CAST Workshop - Forensik und Internetkriminalitaet.* Darmstadt (Germany), 20. December 2012. Presentation.
5. F. Breitinger and H. Baier. Security Aspects of Piecewise Hashing in Computer Forensics. *6. GI SIG SIDAR Graduate Workshop on Reactive Security (SPRING)*. Technical Report SR-2011-01, pp. 11. Bochum (Germany), 21-22 March 2011. Abstract.

Acknowledgments

First of all I would like to thank my parents who played the most important role throughout my whole education. They convinced me to study computer sciences and supported me in all aspects of my studies. I got their help and advice during difficult times and in good times we rejoiced together. They have allowed me become the person I am today. Thank you!

Special thanks go to Harald Baier who encouraged me to continue my scientific career after my masters, offered me a PhD position and acted as my supervisor. His supervision was perfect scientific guidance, which helped me to establish and develop myself in the community. The numerous discussions we had extended my knowledge and improved my research.

I also want to thank Stefan Katzenbeisser and Felix Freiling for accepting me as a PhD student, for their feedback on my work and for acting as supervisors. Additionally, I like to thank Max Mühlhäuser and Stefan Roth for being part of the examination committee, and Matthias Hollick for being the head of the committee.

Furthermore, I would like to say thank you to Vassil Roussev, Ibrahim Baggili and Steffen Lange, as well as my colleagues Sebastian Abt, Christian Rathgeb and Christoph Busch for valuable discussions. Thank you to the NIST forensic group, especially Barbara, Douglas and Terri, who enabled my internship in Gaithersburg and provided me with a pleasant stay.

Special credit goes to all students who supported me in the last three years in different ways – especially Vikas, Knut and Kaloyan.

Finally I'd like to thank all people who have came along with me during my PhD time but are not explicitly mentioned. In this context, I want to express my appreciation to the University of Applied Sciences Darmstadt (h_da), the Technical University Darmstadt (TUD) and the Center for Advanced Security Research Darmstadt (CASED), for their administrative, technical, and financial support throughout the past three years.

Abstract

Handling hundreds of thousands of files is a major challenge in today's digital forensics. In order to cope with this information overload, investigators often apply hash functions for automated input identification. Besides identifying exact duplicates, which is mostly solved running cryptographic hash functions, it is also necessary to cope with similar inputs (e.g., different versions of files), embedded objects (e.g., a JPG within a office document), and fragments (e.g., network packets). Thus, the essential idea is to complement the use of cryptographic hash functions, to detect data objects with bytewise identical representation, with the capability to find objects with bytewise similar representations.

Unlike cryptographic hash functions, which have a wide range of applications and have been studied as well as tested for a long time, approximate matching algorithms are still in their early development stages. More precisely, currently the community is missing a definition, an evaluation methodology and (additional) fields of application.

Therefore, this thesis aims at establishing approximate matching in computer sciences with a special focus on digital forensic investigations. One of our firsts step was to develop a generic definition for approximate matching, in collaboration with the National Institute of Standards and Technology (NIST) which is applicable to the different levels approximate matching, e.g., bytewise and semantic. A subsequent detailed analysis of both existing approaches uncovers different strengths and weaknesses, therefore we present improvements. To extend the range of algorithms, this work introduces three of our new algorithms, that are based on well-known techniques of computer sciences.

A core contribution of this thesis is the open source evaluation framework called `FRASH` which assesses tools on different criteria. Besides traditional properties (borrowed from hash functions) like generation efficiency and space efficiency (compression), we conceive methods to determine precision and recall rates based on synthetic as well as real world data.

Since digital investigations are often time critical, we improve the performance of automated file identification by a mechanism we call prefetching. Compared to a straight

forward analysis, the performance increases by almost 40% without additional hardware. In this context we also discuss the impact of different hashing/approximate matching algorithms for digital investigations and conclude that it is absolutely reasonable to apply crypto hashing as well as bytewise/semantic approximate matching algorithms in a prosecution.

To extend the fields of application, this thesis demonstrates the capabilities of applying approximate matching on network traffic analysis and biometric template protection. Our research shows that approximate matching is perfectly suited for data leakage prevention and can also be applied for biometric template protection, biometric data compression and efficient biometric identification.

Zusammenfassung

Heutzutage ist das Bearbeiten von hunderttausenden Dateien eine der wesentlichen Herausforderungen der digitalen Forensik. Um diese Datenflut zu bewältigen verwenden Ermittler oft Hashfunktionen zur automatisierten Dateierkennung. Neben exakten Duplikaten, wofür meist kryptographische Hashfunktionen verwendet werden, ist es jedoch auch hilfreich ähnliche Dateien (z.B. verschiedene Versionen einer Datei), eingebettete Objekte (z.B. JPG in einem Office Dokument) und Fragmente (z.B. Netzwerkpakete) zu erkennen. Die wesentliche Idee ist den klassischen Ansatz der kryptographischen Hashfunktionen zu ergänzen und somit auch ähnliche Bytesequenzen zu klassifizieren.

Im Gegensatz zu kryptographischen Hashfunktionen, die zahlreiche Anwendungsgebiete haben und ausgiebig getestet werden, sind ähnlichkeitserhaltende Algorithmen (engl. Approximate matching algorithms) noch in einer frühen Entwicklungsphase. Genaugenommen fehlt der Community derzeit eine Definition, eine Evaluations-Methodologie und weitere Anwendungsgebiete.

Diese Dissertation hat daher das Ziel ähnlichkeitserhaltende Algorithmen in der Informatik, mit besonderem Blick auf die digitale Forensik, zu etablieren. In einem ersten Schritt präsentieren wir eine generische Definition, die in Zusammenarbeit mit dem National Institute of Standards and Technology (NIST) entstanden ist. Im Anschluss werden die beiden existierenden Algorithmen untersucht um Stärken / Schwächen zu analysieren. Um das Sortiment dieser Algorithmen zu erweitern, präsentieren wir drei weitere, eigene Algorithmen, die auf bewehrten Techniken der Informatik basieren.

Ein wesentlicher Bestandteil dieser Arbeit ist das open source Evaluationsframework `FRASH` welches Tools anhand verschiedener Kriterien bewertet. Neben den traditionellen Eigenschaften wie Laufzeit oder Kompression, haben wir Methoden entwickelt um die Precision & Recall Raten auf synthetischen und realen Daten zu messen.

Da Ermittlungen in der digitalen Forensik oft zeitkritisch sind, haben wir ein Verfahren namens ‘prefetching’ entwickelt, welches die Laufzeit um ca. 40% verbessert, ohne zusätzliche Hardware. In diesem Zusammenhang wird auch der Einfluss unterschiedlicher Verfahren für kriminalistische Ermittlungen untersucht mit dem Entschluss,

dass es durchaus Bedarf für unterschiedliche Algorithmen (d.h. auch kryptographische Hashfunktionen, semantische Hashfunktionen) in der Strafverfolgung gibt.

Abschließend zeigen wir weitere Anwendungsgebiete und wenden ähnlichkeitserhaltende Algorithmen zur Netzwerkanalyse und zum Schutze biometrischer Daten an. Die Ergebnisse zeigen, dass diese Techniken sich zum Schutze sensibler Daten eignen (Data leakage prevention) und auch in der Biometrie für Template Protektion eingesetzt werden können.

Contents

1. Introduction **1**
- 1.1. Research questions . . . 2
- 1.2. Document structure . . . 3

2. Foundations & definition **7**
- 2.1. Hash functions . . . 7
- 2.2. Use cases . . . 8
- 2.3. Problem description . . . 10
  - 2.3.1. Identification of exact duplicates . . . 10
  - 2.3.2. Identification of near duplicates . . . 11
- 2.4. Definition and terminology for approximate matching . . . 13
  - 2.4.1. Use cases . . . 13
  - 2.4.2. Terminology . . . 14
  - 2.4.3. Essential requirements . . . 15
  - 2.4.4. Reliability of results . . . 16
- 2.5. Bloom filter . . . 16
- 2.6. Test basics . . . 17
- 2.7. Summary . . . 18

3. Related work **19**
- 3.1. Context-triggered piecewise hashing (`ssdeep`) . . . 20
  - 3.1.1. Extensions . . . 22
  - 3.1.2. The F2S2 Software . . . 23
  - 3.1.3. Security analysis . . . 24
- 3.2. Multi-resolution similarity hashing (`mrsh`) . . . 25
- 3.3. Similarity digest hashing (`sdhash`) . . . 26
  - 3.3.1. Extensions and improvements . . . 29

3.3.2. Comparison and evaluation of `ssdeep` and `sdhash` 30
3.4. SimHash 31
3.5. Summary 31

4. Assessment & improvement of existing approximate matching approaches 33
4.1. Context-triggered piecewise hashing 33
4.1.1. Enhancements overview 33
4.1.2. Experimental results 35
4.2. Similarity digest hashing analysis 37
4.2.1. Implementation 38
4.2.2. Compression & runtime efficiency 38
4.2.3. Coverage 39
4.2.4. Similarity score 42
4.2.5. Security analysis 42
4.3. Multi-resolution similarity hashing version 2 (`mrsh`) 45
4.3.1. Pseudo random function 45
4.3.2. Chunk processing 46
4.3.3. Similarity digest 47
4.3.4. Comparing similarity digest 48
4.4. Summary 48

5. Algorithms, concepts and applications 49
5.1. Random building block hashing (`bbHash`) 49
5.1.1. Building blocks 50
5.1.2. Detailed proceeding 50
5.1.3. Assessment and experimental results 53
5.2. Majority vote hashing (`mvhash`) 55
5.2.1. Algorithm design and proceeding 55
5.2.2. Design decisions 59
5.2.3. Assessment and experimental results 61
5.3. Statistical analysis hashing (`saHash`) 62
5.3.1. Algorithm design and proceeding 62
5.3.2. Assessment and experimental results 67
5.4. A concept for an efficient similarity digest lookup 69
5.4.1. Proceeding overview 70
5.4.2. Design decisions 72
5.4.3. Workflow for trees 75
5.4.4. Theoretical assessment for Bloom filter digests in trees 77
5.4.5. Special use case: only a single Bloom filter 80
5.5. File fragment detection on network traffic 86
5.5.1. Foundations 86
5.5.2. Solution and proceeding overview 88
5.5.3. Test methodology 90
5.5.4. Experimental results & assessment 93

5.6. Summary 97

6. Testing bytewise approximate matching 99
6.1. Test categories 100
    6.1.1. Efficiency 100
    6.1.2. Sensitivity & robustness 101
    6.1.3. Precision & recall tests 102
6.2. Realization of tests 103
    6.2.1. Sensitivity & robustness 103
    6.2.2. Precision & recall on synthetic data 104
    6.2.3. Precision & recall on real world data 104
    6.2.4. Approximate longest common substring 106
6.3. Testing results 107
    6.3.1. Efficiency 109
    6.3.2. Sensitivity & robustness 110
    6.3.3. Precision & recall on synthetic data 114
    6.3.4. Precision & recall on real world data 124
6.4. Implementation details of the tests 131
6.5. Summary 133

7. Towards signature based similarity detection in forensic investigations 135
7.1. Reducing time cost in hashing operations 136
    7.1.1. Framework overview or overview of multi-threading with prefetching. 136
    7.1.2. Implementation details 139
    7.1.3. Experimental results 142
    7.1.4. Distinction to existing parallelization tools 145
7.2. Reducing data for forensic investigations using approximate matching . . . 146
    7.2.1. Methodology and experimental Setup 147
    7.2.2. Assessment and Results 150
7.3. Overview semantic approximate matching 154
7.4. Benefits and drawbacks of approximate matching and hashing for investigations 156
    7.4.1. Efficiency 156
    7.4.2. Robustness and discriminability for semantic approaches 158
    7.4.3. Wrap-up of experimental results 160
7.5. Sample use case: analyzing a USB hard disk drive 160
7.6. Summary 162

8. Excursus - Bloom filter in iris recognition 165
8.1. Motivation 165
8.2. Contribution 166
8.3. Combining Bloom filters and iris recognition 167
    8.3.1. Adaptive Bloom filter-based transformation 167
    8.3.2. Comparison in transformed domain 168

8.3.3. Template protection 169
8.3.4. Biometric data compression 170
8.3.5. Adaptive Bloom filter-based identification 170
8.4. Summary 171

9. Conclusion & future work 173
9.1. Answers to research questions 173
9.2. Future work 175

List of Appendices i
A. Evaluation of `sdhash` implementation i
A.1. Popularity Rank Computation Bug i
A.2. Design errors within the comparison class iv
B. All results of the precision & recall test for synthetic data v

Bibliography xxi

1

Introduction

The trend is that everything is going digital, e.g., books, photos, letters and long-playing records (LPs) turned into ebooks, digital photos, email and mp3. Thus, a requirement of modern forensic investigations is the ability to perform large-scale automated filtering and the correlation of data, which reduces the amount of data an investigator has to look at by hand.

A common method deployed for this purpose is known file filtering, which is quite simple: compute the crypto hashes for all files on a target device and compare them to a reference database. Depending on the underlying database, files are either filtered out (e.g., files of the operating system) or filtered in (e.g., known offensive content). A very common database for filter out data is the NSRL [68] maintained by National Institute of Standards and Technology (NIST).

However, the security properties of cryptographic hash functions require that the files are completely identical – a difference by a single bit produces a pseudo randomly hash value. Although this property is desired for cryptographic purposes, it complicates forensic investigations. For instance, one cannot do similarity detection (e.g., different versions of a file), embedded object detection (e.g., JPG in a office document), fragment detection (e.g., analyzing a device on the byte level or network packages) or clustering files (e.g., e-mails and Word documents with similar content). Therefore, it is useful to have algorithms that provide *approximate* matches that can correlate closely related versions of data objects.

While hashing has a long tradition in various fields of application like cryptography [64], databases [56, Sec. 7.8.2] or digital forensics [1, p.56+], approximate matching is a rather new area. Generally one distinguishes between bytewise– and semantic approximate matching.

Bytewise approximate matching[1] had its breakthrough in 2006 with an algorithm called context-triggered piecewise hashing [57]. The essential idea was to complement the use of cryptographic hash functions to detect data objects with bytewise identical representation with the capability to find objects with bytewise similar representations. Thus, it relies

---

[1] Well-known synonyms are fuzzy hashing and similarity hashing.

only on the sequence of bytes that make up a digital artifact, without reference to any structures (or their interpretation) within the data stream. It is the most general, in that it can compare *any* two blobs of binary data and relies on the assumption that similarities between objects are reflected by similarities in their byte level representation.

Semantic approximate matching[2] operates at a higher level of abstraction and provides results that are closest to human perceptual notions of similarity. For example, facial recognition methods could correlate images of the same person.

Although previous research on bytewise approximate matching demonstrated the usefulness of these approaches, there are several issues missing in order to establish them in the community:

- The community requires a definition / standardization for this kind of algorithms.
- Approximate matching will only be accepted by both the scientific community and practitioners if an assessment methodology based on well-known criteria as well as an evaluation of existing approaches is available [42, 37].
- Similar to hashing, approximate matching might have multiple application fields which need to be highlighted and proven.

To solve these issues we come up with research questions presented in the following section which are the motivation for this dissertation.

1.1. Research questions

The leading research question for this dissertation is

What is the utility of bytewise approximate matching and how can we establish it in the computer science community?

To answer this question and the open issues of previous research, we followed a traditional development process and conclude to these more specific research questions (RQ):

Requirements:

RQ1 - Requirements: What is a possible definition for approximate matching?

RQ2 - Analysis of existing approaches: Do the existing algorithms have any strengths or weaknesses (by design) and is it possible to improve them?

Design and implementation:

RQ3 - New concepts for approximate matching: Can we develop new approaches based on well established proceedings from computer science?

---

[2] Well-known synonyms are robust hashing and perceptual hashing.

RQ4 - Efficiency: How can we improve the quadratic lookup complexity of similarity digests?

Verification:

RQ5 - Testing: What is a reasonable methodology to test and evaluate bytewise approximate matching?

Application:

RQ6 - Process model: How can approximate matching improve and support the current forensic investigation process?

RQ7 - Use cases in forensics: Are there further applications/concepts for approximate matching besides automated file identification in 'dead system analysis'[3]?

RQ8 - Further application fields: Is it possible to adapt concepts from byte approximate matching to improve existing biometric template protection schemes?

1.2. Document structure

The introduction has already motivated the necessity and benefits of bytewise approximate matching. In addition, we have shown current drawbacks which concluded in our research questions. The remaining thesis is structured as follows:

Chapter 2 - Foundations & definition

Traditional hash functions are sophisticatedly defined, well-known and established in many different fields such as cryptography, databases or checksums. This chapter briefly presents the properties of hash functions and their deployment in digital forensics. The core of this chapter is the definition, terminology and use cases of approximate matching. At the end we explain Bloom filters which play an important role throughout this thesis and provide some basic information about test data and time measurements.

Chapter 3 - Related work

This chapter summarizes the relevant literature. We briefly discuss the origin of finding similarities between two objects which has a long history and has started in the early twentieth century. Next, we focus on fingerprint based similarity detection (later called approximate matching) that first became popular in 2006 with an algorithm called context-triggered piecewise hashing and an implementation named `ssdeep`. As it was the first of its kind, a few improvements as well as a security analysis were published. Besides we also outline a second algorithm called `sdhash` and a comparison of both algorithms. The last part of this chapter explains `SimHash` which is an implementation for detecting near duplicates only.

---

[3] The process to analyze an HDD when the working station is turned off.

Chapter 4 - Assessment & improvement of existing approximate matching approaches

In here, we focus on analyzing and improving existing approximate matching approaches. First, we present a performance improvement for context-triggered piecewise hashing which also increases the security. With respect to `sdhash`, we highlight weaknesses, peculiarities and improvements which are the result of a detailed design analysis. The last part releases `mrsh-v2` which originates from combining techniques from multi-resolution similarity hash, `ssdeep` and `sdhash`.

Chapter 5 - Algorithms, concepts and applications

Besides improvements on existing algorithms, we also worked on three of our own ideas. Random building block hashing (`bbHash`) was inspired by data deduplication and eigenfaces (biometrics) which are both well-established procedures in computer sciences. Statistical analysis hashing (`saHash`) estimates the byte level similarity of two objects with respect to the Levenshtein distance which makes it unique. Our last approach called majority vote hashing (`mvhash`) is based on the widespread compression technique run-length encoding. The last two sections present a new concept as well as a new application. First, we introduce an idea that improves the database lookup complexity of similarity digests and second, we demonstrate the possibilities of approximate matching on network traffic.

Chapter 6 - Testing bytewise approximate matching

The challenge of testing bytewise approximate matching is a main issue of this thesis and discussed at this point. Basically testing is divided into three parts: efficiency, sensitivity & robustness, and precision & recall. The efficiency tests are borrowed from traditional hash functions and measure the generation efficiency, comparison efficiency and the space efficiency (compression). The second test class focuses on the absolute performance of an algorithm, e.g., what is the smallest fragment an algorithm can identify. Finally, we developed techniques to calculate the precision & recall rates based on synthetic and real world data. The final part of this chapter gives a brief overview of our testing framework called `FRASH`.

Chapter 7 - Towards signature based similarity detection in forensic investigations

After presenting and evaluating some approaches, this chapter shows ideas of how approximate matching can support forensic investigations. In the beginning we highlight a framework that increases the overall runtime efficiency when using hashing/approximate matching in investigations. Next, we briefly present semantic approximate matching approaches, which may also play an important role during an investigation, followed by a brief comparison of different existing hashing and approximate matching techniques. As a last point, we demonstrate a possible usage of these algorithms based on a sample use case.

Chapter 8 - Excursus - Bloom filter in iris recognition

This excursus demonstrates the flexibility of approximate matching and shows how to apply techniques from bytewise approximate matching in biometrics. More precisely, we developed a technique to use approximate matching for biometric template protection, biometric data compression and efficient biometric identification.

Chapter 9 - Conclusion & future work

Based on the results described in the previous chapters, this chapter presents the final conclusion where we answer the research questions. The very last part of this thesis proposes possible future work.

2

Foundations & definition

Hash functions have a long tradition in computer science and various fields of application. Therefore, they are well-known, studied frequently and are properly defined. Besides their common application in cryptography, they are utilized in databases for indexing purposes or in digital forensics. However, their usage in digital forensics is especially questionable as they were never designed for this purpose. The community therefore suggests a new approach called approximate matching which was designed for forensic purposes.

The overall idea of this chapter is to give a common understanding and terminology of hash functions and approximate matching. In Sec. 2.1 we summarize the properties of traditional / cryptographic hash functions which is followed by use cases and limits of these algorithms with respect to digital forensics. Sec. 2.3 is an abstract description of the overall problem. The core of this chapter is Sec. 2.4 which provides common terminology, properties and use cases for approximate matching. This part is the result of a collaboration of leading researchers in approximate matching as well as the National Institute of Standards and Technology. It was published as a NIST special publication [18]. The last two sections briefly explain Bloom filter (Sec. 2.5) and test basics (Sec. 2.6).

2.1. Hash functions

Let $\{0,1\}^*$ denote a set of bit strings of arbitrary length, and let $bS \in \{0,1\}^*$. If we write h for a hash function, then according to [64], h is a function with two properties:

Compression: $h : \{0,1\}^* \longrightarrow \{0,1\}^n$ for $n \in \mathbb{N}$.

Ease of computation: Computation of $h(bS)$ is 'fast' in practice.

In practice bS is a 'document' (e.g., a file, a volume, a device). The output of the function $h(bS)$ is referred to as a *hash value, fingerprint, signature* or *digest*. Sample security applications of hash functions comprise storage of passwords (e.g., on Linux systems), electronic signatures (both MACs and asymmetric signatures), and whitelists/blacklists in digital forensics.

For use in cryptography, hash functions have three further conditions:

Preimage resistance: Let a hash value $H \in \{0,1\}^n$ be given. Then it is infeasible **in practice** to find an input (i.e., a bit string bS) with $H = h(bS)$.

Second preimage resistance: Let a bit string $bS_1 \in \{0,1\}^*$ be given. Then it is infeasible **in practice** to find a second bit string bS_2 with $bS_1 \neq bS_2$ and $h(bS_1) = h(bS_2)$.

Collision resistance: It is infeasible **in practice** to find any two bit strings $bS_1, bS_2 \in \{0,1\}^*$ with $bS_1 \neq bS_2$ and $h(bS_1) = h(bS_2)$.

These security conditions have an important consequence regarding the output of a hash function: Let bS and $h(bS)$ be given. If bS is replaced by bS', $h(bS')$ behaves pseudo randomly, i.e., we do not have any control over the output, if the input changes. This effect is called *avalanche effect*. According to this, if only one single bit in bS is changed to get bS', the two outputs $h(bS)$ and $h(bS')$ look 'very' different. More precisely, every bit in $h(bS')$ changes with probability of 50%, independently of the number of different bits in bS'. Sample cryptographic hash functions are given in Table 2.1.

Table 2.1.: Sample cryptographic hash functions.

Name	MD5	SHA-1	SHA-256	SHA-512	RIPEMD-160
n	128	160	256	512	160

2.2. Use cases

Nowadays a popular use case is to employ hashing methods for *known file filtering* of files which is quite simple: An investigator computes the crypto hashes for all files on a target device and compares them to a reference database. Depending on the underlying database, files are either filtered out (e.g., files of the operating system) or filtered in (e.g., known offensive content). Files not found in the database remain unclassified.

The decision between filter out and filter in depends on the underlying database where the established terms are:

Blacklisting means that the database contains illegal or suspicious inputs that we like to highlight (filter in), e.g., child abuse or leaked company secrets (intellectual property).

Whitelisting means that the database contains non-criminal files that should be masked out, e.g., operating system files or well-known programs.

As it is a challenging task to maintain a capacious database, often global ones are used. The most prominent database for filter out data is the *National Software Reference Library* (NSRL, [68]) from National Institute of Standards and Technology (NIST). They regularly publish *Reference Data Sets* (RDS) containing hashes of the collected software products. The current RDS 2.42 covers approximately 115 million files[1].

In addition to the RDS, there are also non-RDS hash datasets, that may also be used within a digital forensic investigation. For example, there are data sets for SHA-256, MD5 of the first 4096 bytes of a file or an RDS containing `ssdeep` digests (a.k.a. 'fuzzy hashes')[2].

NIST points out that its RDS is not a whitelist, as it also contains entries of files which may be suspicious in some countries depending on regional law, e.g., steganographic tools or hacking scripts. However, there is no illicit data within the RDS, e.g., child abuse images.

Limits of cryptographic hash functions. Currently, mostly cryptographic hash functions are applied for filter in/out data. These algorithms show good results in terms of ease of computation and compression. In addition, they are the most recognized in court (so far) as they do not err, i.e., their security properties allow to identify equal files with nearly 100% probability (except MD5), which is very important for whitelisting.

The impact of applying cryptographic hash functions in computer forensics was analyzed by White [103] as well as Baier & Dichtelmüller [49]. While White propagates an identification rate of 85%, Baier & Dichtelmüller only obtained rates between 15% and 52%. This low detection rates result from changing files during updates. Besides, it is very likely that word/excel documents, logfiles or source code change over the time.

As a consequence, in recent years the forensic community came up with approximate matching which aims at preserving resemblance by mapping similar inputs to similarity digests (similar hash values).

Forensic software. In order to do blacklisting/whitelisting, there are a couple of commercial tools for forensic investigators [50, p11]. All of these tools have an interface to import the RDS from NIST. However, besides exact duplicate detection, common tools start to integrate technologies for similar object detection. For instance:

EnCase by Guidance Software[3] is probably the most common forensic tool. The software has the possibility to identify similar malware using an entropy near-match analyzer technology as said in [90], but this functionality is not available for other files.

Moreover, a script is available which aims at finding partial file matches using `ssdeep` [43]. It searches in non-allocated clusters for fragments of files and compares them against previously created hash-sets.

X-Ways Forensics by X-Ways Software Technology AG[4] seems not to include an approximate matching algorithm. However, there is functionality at the semantic

---

[1] `http://www.nsrl.nist.gov/RDS/rds_2.42/READ_ME.txt` (last accessed 18-Dec.-2013).

[2] `http://www.nsrl.nist.gov/ssdeep.htm` (last accessed 2013-12-18).

[3] `http://www.guidancesoftware.com/` (last accessed 2014-03-12).

[4] `http://www.x-ways.de/` (last accessed 2014-03-12).

level for pictures (i.e., image recognition) as stated in [105, post #118]: “Known pictures can be recognized even if they are stored in a different file format, resized, if the colors or the quality are different or they have been edited, etc.”

Forensik Toolkit (FTK) by AccessData[5] contains an approximate matching utility [51]. The method is advertised to save time for investigators, to match parts of files to the original one or to identify similar files. Although no detailed description is available, it seems that the FTK implements `ssdeep`.

Despite the positive assessment, the manufacturers warn against relying entirely on the results: “investigator will still be required to make the final decision on whether certain documents in a case are similar or not” [51, p9].

Regarding blacklisting, tools like Perkeo ([45], a German blacklisting tool) and Artemis [46] are popular in Germany and accepted in court. Perkeo is a special data scanner for child abuse whose basic database is maintained by the German Federal Criminal Police Office (BKA).

2.3. Problem description

Let X be a finite set and $X_B \subseteq X$, where usually the amount of elements in X_B is much less than in X, i.e., $|X| \gg |X_B|$. For instance, we are focusing in the binary level then X is the set of all byte sequences of a maximum length.

2.3.1. Identification of exact duplicates

Problem A. Let $x \in X$ and let the set X_B be given. It is of interest to answer the following question: Does $x \in X$ belong to X_B?

Solution for Problem A. To answer the question of interest, proceed as follows: Test sequentially for each $x' \in X_B$ whether or not $x = x'$ and return the corresponding answer.

Remark. Obviously, it is guaranteed that this answer is correct. But this procedure has two major problems.

1. In order to compare x and x', we need to have x' available (memory/space problem).
2. A comparison of x and x' might be very time consuming.

In order to solve this problem we use a compression function[6] $h : X \rightarrow Y$ having the following general requirements (GR):

GR1 - The computation $h(x)$ is fast in practice for every $x \in X$ (*ease of computation*).

---

[5] `http://accessdata.com/` (last accessed 2014-03-12).

[6] Note that every traditional hash function serves as an appropriate compression function.

GR2 - There is a constant $c > 0$ such that $|h(x)| = c$ for all $x \in X$ (*compression*).

- **GR2.1** - Current algorithm output a variable length digest and do not fulfill $GR2$.

Problem A′. Let h and the set $h(X_B)$ be given. Using a compression function h, the following question is of interest: Does an $x \in X$ belong to X_B?

Solution for Problem A′. To answer the question of whether or not $x \in X_B$, compute $y = h(x)$, test if $y \in h(X_B)$ and return the corresponding answer. Note, it is not necessarily required to compare y against all elements of $h(X_B)$.

Remark. This solution may be false as we have a one-sided error:

1. Although $x \notin X_B$ it is possible that $h(x) \in h(X_B)$. We denote this by a *false match*.

We accept false matches due to performance and memory/space issues. However, we need to optimize it to have only few false matches.

2.3.2. Identification of near duplicates

Let d_x denote a function $d_x : X \times X \to \mathbb{R}_0^+$ having two properties:

- $\forall x_1, x_2 \in X : d_x(x_1, x_2) = 0$ if and only if $x_1 = x_2$ (coincidence).
- $\forall x_1, x_2 \in X : d_x(x_1, x_2) = d_x(x_2, x_1)$ (symmetry).
- $\forall x_1, x_2, x_3 \in X : d_x(x_1, x_3) \leq d_x(x_1, x_2) + d_x(x_2, x_3)$ (triangle inequality).

d_x is used to measure the similarity/distance between two inputs and thus we call it *distance function*. Clearly, similarity depends on the application context. The upper bound of distance is represented by $\varepsilon_x \in \mathbb{R}$, i.e., two inputs $x, x' \in X$ are considered to be similar, if and only if $d_x(x, x') \leq \varepsilon_x$.

Problem B. Let X_B, d_x and ε_x be given. Then we denote $C(X_B, d_x, \varepsilon_x) = \{x \in X \mid d_x(x, x') \leq \varepsilon_x \text{ for an } x' \in X_B\}$. It is of interest to answer the following question: Does $x \in X$ belong to $C(X_B, d_x, \varepsilon_x)$?

Solution for Problem B. To answer the question of interest, proceed as follows: Test sequentially for each $x' \in X_B$ whether or not $d_x(x, x') \leq \varepsilon_x$ and return the corresponding answer.

Remark. Equal to *Problem A*, it is guaranteed that this answer is correct. However, there are the same performance problems and therefore a similar solution is required.

Additionally, the set $C(X_B, d_x, \varepsilon_x)$ is not explicitly given and a computation of $h(C(X_B, d_x, \varepsilon_x)) = \{h(x) \mid x \in C(X_B, d_x, \varepsilon_x)\}$ is too inefficient in practice, we need an alternative.

In order to solve *Problem B* we introduce *approximate matching* which consists of a *feature extraction function* (h) and a *distance function* (d_y) which is defined as follows:

> **GR3** - $d_y : h(X) \times h(X) \to \mathbb{R}_0^+$ is a comparison function that outputs a distance score for two fingerprints.

Problem B′. Let h, the set $h(X_B)$, ε_y and d_y be given. Then we denote $C(h(X_B), d_y, \varepsilon_y) = \{y \mid d_y(y, y') \leq \varepsilon_y \text{ for an } y' \in h(X_B)\}$. It is of interest to answer the following question: Does $x \in X$ belong to $C(X_B, d_x, \varepsilon_x)$.

Solution for Problem B′. To answer the question of whether or not $x \in C(X_B, d_x, \varepsilon_x)$, compute $y = h(x)$, test if $y \in C(h(X_B), d_y, \varepsilon_y)$ and return the corresponding answer.

Remark. This solution may be false as we have a two-sided error:

1. Although $x \notin C(X_B, d_x, \varepsilon_x)$ it is possible that $h(x) \in C(h(X_B), d_y, \varepsilon_y)$. We denote this by false match.
2. Although $x \in C(X_B, d_x, \varepsilon_x)$ it is possible that $h(x) \notin C(h(X_B), d_y, \varepsilon_y)$. We denote this by false non-match.

We accept false matches and false non-matches due to performance and memory/space issues. However, we need to optimize them.

Additional requirements

In order to be usable in practice, there are requirements concerning correctness and similarity preserving:

1. $\forall x_1, x_2 :$ If $d_y(h(x_1), h(x_2)) \leq \varepsilon_y$ then $d_x(x_1, x_2) \leq \varepsilon_x$ (*correctness*).
2. $\forall x_1, x_2 :$ If $d_x(x_1, x_2) \leq \varepsilon_x$ then $d_y(h(x_1), h(x_2)) \leq \varepsilon_y$ (*similarity preserving*).

As mentioned the solution may be false due to a two-sided error. Therefore, correctness cannot always be fulfilled and might be replaced by usability:

3. $P(d_y(h(x_1), h(x_2)) \leq \varepsilon_y | d_x(x_1, x_2) > \varepsilon_x)$ should be small (*usability*).

2.4. Definition and terminology for approximate matching

Approximate matching is a generic term describing any technique designed to identify similarities between two digital artifacts. In this context, an *artifact* (or an *object*) is defined as an arbitrary byte sequence, such as a file, which has some meaningful interpretation.

Different approximate matching methods may operate at different levels of abstraction. At the lowest level, generic techniques may detect the presence of common byte sequences (substrings) without any attempt to interpret the artifacts. At higher levels, approximate matching can incorporate more abstract analysis that is closer to what a human analyst might do. The overall expectation is that lower level methods would be faster, and more generic in their applicability, whereas higher level ones would be more targeted and require more processing.

One common approach in security and forensic analysis is to find *identical* objects using cryptographic hashing. Approximate matching can be viewed as a generalization of that idea in that, instead of providing a yes/no $\{0, 1\}$ answer to a comparison, it provides a range of outcomes, $[0, 1]$, with the result interpreted as a measure of similarity.

2.4.1. Use cases

Broadly, there are two types of similarity queries that are of interest–*resemblance* and *containment* [28]. In the case of resemblance, we compare two similarly sized objects and interpret the result as a measure of the commonality between them; for example, two successive versions of a piece of code are likely to resemble each other substantially. When the compared objects differ in size significantly, such as a file and a whole-disk image, the test for commonality is interpreted as a *containment* query because it addresses the question of whether the large object contains the smaller one.

An approximate matching algorithm should be able to handle at least one of the following challenges (divided according to whether the query type is (R)esemblance or (C)ontainment) [81, 23]:

> *Object similarity detection (R):* identify related artifacts, e.g., different versions of a document.
>
> *Cross correlation (R):* identify artifacts that share a common object, e.g., a Microsoft Word document and a PDF document containing the same image, or other embedded object.
>
> *Embedded object detection (C):* identify a given object inside an artifact, e.g., an image within a compound document, or an executable inside a memory capture.
>
> *Fragment detection (C):* identify the presence of traces/fragments of a known artifact, e.g., identify the presence of a file in a network stream based on individual packets.

In most analytical scenarios, approximate matching is used to *filter* data *in*, or *out*, based on a known reference set. In security monitoring applications, approximate matching could potentially be used to *blacklist* known bad artifacts, and (by extension) anything closely resembling them. However, approximate matching is not nearly as useful when it comes to *whitelisting* artifacts as malicious content can often be quite similar to benign content; e.g., a backdoored `ssh` server would look very similar to a regular one.

2.4.2. Terminology

Based on the level of abstraction of the similarity analysis performed, approximate matching methods can be placed in one of three main categories [19]:

Bytewise matching relies only on the sequences of bytes which make up a digital object, without reference to any structures within the data stream, or to any meaning the byte stream may have when appropriately interpreted. Such methods have the widest applicability as they can be applied to any piece of data; however, they also carry the implicit assumption that artifacts that humans perceive as similar have similar byte-level encodings. The validity of this assumption varies widely and the analysts must have the appropriate background to interpret the results correctly.

Syntactic matching uses internal structures present in digital objects. For example, the structure of a TCP network packet is defined as an international standard and matching tools can make use of this structure during network packet analysis to match the source, destination or content of the packet. Syntax-sensitive similarity similarity measurements are specific to a particular class of objects that share an encoding but require no interpretation of the content to produce meaningful results.

Semantic matching uses contextual attributes of the digital object to interpret the artifact in a manner that more closely resembles human cognitive processing. For example, perceptual hashes allow the matching of visually similar images and are unconcerned with the low-level details of how the images are persistently stored. Semantic methods tend to provide the most specific results but also tend to be the most computationally expensive ones.

Approximate matching algorithms work in two phases–in the first, a *similarity digest* representation (also referred to as *signature*, or *fingerprint*) is generated for the original data; in the second phase, digests are compared to produce the result. More precisely:

Similarity digest. A similarity digest is a (compressed) representation of the original data object that is suitable for comparison with other similarity digests created by the same algorithm. In most cases, the digest is much smaller than the original artifact and the original object is not recoverable from the digest.

Every approximate matching technique requires at least two core functions:

Feature extraction function, which identifies and extracts features/attributes from each object, allowing a compressed representation of the original object. The mechanism by which features are picked and interpreted depends on the approximate matching algorithm. The representation of this collection is the *similarity digest* of the object; the number or fraction of features which are shared by objects defines their similarity.

Similarity function, which compares two similarity digests and outputs a score s in the $0 \leq s \leq 100$ range, where 0 indicates no similarity and 100 indicates high similarity. Despite its range, this value is not necessarily an estimation of the percentage commonality between the compared objects but rather a measure of confidence. It is primarily meant to serve as a means to sort and filter the results.

In current literature, researchers use a number of terms to refer to various approximate matching methods: *fuzzy hashing*, *similarity hashing*, and *similarity digest* denote bytewise approximate matching; *perceptual hashing* and *robust hashing* denote semantic approximate matching. There appears to be no pre-existing terminology for syntactic approximate matching as it is mostly viewed as pre-processing (to separate the features) before hashing, or applying a bytewise approximate matching algorithms. For example, network flows are usually reconstructed before any processing is done on them.

2.4.3. Essential requirements

Like traditional hash functions, there are several defining characteristics that approximate matching functions should exhibit. Each algorithm should define how it incorporates each of these properties and how it satisfies the reporting requirements for those properties, where appropriate.

Similarity preservation: The algorithms should yield similar 'similarity digests' for similar inputs where the algorithm defines how it measures similarity. The similarity measure may include multiple attributes, and should be accompanied by a measure of the accuracy of the matching technique under the circumstances in which it is used in addition to the matching score, e.g., a margin of error or confidence level. The description of the technique should also state whether it identifies exact matches as such.

Compression: A compact similarity digest is desired as it normally allows a faster comparison and requires less storage space. In the best case, it will have a fixed length like the output of traditional hash functions. If the efficiency and reliability of the results remains unchanged, then a shorter similarity digest is preferable.

Ease of computation: First, the algorithm description should include the results of testing the runtime efficiency of the *feature extraction function* and of the *similarity function*. The former might be expressed relative to a standard hashing algorithm, such as SHA-1.

Second, the algorithm description should state the theoretically complexity for a similarity digest comparison which is known as O-notation. For instance, common lookup complexities for comparing a single digest against a database with n entries,

are
$O(1)$, e.g., crypto hashes stored in hash tables,
$O(\log_2 n)$, e.g., crypto hashes stored in binary trees or
$O(n)$, e.g., no indexing/sorting is possible and bruteforce is needed.

2.4.4. Reliability of results

The reliability of the results for a given approximate matching technique depends on three factors. Each algorithm should define how it incorporates these factors and how it satisfies their reporting requirements.

Sensitivity & robustness: The algorithms should provide some measure of their robustness. A technique's robustness will define the operating conditions in which it can function effectively, also called its *performance envelope*. For example, robustness addresses the minimum and maximum object sizes that an algorithm can reliably distinguish between.

Precision & recall: The algorithms should include a description of the methods used to determine its reliability and to select the test data. Specifically, it should indicate whether test data is culled from existing collections or developed solely to specifically support testing. Test results may include precision & recall rates as well as false positive and false negative rates.

Security of results: The algorithms should indicate whether they include security properties designed to prevent attacks. Such attacks include manipulation of the matching technique or input data such that a data object appears dissimilar to another object to which it is similar or similar to another object with which it has little in common.

2.5. Bloom filter

Bloom filters [6] have a wide field of applications, e.g., database applications [67] or network applications [30] and are commonly used to represent elements of a finite set $\mathbf{S}$. A Bloom filter bf is an array consisting of m bits initially all set to zero. In order to 'insert' an element $s \in \mathbf{S}$ into the filter, k independent hash functions are needed where each hash function h outputs a value in the range $[0, \ldots, m-1]$. Next, s is hashed by all hash functions h. To insert s, the bits $bf[h_0(s)], bf[h_1(s)], \ldots, bf[h_{k-1}(s)]$ of the Bloom filter bf are set to one.

To answer the question if s' is in $\mathbf{S}$, we compute $h_0(s'), h_1(s'), \ldots, h_{k-1}(s')$ and analyze if the bits at the corresponding positions in the Bloom filter are set to one. If this holds, s' is assumed to be in $\mathbf{S}$, however, these bits may be set to one by different elements previously inserted to $\mathbf{S}$. Hence, Bloom filters suffer from a non-trivial false positive rate. Otherwise, if at least one bit is set to zero, we know with certainty that $s' \notin \mathbf{S}$, i.e., it is obvious that the false negative rate is equal to zero.

In case of uniformly distributed data the probability that a certain bit is set to one during the insertion of an element is $1/m$, i.e., the probability that a bit is still zero is $1 - 1/m$. After inserting n elements into the Bloom filter, the probability of a given bit position to be one is $1-(1-1/m)^{k \cdot n}$. In order to have a false positive, all k array positions need to be set to one. Hence, the probability p for a false positive is

$$\begin{aligned} p &= \left(1 - (1 - 1/m)^{kn}\right)^k \\ &\approx \left(1 - \mathrm{e}^{-kn/m}\right)^k \quad \text{for} \quad \frac{1}{m} \ll 1. \end{aligned} \tag{2.1}$$

If we fix m and n, we can determine k such that the false positive probability minimizes. Looking at the derivate of p as a function of k shows that p is minimal for

$$k = m/n \cdot \ln(2). \tag{2.2}$$

2.6. Test basics

This section briefly presents test relevant information and terms which are specially relevant for chapter 5 and 6.

Test data. Basically we distinguish between two kinds of test data throughout this thesis. When talking about random data, we actually mean pseudo random data gathered from `/dev/urandom/` or an analogous program function.

Besides, we also work on real world data which is represented by the *t5-corpus* [82]. This set was introduced by Roussev to evaluate `ssdeep` and `sdhash` in 2011. t5 is a subset of the govdocs corpus [33] which “are real files obtained by spidering US Government websites and are free of copyright restrictions” [81].

The t5-corpus contains 4457 files of the file types given in Table 2.2 with a total size of 1.78 GB. All files are between 4 kB and 16.4 MB which corresponds to an average of 418.91 kB per file.

Table 2.2.: Statistics of the t5-corpus.

JPG	GIF	DOC	XLS	PPT	HTML	PDF	TXT
362	67	533	250	368	1093	1073	711

Measuring times. In general there are three different times [91]:

- **Real** is wall clock time, the time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).

- **User** is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process.

2.7. Summary

This chapter provided the required foundations. First, we explained the general properties (compression and ease of computation) and security properties (preimage resistance, second-preimage resistance and collision resistance) of cryptographic hash functions. As described in the use cases, it is very common to apply them for file identification during forensic investigations (i.e., blacklisting and whitelisting), although hash functions were designed for cryptographic purposes.

Next was an abstract problem description which basically goes back to the nearest neighborhood search. The core of this chapter was the definition and terminology of approximate matching. Besides the use cases, we explained the three levels of approximate matching, bytewise–, syntactic– and semantic approximate matching and that each algorithm requires at least two functions: a feature extraction function and a similarity function. The output of approximate matching is called similarity digest. In addition to this terminology, we presented essential requirements and discussed the reliability of the results.

At the end we briefly introduced Bloom filters which is an important concept for some approximate matching algorithms. Finally, the last section described the test data and the measurement of times.

3

Related work

Identifying similarity has a long history which may go back to the beginning of the 19^{th} century and Paul Jaccard. In order to identify the similarity between two finite sets A and B, the author suggested the Jaccard index [53, 54]: $J(A, B) = \frac{|A \bigcap B|}{|A \bigcup B|}$. For instance, two strings are decomposed into tokens (e.g., by spaces), which are the elements of the respective sets A and B. Then $J(A, B)$ is used to identify the similarity of the two input strings. However, the sets have to be kept in memory to compute $J(A, B)$ which can be very space consuming. Assuming that we split a string (a long byte sequences) into 4-grams of bytes, then in the worst case, we have to keep $2^{4 \cdot 8} = 2^{32}$ different 4-grams in memory, i.e., 16 GiB. Nowadays a common application of the Jaccard index is within computer linguistics for plagiarism and authorship detection.

In 1997, Broder showed the necessity to distinguish between 'roughly the same' (resemblance) and 'roughly contained inside' (containment) [28] which so far hasn't been considered by standard string distance functions (e.g., Levenshtein). To define resemblance r, he utilized the Jaccard index $r(A, B) = J(A, B)$. With respect to containment c, Broder suggested to use $c(A, B) = \frac{|A \bigcap B|}{|A|}$ which "indicates that A is roughly contained within B".

In addition, he reduced these set intersection problems by a process of random sampling which can be done independently for each document. The result is an efficient method to estimate these similarities between two sets [29], called MinHash. Instead of comparing the sets completely, he utilized a hash function $h : A \bigcup B \rightarrow \mathbb{N}$. Further, let S be any set, then $h_{min}(S)$ denotes the minimum hash value in S. According to this, if $h_{min}(A) = h_{min}(B)$, then the minimum hash value is within the intersection $A \bigcap B$ and thus $P[h_{min}(A) = h_{min}(B)] = J(A, B)$. As this is very imprecise, one may apply k different hash functions instead of only one. Thus, estimating $J(A, B)$ is equal to $\overline{k}/k$ where $\overline{k}$ denotes the amount of hash functions with $h_{min}(A) = h_{min}(B)$. To sum it up, one can decide between runtime efficiency (k is small, e.g., $k = 1$) and precision ($k \approx |A \bigcap B|$). In case of a high precision which is desirable during forensic investigations, the runtime efficiency is reduced due to applying k hash functions on all $x \in A \bigcup B$.

A different idea to quantify similarities among text files was presented by Manber in

1994 [63] and implemented in `sif`. “Files are considered similar if they have a significant number of common pieces, even if they are very different otherwise.” `sif` uses a set of *anchors* which are short character sequences. In order to test for similarity, `sif` searches for anchors and considers the neighborhood, e.g., the next 50 characters. As comparing strings directly is time consuming, Manber integrated Rabin fingerprinting [74] to hash the substrings and thus, it was possible to compare numeric values. The main problem is that training data is needed in order to identify reasonable anchors. For instance, text files of different languages need a different set of anchors.

Another concept which is sometimes linked with approximate matching and may be confusing is *locality sensitive hashing* (LSH) [75]. Note, LSH is mostly used for data clustering or nearest neighbor search and thus is more like an indexing strategy. The basic idea of LSH is to process (mostly hash) an input so that similar inputs are mapped to the same buckets with a certain probability. In contrast, approximate matching reduces inputs to small digests whereby similar files are mapped to similar digests.

The beginning of bytewise approximate matching within forensics was in 2002 where Harbour developed a program called `dcfldd`[1], which extends the well-known disk dump tool `dd`. The aim of the tool is to ensure integrity on the sector level during imaging. Therefore, the software splits the input data into chunks of a fixed length (e.g., 512 bytes) and computes the hash value for each of these blocks. Thus, we call this approach *block based hashing*. A key property of Harbour’s method is that a flipping bit in the input only affects the hash output of the corresponding block. However, deleting a byte in the beginning shifts the offsets of all following blocks and leads to a completely different sequence of hash values.

In recent years, approximate matching has become more and more popular and new approaches have been published.

3.1. Context-triggered piecewise hashing (`ssdeep`)

This section introduces the concept of context-triggered piecewise hashing (CTPH) which is probably the best-known approximate matching algorithm. It was presented by Kornblum in 2006 [57] and is based on the spam detection algorithm from Tridgell [99]. The implementation is freely available and is currently in version `ssdeep 2.10`[2].

The overall idea of `ssdeep` is a version of Rabin’s [74] seminal work on data fingerprinting by random polynomials. CTPH identifies trigger points to divide a given byte sequence into chunks. The trigger points are determined by a pseudo random function (PRF) as follows: A window of a fixed size s (`ssdeep` sets $s = 7$) moves through the whole input, byte for byte, and generates a pseudo random number at each step. Let

$$BS_p = B_{p-s+1} B_{p-s+2} \dots B_p \tag{3.1}$$

denote the byte sequence BS in the window of size s consisting of the bytes B_{p-s+1}

---

[1] `http://dcfldd.sourceforge.net` (last accessed 2014-03-10).

[2] `http://ssdeep.sourceforge.net` (last accessed 2013-03-10).

to B_p at position p within the input file. Then, $PRF(BS_p)$ denotes the corresponding rolling hash value. If $PRF(BS_p)$ hits a certain value, the end of the current chunk is identified. We call the current byte sequence BS_p a *trigger sequence*. The subsequent chunk starts at byte B_{p+1} and ends at the next trigger sequence or EOF.

Kornblum denotes the PRF as a rolling hash which is based on the Adler-32 function. The structure of the rolling hash function as proposed in [57] allows to compute $PRF(BS_{p+1})$ cheaply from the previous value $PRF(BS_p)$. Kornblum updates the value $PRF(BS_{p+1})$ by removing the influence of B_{p-s+1} and considering the new byte B_{p+1}. Algorithm 1 shows the pseudocode of the rolling hash implemented in `ssdeep`. As there are only low-level operations, Kornblum's PRF is very efficient in practice.

Algorithm 1 Pseudocode of the rolling hash

```
$h_1, h_2, h_3$                ▷ Unsigned 32-bit integers, set to zero after each trigger point.
$B_i$                          ▷ Current byte at position $i$.
$window$                       ▷ Is an array of length $size$ (default $size = 7$).

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

In order to define a hit for $PRF(BS_p)$, Kornblum introduces a modulus, which he calls a *block size*. If bs denotes the block size, then BS_p is a trigger sequence if and only if $PRF(BS_p) \equiv -1 \bmod bs$. If PRF outputs equally distributed values, then the probability of a hit is reciprocally proportional to bs. Thus, if bs is too small, we have too many trigger points and vice-versa. As Kornblum aims at having 64 chunks, the block size has to be approximately $bs_{init} \approx N/S$ where S is the desired number of chunks with a default value of 64, and N is the file size in bytes. To obtain an equal block size for similar sized files, he generates the initial block size bs_{init} as follows:

$$bs_{init} = bs_{min} \cdot 2^{\lfloor \log_2(N/S \cdot bs_{min}) \rfloor} , \tag{3.2}$$

where the minimum block size bs_{min} is set to 3. However, we discovered that the calculation of the block size as given in the description does not conform with the implementation (floor vs. ceiling operation in the exponent) [4]. Actually the block size is calculated as follows:

$$bs_{init} = bs_{min} \cdot 2^{\lceil \log_2(N/S \cdot bs_{min}) \rceil} . \tag{3.3}$$

Once a chunk is identified, it is hashed using 32-bit FNV-1a [69]. To save space, Kornblum only takes the least significant 6 bits of the hash, which is a Base64 character.

Thus, the similarity digest of `ssdeep` for a file is simply the concatenation of all Base64 hash characters.

Since the block size is used for determining the chunks and depends on the length of the input, only similarity digests with the same block size can be compared. To be a little bit more flexible two different block sizes are used: bs_{init} and $2\,bs_{init}$. If there are too few Base64 characters for block sizes bs_{init} (i.e., at most $S/2 - 1 = 64/2 - 1 = 31$), Kornblum sets $bs_{init} \leftarrow bs_{init}/2$ and the whole process is repeated.

```
$ ssdeep file1
ssdeep,1.0--blocksize:hash:hash,filename
384:exQOElbn4NOTSNVyCCvIiebjYKKjoKUTDeueZdTmvk1ac9slOjRXMImRHgnY5:
   A8ZQVy6jYKKElPeXZdT1NaHJ5,"/home/user/file1"
```

Figure 3.1.: A sample similarity digest of `ssdeep`.

A sample output of `ssdeep` is given in Fig. 3.1. The `ssdeep` similarity digest is computed over the file `file1` with its 24,000 bytes. The number at the beginning of the output of `ssdeep` is the block size used to trigger the PRF. In our example the block size is 384, which can be computed with Eq. 3.3 or estimated by $24,000/64 = 375$. Next are both similarity digests for block sizes 384 and $2 \cdot 384 = 768$, respectively. Finally, we see the path and name of the processed file.

3.1.1. Extensions

In 2008, Chen and Wang [32] described an efficiency improvement which is supposed to reduce the scan passes. Recall, if the `ssdeep` similarity digest is too short then $bs_{init} \leftarrow bs_{init}/2$ and the input is processed again.

The authors tested with 1575 different files from Linux and Windows systems and showed that this re-processing happens in 38% of the cases. They modified `ssdeep` to "generate intermediate hashes using numbers in the geometric progression with factor 4 as block size, generate hashes with other block sizes used in spamsum by rehashing the intermediate hashes to decrease the scan passes and hashing passes." More formally, bs_{init} is determined by

$$bs_{init} = \begin{cases} bs_{min}, & \text{if } N < 2 \cdot S \cdot bs_{min} \\ 2 \cdot bs_{min} 4^{\lfloor log_4(N/(2 \cdot S \cdot b_{min})) \rfloor}, & \text{else} \end{cases} \tag{3.4}$$

where bs_{min} is set to 3. According to [32], "3 and 6 are used for small pieces, following numbers are quadruple of the prior one (24, 96, 384, etc.). [...] With current block size bs, we compute two traditional hashes h and H at block level bs and $4\,bs$"[3] and count the trigger sequences for bs, $2\,bs$, $4\,bs$ and $8\,bs$. An example is given in Fig. 3.2 which shows that there are 8 pieces for bs, 5 pieces for $2\,bs$, and so on. Then, the authors claim that "the hash of the first piece at $2\,bs$ level can be computed from $h1$, $h2$, $h3$." In other

---

[3] The original paper used the variable b which we replaced by bs.

words, the authors used FNV as a homomorphic hash function and created the hashes for $2\,bs$ by using the hashes of bs. Actually, FNV isn't a homomorphic hash function and hence we do not think that this procedure is working.

<table>
<tr><td>recorded block size</td><td>b</td><td>b</td><td>2b</td><td>4b</td><td>b</td><td>8b</td><td>4b</td><td>2b</td></tr>
<tr><td>pieces (b)</td><td>h1</td><td>h2</td><td>h3</td><td>h4</td><td>h5</td><td>h6</td><td>h7</td><td>h8</td></tr>
<tr><td>pieces (2b)</td><td colspan="3">1</td><td>2</td><td colspan="2">3</td><td>4</td><td>5</td></tr>
<tr><td>pieces (4b)</td><td colspan="4">H1</td><td colspan="2">H2</td><td>H3</td><td>H4</td></tr>
<tr><td>pieces (8b)</td><td colspan="6">1</td><td colspan="2">2</td></tr>
</table>

Figure 3.2.: Similarity digest generation overview [32][4].

In 2011, the PRF of `ssdeep` was improved with respect to both efficiency and randomness [4]. As a consequence, the performance of `ssdeep` is enhanced with respect to both speed and stochastic properties of CTPH. The exact implementation is shown in Algorithm 2 which is a variation of the rolling hash.

The main two differences compared to Kornblum's one are first the initial values for the registers $h3$ and $h2$, and second the right-shift-operation on register $h2$. $h2$ is used to mutate the higher bits of the rolling hash (`c << 24`) and $h3$ for the lower bits. Due to the multiplication in the return-value, $h1$ influences all 32 bits too.

In order to test the randomness of a PRF, we first used an existing framework of NIST [86]. This framework is a test suite for the validation of pseudo random number generators for cryptographic applications consisting of 15 different tests: Frequency (Monobit) Test, Frequency Test within a Block, Runs Test, tests for the Longest-Run-of-Ones in a Block, and so on. Furthermore, we analyzed the behavior for four files all a of different type. We conclude that our proposal of a PRF induces significantly less second preimage collisions.

The runtime efficiency was analyzed measuring the time to process a 50 MiB and a 100 MiB from `/dev/urandom` and considers exclusively the rolling hash. The results show that the new implementation is approximately 1.5 times faster.

3.1.2. The F2S2 Software

This section briefly describes an indexing scheme for `ssdeep` similarity digests in order to speed up the database comparison.

Let z be the amount of similarity digests in the database. Cryptographic hash values are stored within hash tables or binary trees and hence the lookup complexity for a single

---

[4] Note, the authors used b for block size, which we denote by bs.

Algorithm 2 Pseudocode of the new rolling hash

```
$h_1, h_2, h_3$                    ▷ Unsigned 32-bit integers, set to zero after each trigger point.
$B_i$                              ▷ Current byte at position $i$.
$window$                           ▷ is an array of $size$ (default $size = 7$)
$h2 = 0x81a5c9f3$
$h3 = 0xa51fbc31$

to update the rolling hash for a byte $c$
    h1 = h1 + $B_i$
    h1 = h1 - window[i mod size]            ▷ h1: is the sum of the bytes in the window
    window[i++ mod size] = $B_i$

    h2 = (h2 >> 5) ⊕ ($B_i$ << 24)          ▷ h2: right-shift; new byte influences higher bits
    h3 = (h3 << 5) ⊕ $B_i$                  ▷ h3: left-shift; new byte influences lower bits

    return h3 ⊕ h2 ⊕ (h1*0x7FFF FFFF)       ▷ 0x7FFF FFFF = $2^{31} - 1$
```

digest is $O(1)$ or $O(\log_2(z))$, respectively. Considering approximate matching, it is not possible to sort digests in such a trivial way. Therefore, Winter *et al.* presented an indexing scheme based on n-grams for `ssdeep` [104].

Roughly speaking, F2S2 initializes a hash table that allows to insert n-grams of the Base64 similarity digest. Each similarity digest is split into its n-grams and the ID to the corresponding file is put into its corresponding hash table bucket. In order to look for a similarity digest, the queried digest is split into its n-grams. Next, the index is used to find all candidates, i.e., the tool returns all similarity digests that contain at least one n-gram from the query. The final decision is then made by using the `ssdeep` comparison function.

The authors propagate an improvement of a factor of over 2000 which is 'practical speed'. For instance, they decrease the time for verifying 195,186 files against a database with 8,334,077 entries from 442 h to 13 min.

3.1.3. Security analysis

A security analysis concludes that CTPH is reasonable for detecting similar files, if no anti-forensic measures are present, however, it does not withstand an active adversary. In [4, 9] we give a proof of concept of how to circumvent a blacklist / whitelist which is demonstrated for the file types TXT, JPG, BMP and PDF.

In general we present two ideas: *editing between trigger sequences* and *adding trigger sequences*. The former one is mainly for TXT and BMP files which allow small changes all over the file, e.g., changing a letter from lowercase to uppercase. The attack simply changes one byte within each chunk and thus all chunk hashes change. This attack can be optimized so that an attacker only needs to change one byte in every 7-th chunk as two similarity digest must have a common substring of 7 characters.

The second attack uses the possibility of pre-computed trigger sequences which are independent of the file at hand. Table 3.1 shows a sample set of twelve global trigger sequences, which serve as triggers for any file of size $\leq$ 15 MiB.

Table 3.1.: Sample pre-computed global trigger sequences and their corresponding Base64 characters [4].

Trigger sequence	Base64 character	Trigger sequence	Base64 character
`AAAD?Hp`	9	`AAAV?Hf`	l
`AAAD?Og`	v	`AAAf?Ft`	p
`AAAD?QI`	7	`AAAr?xj`	V
`AAAJ?MW`	P	`AAAx?Fj`	1
`AAAJ?PJ`	F	`AAAx?OC`	n
`AAAJ?VO`	Z	`AAAx?tx`	5

The generation of these global trigger sequences is very efficient. Once these sequences are generated, they only need to be inserted at the beginning of a file. For instance, PDF and JPG allow to insert header information like author name which can be misused. If we could insert multiple trigger sequences in a row, we can create any similarity digest.

3.2. Multi-resolution similarity hashing (`mrsh`)

Multi-resolution similarity hashing (abbreviated `mrsh`) was presented in 2007 by Roussev et al. and is a powerful variation of CTPH [85]. The authors made four major changes:

Rolling hash. While the original implementation uses a rolling hash which is a variation of the Adler-32 checksum, the new version uses the polynomial hash function `djb2` which is defined as follows:

$$h_{-1} = 5381; \quad h_i = 33\, h_{i-1} + c_i; \quad \text{for } i \geq 0$$

where c_i denotes the i-th character of the byte sequence. The decision is based on a detailed comparison between MD5 and `djb2` which shows that the latter one is completely sufficient.

Chunk hash function. Instead of using FNV, the authors use MD5 for hashing the chunks as “FNV is not a collision-resistant function and has some known collision issues, which are common among multiplicative functions”.

Similarity digest representation. As described in Sec. 3.1, `ssdeep` outputs a nearly fixed size Base64 string, which can be compared using the edit distance. In contrast, `mrsh` generates a variable sized similarity digest and adapts a technique from *md5bloom* [84] that uses Bloom filters to represent the MD5 chunk hashes.

The motivation for Bloom filters is the better compression and the faster comparison therefore they accept a higher false positive rate. For instance, “consider a 256 byte Bloom filter versus a 256 byte LSB6 hash. The filter can accommodate

256 elements at 8 bits per element with four hash functions and will have a false positive rate of 0.024. The LSB6 hash will have at most $^{256}/_{6} = 42$ elements (in reality, 32 with byte alignment) with a false positive rate of $^{1}/_{64} = 0.015$".

In order to insert a chunk hash into the Bloom filter, the MD5 chunk hash is split into four 32 bit sub hashes. Next, the sub hash is reduced to 11 bits. Finally, each sub hash sets one bit within the 2^{11} bits = 2048 bits = 256bytes Bloom filter. For instance, the sub hash $010\,1100\,0011_2 = 707_{10}$ set bit 707 to 1. After inserting 256 chunks, the Bloom filter reaches its maximum and a new Bloom filter is created. Therefore, the final similarity digest is a sequence of Bloom filters.

Similarity digest comparison. While `ssdeep` uses the weighted edit distance to compare two Base64 sequences, this approach has to compare sequences of Bloom filters. Let $SD_1 = \{bf_1, bf_2, \dots bf_s\}$ and $SD_2 = \{bf'_1, bf'_2, \dots bf'_r\}$ the similarity digests of two inputs and $s \leq r$. Furthermore, let $z(SD_1, SD_2)$ be the similarity score with $0 \leq z \leq 1$ and

$$\begin{aligned}
z_1 &= \max\{z(bf_1, bf'_1), z(bf_1, bf'_2), \dots, z(bf_1, bf'_r)\} \\
z_2 &= \max\{z(bf_2, bf'_1), z(bf_2, bf'_2), \dots, z(bf_2, bf'_r)\} \\
&\dots \\
z_s &= \max\{z(bf_s, bf'_1), z(bf_s, bf'_2), \dots, z(bf_s, bf'_r)\}
\end{aligned}$$

then $z(SD_1, SD_2) = (z_1 + z_2 + \cdots + z_s)/s$. The authors claim the following: "We refer to the similarity between two filters as a Z-score, since it is derived from counting the number of zero bits in the filters and their inner product. It can be shown [...] that the magnitude of the intersection of the original sets from which the filters are derived is proportional to the logarithm of $Z = (Z_1 + Z_2 - Z_{12})/(Z_1 Z_2)$, where Z_1 is the number of zero bits in the first filter, Z_2 - in the second filter, and Z_{12} in their inner product (bitwise AND). The expected minimum is reached for $Z_1 = Z_2$ and $Z_{12} = 0$ (no common elements) so $Z_{min} = 1/Z_2 = 1/2048$, for our specific case of 256 bytes. Similarly, maximum is achieved whenever the two filters are identical so $Z_1 = Z_2 = Z_{12} \rightarrow Z_{max} = 1/Z_1$. Thus, we can map the interval $[\log(1/2048), \log(1/Z_1)]$ interval to $[0, 1]$ to obtain a score"[85, p6].

The overall proceeding is then like for `ssdeep`. `djb2` is used to split a file into chunks which are hashed using the new chunk hash function MD5. Instead of calculating a trigger value, `mrsh` uses several fixed trigger values, 8, 12, 16, 20, 24, 28, and 32. Thus, multiple levels / *resolutions* are created for larger files which allows an efficient comparison.

3.3. Similarity digest hashing (`sdhash`)

In the following we summarize the original implementation of `sdhash` as proposed by Roussev in 2010 [80, 83]. As it is ongoing project, there were several improvements in recent years which are explained at the end.

Each input can be denoted by a byte sequence $B_0, B_1 \dots B_{L-1}$ of length L. Then a feature f_k is a sub byte sequence of length $l = 64$ starting at B_k with $0 \leq k \leq L - l$:

$$\begin{aligned} f_0 &= B_0, B_1 \dots B_{l-1} \\ f_1 &= B_1, B_2 \dots B_l \\ &\dots \\ f_{L-l} &= B_{L-l}, B_{L-l+1} \dots B_{L-1} \end{aligned}$$

For every feature f_k the following two steps are required:

- Firstly, the normalized Shannon entropy score H_{norm} is calculated on base of the empirical entropy H of f_k

  $$H = -\sum_{i=0}^{255} P(X_i) \cdot \log_2 \left(P(X_i) \right) \ , \tag{3.5}$$

  where $P(X_i)$ is the empirical probability (i.e., the relative frequency) of encountering byte value i in f_k. Then H is scaled to a value in the integer range $[0, 1000]$ using

  $$H_{norm} = \lfloor 1000 \cdot H / \log_2 l \rfloor \ . \tag{3.6}$$

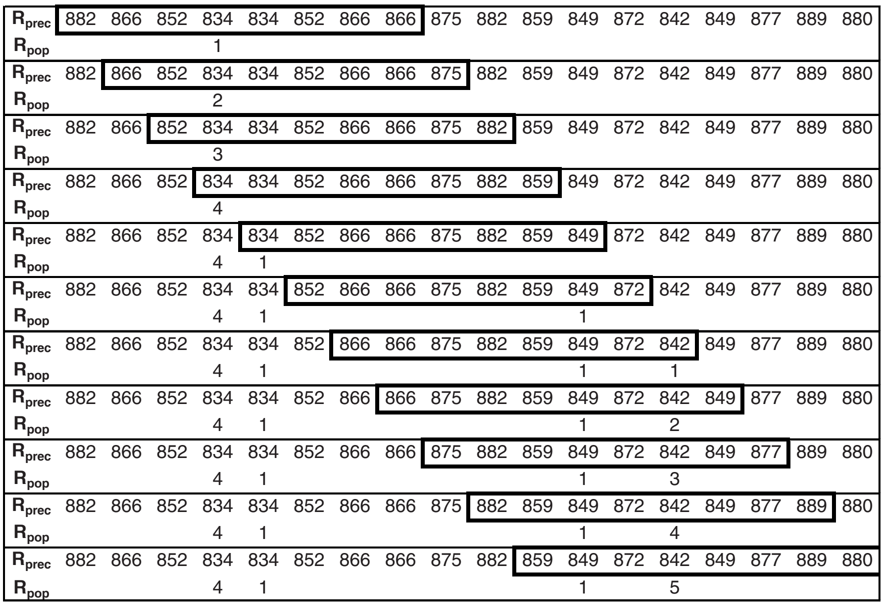
- Secondly, according to [79], "we associate a *precedence rank* [(abbreviated R_{prec})] with each entropy measure value that is proportional to the probability that it will be encountered. In other words the least likely feature is measured by its entropy score gets the lowest rank." The result is a sequence of R_{prec} values.

Next is the identification of the *popular* features, which is done using a sliding window Win of a fixed size W (`sdhash` uses $W = 64$) going through all R_{prec} values. At each position `sdhash` increments the R_{pop} score for the leftmost feature with the lowest R_{prec} within Win.

An example is given in Fig. 3.3 where the size of the window is set to $W = 8$. Let $R_{prec}(i)$ and $R_{pop}(i)$ denote the precedence and popularity rank of f_i, respectively. In Fig. 3.3 we have $R_{prec}(0) = 882$, $R_{prec}(1) = 866$, $\dots$. As $R_{prec}(3) = 834$ has the leftmost lowest R_{prec} within Win, $R_{pop}(3)$ is incremented and the window slides. Within the second iteration $R_{prec}(3)$ is still the leftmost lowest R_{prec} in Win and $R_{pop}(3)$ is incremented again, and so on. All features whose R_{pop} score are higher-equal than a given threshold (`sdhash` uses 16) are part of the fingerprint. We denote these features $F_0, F_1, \dots F_p$ (capital F).

As the threshold is 16, the minimum byte distance between neighboring features F_i and F_{i+1} is 16. For instance, let E be the last element within the window and also having the lowest R_{prec}. As E is the last element, the R_{pop} could be at most one. When sliding the window, there are two possibilities: The R_{prec} of the new element

1. is higher-equal, then R_{pop} of E is increased or



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 3.3.: Example for the R_{pop} calculation from [80].

2. is lower, then the R_{pop} of the new element is increased.

A more general argumentation shows that if $R_{pop}(i) = k$ $(1 \leq k \leq 64)$, then $R_{pop}(i+n) \leq n$ for all $1 \leq n < k$.

In order to generate the similarity digest, the byte string of each corresponding feature $F_0, F_1, \ldots F_p$ is hashed using SHA-1 and the resulting 160 bit hash value is split into five sub hashes of 32 bit length. As Roussev's Bloom filters consist of 256 bytes = 2048 bits = 2^{11} bits, he uses 11 bits from each sub hash to set the corresponding bit in the Bloom filter.

Roussev decided on a maximum of 128 features per Bloom filter, which results in a maximum of 128 features $\cdot$ 5 bits/feature = 640 bits per Bloom filter. If a Bloom filter is full, a new Bloom filter is created and added to the final similarity digest.

To define the similarity of two Bloom filters, we have to do some calculations about the minimum and maximum overlapping bits by chance whereas Roussev introduces a cutoff point C. Let $|bf|$ denote the number of bits set to one within a Bloom filter. If $|bf \cap bf'| \leq C$, then the similarity score is set to zero.

C is determined as follows

$$C = \alpha \cdot (E_{max} - E_{min}) + E_{min} \tag{3.7}$$

where α is set to 0.3, E_{min} is the minimum number of overlapping bits due to chance and E_{max} the maximum number of possible overlapping bits. Thus, E_{max} is defined as

$$E_{max} = \min(|bf|, |bf'|). \tag{3.8}$$

Let k be the number of sub hashes ($= 5$ within `sdhash`), $\overline{bf}$ the number of features within a Bloom filter, m the size of a Bloom filter in bits ($= 2048$) and $p = 1 - {}^{1}\!/_{m}$ the probability that a certain bit is not set to one when inserting a bit. Thus,

$$E_{min} = m \cdot (1 - p^{k \cdot \overline{bf}} - p^{k \cdot \overline{bf'}} + p^{k \cdot (\overline{bf} + \overline{bf'})}) \tag{3.9}$$

is the expected number of common bits set to one in the two Bloom filters bf, bf' under the assumption that they do not have a common feature.

Let $SD_1 = \{bf_1, bf_2, \dots bf_s\}$ and $SD_2 = \{bf'_1, bf'_2, \dots bf'_r\}$ the similarity digests of two inputs and $s \leq r$. If $\overline{bf_1} < 6$ or $\overline{bf'_1} < 6$ then the original input does not contain enough features and so the similarity score is -1, i.e., not comparable. Otherwise the similarity score is the mean value of the best matches of an all-against-all comparison of the Bloom filters, formally defined as

$$SD_{score}(SD_1, SD_2) = \frac{1}{s} \sum_{i=1}^{s} \max_{1 \leq j \leq r} SF_{score}(bf_i, bf'_j) \tag{3.10}$$

where SF_{score} is the similarity score of two Bloom filters

$$SF_{score}(bf, bf') = \begin{cases} 0, & \text{if } e \leq C \\ [100 \cdot {}^{(e - C)}\!/_{(E_{max} - C)}], & \text{otherwise} \end{cases} \tag{3.11}$$

with $e = |bf \cap bf'|$.

3.3.1. Extensions and improvements

`sdhash` is currently available in version 3.3 and thus we summarize the main changes, extensions and improvements in the following.

Primarily, approximate matching was designed to do a file against file comparison which is a sequential process. However, in 2012 the authors came up with a *parallelized version* of `sdhash` which uses block-aligned similarity digests [83]. Instead of treating the target as one single piece, it is divided into fixed blocks first with a default size of 16 KiB. Each block is hashed separately and put in exactly one Bloom filter. The amount of features per Bloom filter increased from 128 to 192. As a block could have more than 192 features, the authors choose the features that have the highest popularity score. In literature, this new version is referred as `sdhash-dd` whereby the original/sequential version is still called `sdhash`. Here, the number of inserted features increased from 128 to 160 which improves the compression ratio.

To speed up the comparison of the similarity digests, Roussev introduces a `-s` option called sampling. In sampling, the idea is to pick a few filters from the queried similarity digest and look for those only. This is useful when looking for a whole file inside a disk/RAM image. If the whole file is there, then we could use a small piece of it to find it (note: for every comparison, only the first argument is sampled). This can speed up the scan of a target tremendously. Allowed range for `-s` is 1-16, recommended value is 4; by default it is set to zero (no sampling).

A second extension is the segmentation mode. This just sets the size of the input that the algorithm processes at once (i.e., makes a single digest out of). For example, if a 512 MiB file serves as input and the segment size is set to 128 MiB, `sdhash` outputs 4 similarity digests. Hence, it acts as one passed it 4 separate files, one for each 128 MiB chunk. The similarity digest will include information regarding which chunk they were generated from.

3.3.2. Comparison and evaluation of `ssdeep` and `sdhash`

A comprehensive evaluation of `ssdeep` and `sdhash` was done by Roussev in 2011 [81]. In his paper he tries to find out what “kind of byte-level correlations do these tools actually detect [and] how do detected correlations relate to human-perceived correlations between the same artifacts?”. The evaluation is two-tier and presents a controlled study based on pseudo-random data and a manually evaluated study on real world data.

Random data. For the evaluation, Roussev analyzed the behavior of the algorithms for the following three scenarios:

Embedded object detection measures if the tool can correlate files and arbitrary blobs. For example, assuming a JGP file and a Word document, then the JPG could be embedded. Another scenario is a memory dump and a JPG.

Single-common-block file correlation “simulates a situation where two files have a single common object. [...] For example, documents sharing an image, or pieces of software sharing library code.”

Multiple-common-blocks file correlation is similar to single-common-block but now the “commonality might be significant but fragmented. [...] Generally, this is a more difficult task and demands more precision from the tools.”

For all tests, `sdhash` showed vastly better results. For instance, given an object of 256 KiB, `ssdeep` needs a minimum single-common-block of 80 KiB while `sdhash` is satisfied with a 16 KiB block. The results show a similar behavior for all test scenarios.

Real world data. This test is based on the t5-corpus and is done manually based on the following idea. As the tools work on “syntactic commonality [..., hence] neither tool performs any kind of semantic analysis; therefore, any correlation needs to be visible rather quickly. For web pages, this almost always means a common template (header/footer/navigation); for office documents, and across different file types it means common images, or large blobs of common text.”

The manual inspection of 1699 files showed that `ssdeep` has less true positives but more false positives compared to `sdhash`. The main conclusion for both tests is, that `sdhash` “significantly outperforms in terms of recall and precision in all tested scenarios and demonstrates robust and scalable behavior.”

3.4. SimHash

SimHash was presented in 2007 by Sadowski & Levin [87] and is another tool for "hash-based similarity detection". The following presents a brief summary of the algorithm, however, we give all of the details as SimHash is "focused on files which have a strong degree of similarity" [87].

SimHash consists of a feature extraction function and a similarity function. For hashing, the authors define a set of 16 8-bit strings, called *tags*, which are randomly chosen (`0x00` is excluded). Next, an input is processed bit for bit, the occurrences of the tags are counted and stored in a sum table. Based on the sum table, SimHash computes a hash key which is a linear combination of the sums. This is needed for indexing/ordering the hashes in a database. "Once this has all been computed, the file name, path, and size, along with its key and all entries in the sum table, are stored in a MySQL database". Besides this original implementation, the authors tried a second key function which considers the file extensions. The motivation is that "it is not unreasonable to claim that two files are inherently different if they contain different extensions".

To find entries in the database, SimFind is used to search for similar keys. First, a single tolerance level is set, which is multiplied by the file size as key values are expected to increase proportionally to file size. For all returned values, SimFind verifies that the file size does not differ too much. The last step of the comparison calculates the distance of the sum tables "which is the sum of the absolute values of the differences between their entries. If this distance is within a certain tolerance, then we report the two files as similar."

Compared to other existing approaches, SimHash works for near duplicates only as it expects a very similar file size. The variation of the key function to consider the file extension narrows the effectiveness even more. To conclude, SimHash could only be used for near duplicates of the same type. Hence, fragment detection or embedded objects (e.g., a JPG in a office document) are not identified.

3.5. Summary

This chapter gave an overview of existing approximate matching algorithms. It pointed out that SimHash and MinHash are only usable for very similar files (i.e., no fragment detection is possible) and hence they are excluded from further consideration. Besides those, there are three more approaches.

`ssdeep` is probably the most popular one and was studied by many different researchers. An important extension for this algorithm was F2S2. Now it is possible to index the similarity digests which brings `ssdeep` to practical speed when doing a forensic investigation. However, a security analysis demonstrated different exploits.

The second algorithm called `sdhash` is way more secure and ongoing work. A detailed comparison between `ssdeep` and `sdhash` showed that `sdhash` outperforms `ssdeep` with respect to precision and recall.

The last tool was `mrsh` which actually was never really studied or compared against other approaches.

4

Assessment & improvement of existing approximate matching approaches

The previous chapter described existing algorithms and implementations which are studied in more detail here. In the case of `ssdeep` we contribute with an improvement of generation efficiency of 55% by changing the overall proceeding of the implementation. The results were published in [13]. Similar to the security analysis for `ssdeep` from the last chapter, we analyzed `sdhash`. Our contribution discusses several bugs which yield to an unexpected behavior. Additionally, we present some weaknesses that allow uncovered changes and attacks. We published these results in [16, 14]. The last section focuses on `mrsh`. We extended the algorithms which increased the overall performance. The result is `mrsh-v2` which has a better generation efficiency and compression. With respect to similarity detection, it has a similar behavior than `sdhash`. The results are published in [15].

4.1. Context-triggered piecewise hashing

The upcoming paragraphs discuss multiple performance improvements of context-triggered piecewise hashing (CTPH) which were published in [13]. The contribution besides the detailed algorithm and design analysis, was a runtime improvement of 55%. In addition, we also present a way to identify manipulated files, which could be the act of an active adversary.

4.1.1. Enhancements overview

Due to the design of CTPH, it is possible that `ssdeep` needs to process an input several times. More precisely, if there are too few Base64 characters for block size bs_{init} (i.e., at most $S/2 - 1 = 64/2 - 1 = 31$), Kornblum sets $bs \leftarrow bs_{init}/2$ and the whole process is repeated.

Our improvement aims at three main points:

1. Each file should be processed only once and thus we have a runtime directly proportional to the input length.
2. The implementation should be flexible so that we can change the pseudo random function (PRF) and chunk hash function h.
3. It should be able to determine an untypical behavior of trigger sequences, which may be caused by an active adversary.

The main idea is to process an input and count the trigger sequences for all reasonable block sizes (the term *reasonable* is explained later). In the next step, we read the file again and set the block size bs to the largest value that yields at least 32 Base64 characters.

An important point with respect to security is the restriction of the similarity digest length. Kornblum asserts $32 \leq length \leq 64$. However, he does not give any justification. We deem the lower boundary to be useful in order to be able to make statements about similarity. The upper boundary is a weakness and was exploited in [4]. Maybe a reason to set this boundary was to satisfy the compression conditions for hash functions. Nevertheless, some attacks are not possible if we ignore this condition.

Implementation details. We use the original software for our improvement and insert our ideas. This means that all shown performance improvements are only based on some algorithm changes and not on some implementation issues.

We consider a reasonable block size to be of order of magnitude of Kornblum's proposal for bs_{init} and thus, we allow $bs \in \{2\,bs_{init}, bs_{init}, {}^{bs_{init}}/2, {}^{bs_{init}}/4\}$. The file is read byte for byte and the PRF is computed for each byte. If we found a trigger sequence for one of the four reasonable block sizes, we save the offset and increase a counter for this block size. As a reminder, a trigger sequence is found, if $PRF(BS) \equiv -1 \mod bs$. In most of the cases we only have to check one if-condition as BS can only be a trigger sequence for bs if it is a trigger sequence for ${}^{bs}/2$, too.

For the second run, the file is read again byte for byte. Now we use the stored offsets to determine each chunk and run the chunk hash function h. As a result, we preserve the flexibility to change PRF and the hash function.

Untypical behavior of trigger sequences. We demonstrated that an active adversary can manipulate a file and bypass blacklisting and whitelisting [4]. For this, we exploited a peculiarity that a similarity digest can have at most 64 characters. The attack randomly generates trigger sequences and inserts them at the beginning of the file.

Generally, we would expect that no file has significantly more than 64 trigger sequences for their initial block size b_{init}. In general there are two extreme examples:

- Locally non-sensitive files: They are expected to have long runs of a specific byte, e.g., 0-byte-sequences. As this will not cause a triggering, there are too few trigger sequences instead of too much. Example file types are DOC or BMP.

- Locally sensitive files: They are expected to have very variable byte-sequence. A well-tested PRF is assumed to produce approximately 64 trigger sequences. Example file types are JPG, ZIP or a truecrypt container.

A possible manipulation can be detected by comparing the trigger sequences counter against a certain threshold. Of course, this mechanism could be improved by considering the distribution of the trigger sequences, i.e., we would expect that all chunks have a similar length.

Assessment of our improvement. As it is described above, the modification needs to read the file two times: one run for receiving the amount of trigger sequences including their offsets and one run for hashing each junk. The amount of computations, i.e., building the PRF and FNV-Hash, is similar in both `ssdeep` versions and hence there should not be a significant difference. The main disadvantage is that our proposal reads the file two times from the hard disk / cache / RAM.

On the other side, the new algorithm is superior if there are not enough trigger sequences. Thus, the original version needs to do two complete runs, which means: reading the file from the hard disk, generate the PRF, and compute chunk hash. If this is the case, we expect the new version to be faster. As then both algorithms need to read the file from the hard disk two times, we expect to have an improvement over 50%.

If the file has even less trigger sequences (e.g., the block size needs to be quartered) and therefore needs to be read more often, we expect a great performance difference between both algorithms: $bs \leftarrow {}^{bs_{init}}/_{4}$ results in 33% runtime, $bs \leftarrow {}^{bs_{init}}/_{8}$ in 25% runtime.

Thus, a new question raises up, which will be answered in the next section: *How often does `ssdeep` on average adapt the block size?*

4.1.2. Experimental results

This section shows some experimental results of our enhanced version. Firstly, we present the practical relevance of our enhancement on base of a 500 MiB file and secondly, we discuss the performance advantage with respect to a real-life scenario.

Efficiency improvement. In the following we make a practical test to verify our improvement. We compare the original `ssdeep` version to a modified copy for three different 500 MiB files where

File1 has enough trigger sequences for bs_{init},

File2 has only enough trigger sequences for ${}^{bs_{init}}/_{2}$, but not for bs_{init}, and

File3 has only enough trigger sequences for ${}^{bs_{init}}/_{4}$, but for no larger block size.

Hence, file2 and file3 are processed 2 and 3 times, respectively.

The results are given in Table 4.1 where the time is measured using the Linux `time`-command. *Sum* denotes the total time. The user-time results essentially from processing

Table 4.1.: CPU time to process different 500 MiB files with `ssdeep` and our modified improved version.

	File1	File2	File3
org. `ssdeep`	user-time: 2.76	user-time: 5.45	user-time: 7.53
	sys-time: 4.89	sys-time: 10.06	sys-time: 15.47
	sum: 7.65	sum: 15.51	sum: 23.00
mod. `ssdeep`	user-time: 1.50	user-time: 1.46	user-time: 1.74
	sys-time: 7.60	sys-time: 7.58	sys-time: 7.52
	sum: 9.10	sum: 9.04	sum: 9.26
$^{\text{mod. sum}}/_{\text{org. sum}}$	1.19	0.58	0.40
estimated $^{\text{mod. sum}}/_{\text{org. sum}}$	1.00	0.50	0.33

the input, e.g., generating the PRF and the FNV-Hash. In contrast, the sys-time is mostly influenced by reading/buffering the file.

As expected, we have a constant user-time and a higher sys-time for the modified version. Our version needs to do less computations, i.e., we do not generate all trigger sequences and have less if-conditions to check. Having a look at the results from the original version, we recognize that there is a linear increase; a halving block size increases the runtime by approximately one run.

To conclude, the modified version has a constant runtime, but is approximately 20% slower compared to the original version if there is only one run. In addition, we expected an improvement of 50% if the block size $bs \leftarrow {}^{bs_{init}}/_{2}$ is used; the practical relation is 58% in our test. Overall we can say that the improvement depends on the files we investigate.

Impact on real world data. In order to receive trustful results which mirror a real-life scenario, we set up a system running Windows XP Service Pack 3 including some basic applications and user specific files. We assume that nowadays nearly every personal computer has at least an office suite, a browser and a PDF-Viewer installed; we installed OpenOffice 3.3, Mozilla Firefox 4.01 and Acrobat Reader 10.0.1. Additionally, we inserted

- 1,000 images having a size from a few KiB to 1 MiB,
- 20 MP3 audio files having a size from 4 to 11 MiB, and
- several free available PDF files.

Even though this only reflects a very small system, it allows estimations what happens if a hard disc image would have about 200 GiB or more.

Our sample image comprises 15,036 files and 3,936,437,740 bytes, which leads to an average file size of

$$^{3,936,437,740\ \text{bytes}}/_{15,036\ \text{files}} = 261,800.86\ ^{\text{bytes}}/_{\text{file}} = 255.7\ ^{\text{KiB}}/_{\text{file}}\ .$$

The adjustment of the block size is the critical point. If an image contains many files where the block size changes, the performance of our modified version is better. As already investigated by [32], we expect that there is a change of the block size within the `ssdeep` processing in 38% of the cases. The results of our sample test environment of 15, 036 files are given in Table 4.2.

Table 4.2.: Distribution of block size changes.

1 time	2 times	3 times	4 times	5 times	$\geq$ 6 times
10,125	3,944	645	176	32	114
67.3%	26.2%	4.3%	1.3%	0.2%	0.8%

10, 125 files only need to be processed once, which means that approximately 33% needs to be processed more than once. This is in conformance with the claim given in [32].

However, if we examine the relationship between the file size and the amount of trigger sequences, we find an interesting relationship. The files, which only need one run (i.e., each of these files has enough trigger sequences) have a total size of 1, 302, 435, 802 bytes which is an average file size of 127 KiB. On the other hand, all the rest has a total size of 2, 634, 001, 938 bytes which is an average file size of 535 KiB and thus about 4 times larger than one-processed files. In general we can say that mostly large files need more than one run. One possible answer would be that large files, with some exceptions, often contain long 0-byte-sequences, which do not trigger the PRF.

Next, we concentrate on a real-world scenario and compare the runtimes of both `ssdeep` versions. Additionally, the runtime for SHA-1 is measured and serves as a benchmark. The results are given in Table 4.3. We performed two runs that we can exclude caching issues.

Comparing the runtime of our modified `ssdeep` version to the original one yields an improvement of approximately 55% for the complete image (because the runtime of the modified version is only about 45% of the original version). The large improvement can be explained by the 2.6 GB that needs to be processed more than one time. However, compared to the cryptographic hash function SHA-1, both implementations are severely slower.

4.2. Similarity digest hashing analysis

This section is a detailed analysis of `sdhash` that is split into two parts. Firstly, we made a detailed implementation analysis which was published in [16]. The contribution was to prove if the current implementation (version 1.2) coincides the specification. The main result was that there are a couple of implementation and design errors that lead to an untypical behavior of the algorithm. All findings were sent to the authors and thus should be fixed in the current version 3.3. However, we did not conduct a second analysis

Table 4.3.: Runtimes (in seconds) to compute similarity digest of all files on our sample image.

	first run	second run
org. `ssdeep`	user-time: 202.96	user-time: 201.94
	sys-time: 52.01	sys-time: 56.02
	sum: 254.97	sum: 257.96
mod. `ssdeep`	user-time: 88.58	user-time: 86.04
	sys-time: 26.29	sys-time: 29.19
	sum: 114.87	sum: 117.23
mod. sum/org. sum	0.45	0.45
sha1sum	user-time: 26.98	user-time: 26.57
	sys-time: 12.96	sys-time: 13.46
	sum: 39.94	sum: 40.04

to verify the fixing. Some details are given in Sec. 4.2.1.

The second part compares the behavior of `sdhash` against self-defined properties which was presented in [14]: compression, runtime efficiency, coverage and similarity score. Besides comparing against properties, our security analysis identified some weakness of the algorithm. More details are given from Sec. 4.2.2 to Sec. 4.2.5.

4.2.1. Implementation

In the following we briefly summarize our findings. A detailed description of the bugs is given in Appendix A. We discovered two important bugs when computing the popularity rank R_{pop}:

1. The *window size bug* is a typical off-by-one error concerning the window size used to compute R_{pop}. Thus, the implementation does not correctly identify the minimal R_{prec} value in the current window and does not always select the 'statistically improbable feature' as defined by the specification.
2. The *leftmost bug* means that the implementation does not necessarily uses the leftmost feature as described in the specification. The impact of this second bug is low but we argue that the specification and the implementation should coincide.

In addition, Appendix A.2 shows three bugs in the comparison class.

4.2.2. Compression & runtime efficiency

As stated by the authors [80], the similarity digest size of `sdhash` is proportional to the input size and has a compression rate of approximately 2.6%. The basis for this calculation are six sample 100 MiB document sets from the NPS Corpus [34] containing DOC, HTML, JPG, PDF, TXT, XLS and a 100 MB file from `/dev/urandom`. In order to validate this result, we used the t5-corpus, processed all files and compared the similarity digest length to the original file size. As shown in Table 4.4 line 17, we obtained a compression rate of approximately 3.3%.

Regarding the runtime, the algorithm of `sdhash` is by design more complex than `ssdeep`. For instance, while `ssdeep` divides an input into chunks which are hashed with FNV, `sdhash` identifies statistically improbable features that are hashed with SHA-1. Although SHA-1 is optimized for performance, it is slower than non-cryptographic hash functions or cryptographic hash functions like MD5. Thus, we identified two possibilities to increase the performance:

1. In the case that preimage resistance is important, one could change the feature hash function to MD5 as it is faster [8]. The security benefits of SHA-1 can be neglected, as `sdhash` only relies on 55 bits out of the 160 bit SHA-1 hash.

2. In the case that preimage resistance is not a prerequisite one could use FNV for feature hashing or any other non-cryptographic hash function. It is obvious that this simple hash function containing one multiplication and a XOR outperforms SHA-1 and MD5.

4.2.3. Coverage

In general hash functions are designed so that each bit of the input influences the hash value which we denote by *full coverage*. Otherwise it might be possible that a modification is not discovered. Therefore, we present two major drawbacks of `sdhash` that allow to change up to 20% of an input with an unaltered similarity digest.

Gaps and overlaps. In the following we show that only approximately 80% of an input is considered within the similarity digest although it is theoretically possible to have full coverage. This raises the question if the parameters for the window size and the popularity rank threshold are chosen suitably.

A core result of our work are the statistics given in Table 4.4, which show the *average measurements* for all files within the t5-corpus (the min/max values are no absolute values, but averaged). Besides the original version we also built the statistics for an *modified version* of `sdhash`, where we fixed the bugs from Sec. 4.2.1.

The first block describes the statistics for gaps where row 2 is the amount of gaps followed by the minimum, maximum and average gap in bytes. Row 6 gives the ratio between the gap and the file size. The next block (rows 7-11) is constructed identically but with respect to overlaps. Details about the features are given in the last block (rows 12-16). Row 13 describes the ratio between features and file size *without* overlappings. To conclude, there are two core statements:

1. The first two blocks (rows 2-11) show that many features are overlapping, resulting in wide gaps between two non-overlapping consecutive features.

2. The last block (rows 12-16) describes the impact of the overlappings. In the case of wanting to manipulate each feature (e.g., to achieve a non-match) we do not need to change all *inserted features* (row 12) but only 60% of it due to the overlappings.

Table 4.4.: Different statistics on `sdhash` using the t5-corpus.

	average...	**modified**	**original**
1.	file size*	428,912	428,912
2.	gaps count	2888	2889
3.	min_gap*	1.090	1.076
4.	max_gap*	1834	1834
5.	avg_gap*	33.46	34.27
6.	ratio to file size	20.65%	21.21%
7.	overlap count	4387	4402
8.	min_lap*	1.110	1.108
9.	max_lap*	47.81	62.50
10.	avg_lap*	22.53	22.71
11.	ratio to file size	21.41%	21.86%
12.	inserted features	6923	6937
13.	ratio to file size	58.47%	57.45%
14.	all features	7276	7292
15.	required mod.	4248	4214
16.	ratio mods.	60.14%	59.48%
17.	SDsize % of file	3.321%	3.344%

* values are given in amount of bytes.

In the following we analyze the coverage of `sdhash`. Let L denote the length in bytes of an input and s the size of all Bloom filters as percentage in relation to L. Thus, the size of all Bloom filters (SD, the similarity digest) can be estimated by $SD_{size} = {}^{L \cdot s}/_{100}$. As each Bloom filter consists of 256 bytes, the number of Bloom filters is $SD_{amount} = \lceil {}^{SD_{size}}/_{256} \rceil$. Furthermore, each Bloom filter except the last one contains 128 features and therefore can represent at most $128 \cdot 64$ bytes $= 8192$ bytes of the input. An upper bound of *coverage* (cv) (influencing bytes) can be estimated by

$$\begin{aligned} cv &= SD_{amount} \cdot 8192 \\ &= {}^{L \cdot s}/_{100 \cdot 256} \cdot 8192 \\ &= 0.3200 \cdot s \cdot L. \end{aligned}$$

As we aim at a coverage of 100% we equate cv with L.

$$\begin{aligned} L &= 0.3200 \cdot s \cdot L \\ s &= 3.125\ . \end{aligned}$$

In order to achieve a full coverage, a similarity digest length of 3.125% is sufficient. However, although `sdhash` has approximately 3.3%, it does not have full coverage. The coverage can be calculated by adding up rows 11 and 13 of Table 4.4. Thus, the modified version results in 79.88% and the original one in 79.07% which also coincides with the gaps from row 6. Due to the averaging the percentage values are not exactly 100%.

Although it is questionable if all bytes need to influence the similarity digest, `sdhash` statistically allows to change approximately 20% of an input and the similarity score its still maximal. On the other hand, due to the overlapping of features one changed byte of the input changes multiple features.

Unnoted Footer Changes. This paragraph shows the possibility to have two inputs that differ by 11% but still result in the highest similarity score of 100. As this issue is not addressed within the specification [80], it was found during the code review and is based on appending, deleting or modifying the end of an input.

Let r denote the amount of Bloom filters of a similarity digest. Based on [80] there is only one restriction: If $r = 1$ and $\overline{bf_r} < 6$, the generation process stops and prints an error message.

In fact, this is different to the actual implementation where a second condition is present: If $r \geq 2$ and $\overline{bf_r} < 16$ then bf_r is skipped.

Due to this behavior it is possible to append or delete data at the end of a file. For instance, if a file has exactly 128 features, we can append data containing up to 15 features or the other way round we can cut off features if the input has between 129 and 143 features. In both cases `sdhash` outputs the highest similarity score of 100.

The average byte length of an input containing exactly 15 features can be estimated using Table 4.4. In average a file has 428, 912 bytes and contains 7292 features. Thus, a byte sequence containing 15 features has a medial length of approximately $^{428,912 \cdot 15}/_{7292} = 882.29$ bytes.

To conclude, if we have an input containing exactly 128 features which results in a length of approximately $^{428,912 \cdot 128}/_{7292} = 7528.90$ bytes then we can append 882.29 bytes (which is over 11%) and `sdhash` outputs the highest similarity score. It is questionable if the highest match score is acceptable if a file changes by 11%.

Unnoted byte changes. Besides overlaps and gaps, we also discovered a minor issue concerning the first and last 15 bytes of a byte sequence [16]. By design the first and last 15 bytes will never influence the similarity digest as there have to be 16 slides of the window to obtain a popularity score above 16. We rate this as a minor weakness due to the following two reasons:

- Besides text files, most file types do not allow to change the header or footer information.
- After changing 30 bytes both files are quite similar for files of practical interest. Nevertheless it is a drawback of `sdhash` that it outputs the highest match score even if the first and final 15 bytes are modified.

An easy way to resolve this weakness is to create a cryptographic hash value over the whole input, treat it as a ‘feature’ and insert it as first element into a Bloom filter.

4.2.4. Similarity score

`sdhash` is promoted in two ways. First, it can be used to identify similar files and second it can be used to find small pieces of an original file (fragments). Both issues are very important in the area of computer forensics but `sdhash` does not distinguish between these two cases. For instance, assuming a file containing 128 features and a fragment of this file with 64 features, `sdhash` outputs the maximum score of 100. This might be desired for fragment detection but with respect file similarity detection, one would assume a score round about 50. However, we agree that fragment detection has better detection rates and is the default mode. In the following we explain how the comparison algorithm could be adapted to have two modes.

The fragment mode uses the original comparison algorithm. Recall, the computation of the similarity score $SF_{score}(bf, bf') = [100\,{}^{(e-C)}/_{(E_{max}-C)}]$ and the definition of $E_{max} = \min(|bf|, |bf'|)$ (see Sec. 3.3 on page 26) where e denotes the amount of common bits in bf and bf'.

A full match score occurs, if a full Bloom filter is compared to a non-full Bloom filter, and if both Bloom filters are related to each other. More precisely, let bf be a full Bloom filter and bf' be a Bloom filter where we dropped x features compared to bf. Roussev scales the comparison score to the number of bits set in the non-full Bloom filter bf', i.e., we have $E_{max} = \min(|bf|, |bf'|) = |bf'|$. However, as every feature of bf' is also represented by bf, we have $e = |bf'|$, too. Thus, if $e > C$, the similarity score is 100 although bf and bf' are obviously different. Hence, the comparison algorithm yields the highest match score by design.

In order to obtain file similarity detection, we recommend to change the min-function in SF_{score} into a max-function: $E_{max} = \max(|bf|, |bf'|)$. The cutoff point should still use the old variant of E_{max}. As an example we consider a non-full Bloom filter bf', where we ignore 64 features compared to bf. The number of expected bits set in bf' is $2048 \cdot \left(1 - [1 - {}^{1}/_{2048}]^{5 \cdot 64}\right) = 296.3$. As we now have $E_{max} = 549.8$ instead of 296.3, our proposal yields an overall similarity score of $100 \cdot {}^{(296.3 - 42.87)}/_{(549.8 - 42.87)} = 50.0$, which is a reasonable result for this setting.

To conclude, it is easily possible to extend the implementation to have two modes, one for finding fragments and one for finding similar files. If a user aims at finding fragments the original setting is perfect. Otherwise our suggestions should be considered.

4.2.5. Security analysis

Bloom filter resistance. The most obvious idea to obtain a non-match is to manipulate the features as the similarity digest is based on the hashed features. Due to the comparison mechanism, it is not necessary to modify all features. Hence, the following question arises: *How many features need to be changed in order to obtain a false negative?*

The average amount of bits set to one within a full Bloom filter $\overline{bf} = 128$ of size $m = 2048$ with $k = 5$ sub hash functions by chance is

$$\begin{aligned} E_{avg} &= m \cdot \left(1 - (1 - {}^{1}\!/\!_{m})^{k \cdot \overline{bf}}\right) \\ &= 2048 \cdot (1 - 0.99951172^{5 \cdot 128}) = 549.77. \end{aligned}$$

As both filters approximately contain 550 bits, E_{max} is equal to E_{avg}.

On the other side the minimum overlapping bits by chance is

$$\begin{aligned} E_{min} &= m \cdot (1 - p^{k \cdot \overline{bf}} - p^{k \cdot \overline{bf'}} + p^{k \cdot (\overline{bf} + \overline{bf'})}) \\ E_{min} &= 147.433. \end{aligned}$$

Based on this key data, the cutoff point is $C = 0.3 \cdot (549.77 - 147.43) + 147.43 = 268.13$.

After defining the underlying conditions, we estimate how many bits change when we manipulate one feature. The probability that one bit is set to zero in a full Bloom filter is $0.99951172^{5 \cdot 128} = 0.7315598$. Thus, the manipulation of one feature will approximately change $0.7315598 \cdot 5 = 3.65779898$ bits within a Bloom filter. We successfully verified this value through an empirical test where we used 10,000 random files having exactly 128 features, manipulated the first one and analyzed the amount of varying bits.

Let e be the bits in common. If $e \leq C$, the Bloom filters are treated as a non-match. Thus, we need to change $e - C$ bits. Two identical full Bloom filter have approximately 550 bits in common and a cutoff point C of 268 which result in $550 - 268 = 282$ bits. Due to the pseudo-randomness of SHA-1 we do a linear approximation. By changing one feature we approximately change 3.66 bits therefore we have to manipulate ${}^{282}\!/\!_{3.66} = 77.05$ features.

A test on real-world data showed that approximately 83.37 changes are necessary to receive a non-match which might be because of the reoccurring features. Thus, each feature changes ${}^{282}\!/\!_{83.37} = 3.383$ bits within a Bloom filter.

The average byte length of an input containing exactly 128 features can be estimated using Table 4.4 where we averaged a large file corpus. In average, a file has $428,912$ bytes and contains 7292 features. Thus, a byte sequence containing 15 features has a medial length of approximately ${}^{(428,912 \cdot 15)}\!/\!_{7292} = 882.29$ bytes.

From Sec. 4.2.3 we know that a lot features are overlapping which reduces the amount of needed changes. As a consequence, within each chunk of 7545.46 bytes we have to change approximately $83.37 \cdot 0.6 = 50.02$ bits. This results in a lot of changes all over the file which is only feasible for locally non-sensitive file types, e.g., BMP, TXT but hardly only for locally sensitive file types, e.g., JPG, PDF.

Bloom filter shifts. In the following we describe an easy way how to reduce the similarity of both digests down to approximately 25. Let SD, SD' be two identical similarity digests. A SD is comprised of $bf_0, bf_1, ...bf_s$ where

- bf_0 contains $F_0, F_1, \ldots F_{127}$,

- bf_1 contains $F_{128}, F_{129}, \dots F_{255}$,
- and so on.

The idea to diminish the similarity score is to build our own features which would be inserted at the beginning of an input. For instance, we build a feature F^*_{-1} and insert it. As a consequence

- bf_0 contains $F^*_{-1}, F_0, \dots F_{126}$,
- bf_1 contains $F_{127}, F_{128}, \dots F_{254}$
- and so on,

which will reduce the similarity score for all following bf. As all Bloom filters, except the last one, contain 128 features, we expect to achieve the lowest score by inserting 64 new features F^*.

The previous section analyzed the impact of changing one feature and concluded that approximately 3.383 bits change within the Bloom filter for real-world data. Thus, if we insert 64 new features in the beginning, this will approximately change $3.383 \cdot 64 = 216.51$ bits. The similarity score of two Bloom filters is $SF_{score} = 100 \cdot {}^{(e - C)}/_{(E_{max} - C)}$. Due to the feature insertion, e should reduce from 550 to $550 - 216 = 334$ and

$$\begin{aligned} SF_{score} &= 100 \cdot {}^{(e - C)}/_{(E_{max} - C)} \\ &= 100 \cdot {}^{(334 - 268)}/_{(550 - 268)} \\ &= 23.40 \ . \end{aligned}$$

In order to test our conclusion, we used our modified version of `sdhash` and did three tests:

1. We changed 64 features of 10,000 randomly generated files containing exactly 128 features and obtained an average similarity score of 24.61.
2. We inserted 64 features into cut files from the t5-corpus where 'cut' means we reduced to original files down to 128 features. The result was 21.34.
3. We inserted 64 features into each file of the t5-corpus which resulted in an average similarity score of 27.27.

Inserting byte sequences is often possible for a lot of file types. In [4] we demonstrated that for the locally sensitive file types like JPG and PDF. Regarding TXT or BMP, it is even more trivial.

An important question in this context is: *how much of an input do we have to change at least in order to reduce the similarity score to a minimum?* Theoretically it is possible to create a shortest byte sequence with $16 \cdot 63 + 64 = 1072$ bytes as there is a minimum distance of 16 between two features. However, in our test case we took the shortest sequence we found within the t5-corpus which has a length of 2765 bytes. Once we identified such a byte sequence, the time complexity of manipulation is $O(1)$ – this precomputed feature sequence is simply inserted in a file.

Since version 2.0, `sdhash` has a parallelized mode which splits an input in fixed blocks of l_b bytes. Therefore, the proceeding differs only by one detail. Instead of inserting all features above a given threshold, the features with the highest popularity score were used. Thus, an active adversary can insert/delete a byte sequence of $l_b/2$ in the beginning which shifts the offset of all blocks.

The problem due to these shifts is that we now do have two best matching Bloom filters. Thus, an idea to overcome this issue would be to use to change the matching algorithm and consider the two consecutive Bloom filters (if the similarity score of the first one is lower than a certain threshold).

4.3. Multi-resolution similarity hashing version 2 (`mrsh`)

`mrsh-v2` is an improvement of the original implementation from Sec. 3.2 and was published in [15]. The contribution was to create a new implementation that is based on the best features of the existing algorithms `ssdeep`, `sdhash` and `mrsh`. Therefore, we analyzed the different design decisions, combined them and received a very fast and robust approximate matching algorithm.

4.3.1. Pseudo random function

We propose two important requirements on a pseudo random function (PRF). First, its output should behave pseudo randomly and second, it has to be very efficient with respect to its runtime as it is invoked for roughly every byte of the input.

[85, Sec. 3] compares the randomness of `djb2` with MD5 and concludes that `djb2` totally fulfills the expectations of a fast, random PRF. In addition, [4, Sec. V] shows that the original PRF (rolling hash) is appropriate, too.

In order to test both algorithms we separated them, run them 'stand-alone' and used all optimizations modes of the `gcc` compiler[1]. Of course, both versions are improved for performance, e.g., the struct of the rolling hash from `ssdeep` was removed. The result is given in Table 4.5 (we excluded the time it takes to read the file into a buffer).

Table 4.5.: Runtime efficiency for rolling hash and `djb2` for 500 MiB random data.

optimization mode	None	O1	O2	O3
djb2	23.620 s	11.021 s	1.236 s	1.241 s
rolling hash	9.835 s	4.315 s	1.138 s	1.085 s
djb2 / rolling hash	2.402	2.554	1.086	1.143

When integrating both PRF into `mrsh-v2`, the differences are actually higher. While rolling hash only needs 3.808 s, `djb2` needs 8.532 s within `mrsh-v2` for processing the

---

[1] `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` (last accessed 2014-03-12).

same file. Actually we cannot explain these serious differences, we recognized them by testing.

Regarding the runtime, our test concludes that `rolling_hash` outperforms `djb2`. The point is, although latter algorithm looks less complex (see Sec. 3.2), it needs to compute the hash value over the whole window at each time (7 loops per window) whereas the original version (rolling hash) is able to remove the last byte and add the new one to the hash value (only one loop per window).

4.3.2. Chunk processing

Chunk hash function. The motivation to change the chunk hash function in `mrsh` from FNV to MD5 was that "FNV is not a collision-resistant function and has some known collision issues [...] especially for inputs with lower entropy which would present a serious problem for simple hashes" [85].

The latter argument is in contrast to [69] where it says that "the high dispersion of the FNV hashes makes them well suited for hashing nearly identical strings". Moreover, `mrsh` reduces the MD5 hash value from 128 bit to 44 bit in order to insert it into the Bloom filter. Thus, the hash looses its cryptographic properties. Additionally, we discussed the necessity of security requirements for approximate matching in Sec. 2.4.

If we neglect the cryptographic properties and only focus on efficiency, then FNV outperforms MD5. To test the runtime of MD5 we took the OpenSSL library and used a freely available version of FNV-1a. The test is focused on the algorithm time (read-in time is neglected) and is solved by the `clock()`-function from C++. The result is 0.742 s from FNV versus 1.354 s of MD5. Using these functions within `mrsh-v2`, the times are 5.235 s and 6.569 s, respectively.

To conclude, `mrsh-v2` integrated the 64bit hash function FNV-1a.

Minimum chunk size. A minimum chunk size comes with two improvements. First, it overcomes one of the main attacks on `ssdeep` presented in [4] called 'adding trigger points'. Second, it increases the runtime as the PRF needs not to be computed at each offset within the input sequence. A drawback is that some details may lost. This is the case if two subsequent trigger sequences have a distance of at most $^{bs}/_{4} - 1$ (bs denotes the block size).

We illustrate this characteristics on base of an extreme example. We assume that the input byte sequence has a trigger sequence every ($^{bs}/_{4} - 1$)-th byte. They are denoted by $t_0, t_1, t_2, ...$ Then, every second trigger sequence is skipped (only trigger sequences with an even index are used). Removing the first trigger sequence t_0 from the input results in considering the trigger sequences $t_1, t_3, ...$ yielding a fundamental different similarity digest.

However, for performance reasons we agree on the same minimum chunk size as used in `mrsh`, $^{bs}/_{4}$.

4.3.3. Similarity digest

Recall, `mrsh` uses Bloom filters of size $m = 2048$ and inserted $BF_{max} = 256$ chunks each setting 4 bits within the Bloom filter. This is in contrast to our implementation which is based on the similarity digest representation and comparison of `sdhash` (see Sec. 3.3).

The Bloom filter size is still $m = 2048$ bits but we changed $BF_{max} = 160$ and $k = 5$ (five sub-hashes). The maximum is therefore 800 bit in one Bloom filter.

In order to insert the chunk hash value into a Bloom filter, we use the least significant $k \cdot \log_2(m)$ bit. As a consequence, our chunk hash function needs at least so many bits which is fulfilled by FNV-1a using the default setting $k = 5$, $m = 2048$. Algorithm 3 shows a performant proceeding how to set bits in a Bloom filter based on the FNV. In addition, the design of `mrsh-v2` allows to change the parameters like k, m or the chunk hash function.

Algorithm 3 Insertion of a chunk hash into a Bloom filter

```
h is the chunk hash value
  k = 5                                        ▷ Amount of sub-hashes.
  MASK = 0x7FF                                 ▷ To use least significant bits.
  SHIFTOPS = 11                                ▷ Calculated by $\log_2(m)$.
  filter[m]                                    ▷ Array of length m.

  for j = 0 → k − 1 do                         ▷ Create k sub hashes.
      masked_bits = ( h >> (SHIFTOPS ·j)) & MASK;
      byte_pos = masked_bits >> 3;
      bit_pos = masked_bits & 0x7;
      filter[byte_pos] |= (1 <<(bit_pos));
  end for
```

Compression. The similarity digest length depends on the block size bs, the amount of chunks per Bloom filter BF_{max} and the size of a filter m (in bits). Each Bloom filter represents approximately $BF_{max} \cdot bs$ bytes of a given input and thus the compression ratio is $m/8 \cdot 1/BF_{max} \cdot bs$.

As stated in the beginning, our implementation uses $m = 2048$ bits and $BF_{max} = 160$. Assuming these fix values, the compression ratio is $2048/8 \cdot 1/(160 \cdot bs) = 8/(5 \cdot bs)$ for $bs > 0$ and therefore adjustable by changing bs. Table 6.3 shows the proportion between block size bs and the expected similarity digest length. For instance, by default we set $bs = 160$ and thus the compression ratio is at 1.000%.

Table 4.6.: Proportion between block size bs and the similarity digest length in percent.

bs	128	160	256	320	512
expected length in %	1.250	1.000	0.625	0.500	0.313

False positive rate. Due to the changes from $[k = 4, BF_{max} = 256]$ to $[k = 5, BF_{max} = 160]$ we reduced the false positive rate

$$\left(1 - (1 - {}^{1}\!/\!{}_{m})^{k \cdot BF_{max}}\right)^{k} = \left(1 - (1 - {}^{1}\!/\!{}_{2048})^{4 \cdot 256}\right)^{4} = 0.0240 \tag{4.1}$$

down to

$$\left(1 - (1 - {}^{1}\!/\!{}_{m})^{k \cdot BF_{max}}\right)^{k} = \left(1 - (1 - {}^{1}\!/\!{}_{2048})^{5 \cdot 160}\right)^{5} = 0.0035\ . \tag{4.2}$$

which is a factor of approximately 7.

4.3.4. Comparing similarity digest

The overall idea is copied from `sdhash` and described in Sec. 3.3. In Sec. 4.2.4 we discussed the difference between fragment detection and similar file detection. Hence, our implementation has two different modes where fragment detection is set by default.

Fragment detection mode. Is an exact copy of the `sdhash` implementation.

File similarity detection. In order to achieve file similarity there are two adaptations:

1. We make use of a new function $E'_{max} = \max(|bf|, |bf'|)$ (the min function is replaced by the max function).
2. Additionally, we replace ${}^{1}\!/\!{}_{s}$ by ${}^{1}\!/\!{}_{r}$ where r and s are the amount of Bloom filters of the larger and smaller similarity digests, respectively.

4.4. Summary

This chapter analyzed the three existing approximate matching algorithm and presented several improvements with respect to efficiency and security. However, all implementations have some drawbacks. `ssdeep` is extremely efficient but is not secure against an active adversary. For `sdhash` it is the other way round. While the efficiency and the similarity digest length might be improved, it is very robust against attacks. In addition, we helped to improve `sdhash` by reporting some bugs which lead to inconsistency between implementation and algorithm description.

In `mrsh-v2`, we tried to combine the best properties of all algorithms. The result is a very efficient algorithm that has a similar robustness than `sdhash`. By design we obtain full coverage, which is a benefit in our point of view.

The developers of these algorithms pursue different strategies for proceeding and similarity digest representation which raises some further questions. For instance, which is the better format for similarity digest representation? Are there more strategies how approximate matching can be done?

5

Algorithms, concepts and applications

This chapter presents new algorithms, concepts and applications for approximate matching. Within the first three sections we describe the new algorithms called `bbHash` (Sec. 5.1, [12]), `mvhash` (Sec. 5.2, [11, 3]) and `saHash` (Sec. 5.3, [27, 111]). The origin of all algorithms are well-known concepts from different areas of computer sciences.

Sec. 5.4 addresses the lookup problem for Bloom filter based similarity digests. We explain a divided & conquer based concept including a theoretical assessment (currently in review [21]). Additionally, we present an implementation which is totally sufficient for the use case blacklisting [17]. Sec. 5.5 studies the behavior of approximate matching on network traffic which is a complete new and promising working field [48, 10]. The last section summarizes this chapter.

5.1. Random building block hashing (`bbHash`)

`bbHash` is a new algorithm for approximate matching and was published in [12]. The main contribution was to design this new algorithm which is based on proceedings from data compression and biometrics.

The overall idea of `bbHash` is based on data deduplication (e.g., [62, Sec. II]) and eigenfaces (e.g., [100, Sec. 2]). Deduplication is a backup scheme for saving files efficiently. Instead of saving files as a whole, it uses lots of small pieces. If two files share a common piece, it is only saved once, but referenced for both files. Eigenfaces are a similar approach which are deployed in biometrics for face recognition. They correspond to the well-known method in linear algebra of representing each vector of a vector space by a linear combination of the basis vectors. According to this, having a set of N eigenfaces, then any face can be represented by a combination of standard faces.

The overall proceeding is quite simple. `bbHash` creates a fixed set of random byte sequences with a length l called *building blocks*. Once the building blocks are created, `bbHash` slides through the input file byte for byte, reads out the current input byte sequence of length l, and computes the Hamming distances (HDs) of all building blocks against the current input byte sequence. If the building block with the smallest HD is smaller than a certain threshold, its index contributes to the file's similarity digest.

In contrast to other approximate matching approaches, `bbHash` is based on a comparison to external data structures, the building blocks. The building blocks are randomly chosen static byte blocks which are independent of the processed input.

5.1.1. Building blocks

A building block is a random byte sequence of length l. Our current implementation uses a fixed set of $N = 16$ building blocks. We decided on 16 as we can index each building block by a unique hex digit $0, 1, 2 \dots f$ (half a byte). Thus, we have the building blocks $bb_0, bb_1, \cdots, bb_f$. The index is later used within the similarity digest to uniquely reference a certain building block.

The length of a building block in bytes is referred to by l and influences the following two aspects:

1. A growing l decreases the speed performance as the Hamming distance is computed at each offset i for l bytes.
2. An increased l shortens the length of the hash value as there should be a trigger sequence approximately every l bytes (depending on the threshold t).

Due to runtime efficiency reasons we decided to use 'short' building block sizes compared to the file size. Currently we set $l = 128$ bytes.

To create a building block, we call the `rand()` function to fill an array of unsigned integers. Thus, all building blocks are stored in one array whereby the boundaries can be determined by using their size. As `rand()` gets the same seed each time, it is a deterministic generation process. One may consider the building blocks as a kind of initialization vectors (IV) which are common for many hash functions, e.g., SHA-1.

5.1.2. Detailed proceeding

To find the optimal representation of a given file by the set of building blocks, we slide through the input file, byte for byte, read out the current input byte sequence of length l, and compute the Hamming distances (HDs) of all building blocks against the current input byte sequence. If the building block with the smallest HD is smaller than a certain threshold, its index contributes to the file's similarity digest.

The pseudo code of our algorithm `bbHash` is given in Algorithm 4. Let L_f denote the length of the input file in bytes. Then, for each offset i within the input file, $0 \leq i \leq L_f - 1 - l$ the algorithm proceeds as follows: If BS_i denotes the byte sequence of length l starting at the i-th byte of the input, then the algorithm computes the N Hamming distances of BS_i to all N building blocks. $hd_{k,i} = HD(bb_k, BS_i)$ is the Hamming distance of the two parameters bb_k and BS_i, $0 \leq k < N$. As the HD is the number of different bits, we have $0 \leq hd_{k,i} \leq 8 \cdot l$ where we defined $l = 128$ bytes. For instance, $HD(bb_2, BS_{100})$ returns the HD of the building block bb_2 and the bytes B_{100} to B_{227} of the input. In

Algorithm 4 Pseudocode of the `bbHash` Algorithm

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

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

other words, the algorithm slides through the input, byte for byte, and computes the HD at each offset for all N building blocks as is given in Fig. 5.1.

The similarity digest of `bbHash` is formed by the ordered indicies of triggered building blocks. In order to trigger a building block to contribute to the `bbHash`, it has to fulfill two conditions:

1. For a given i (fixed offset), we only make use of the closest building block, i.e., we are looking for the index k with the smallest Hamming distance $hd_{k,i}$.
2. This smallest $hd_{k,i}$ also needs to be smaller than a certain threshold t.

Each BS_i that fulfills both conditions will be called a trigger sequence. To create the final similarity digest, we concatenate all indicies k of all triggered building blocks (in case we have two triggered building blocks for BS_i, only the smallest index k is chosen).

We have already explained why the choices $l = 128$ and $N = 16$ are appropriate for our approach. In what follows we explain how to choose a fitting threshold t.

Equal to cryptographic hash functions, we designed `bbHash` to have *full coverage*, i.e., *every byte of the input file is expected to be within at least one offset of the input file, for which a building block is triggered to contribute to the* `bbHash`. Thus, we expect to trigger every l-th byte. In order to have some overlap, we decrease the statistical expectation to trigger at every 100-th byte.

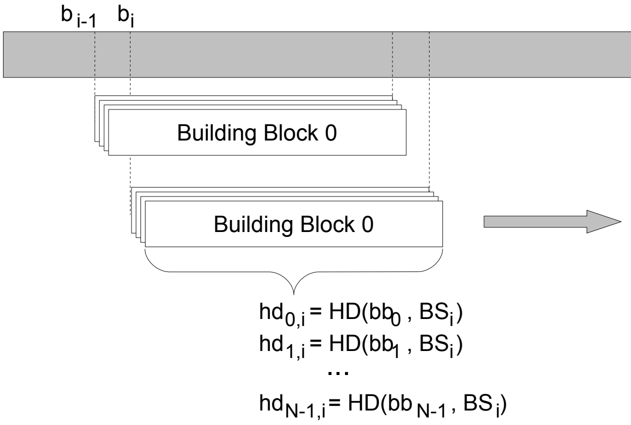


Figure 5.1.: Workflow of `bbHash`.

For our theoretical considerations we assume a uniform probability distribution on the input file blocks of byte length l. Let d be a non-negative integer (the distance) and $P(hd_{k,i} = d)$ denote the probability, that the Hamming distance of our building block k at offset i of the input file is equal to d.

1. We first consider the case $d = 0$, i.e., the building block and the input block coincide. Thus, we simply have $P(hd_{k,i} = 0) = 0.5^{l_{bit}} = 0.5^{1024}$.
2. For $d \geq 1$ we have $\binom{l_{bit}}{d}$ possibilities to find an input file block of Hamming distance d to bb_k. Thus, $P(hd_{k,i} = d) = \binom{l_{bit}}{d} \cdot 0.5^{l_{bit}} = \binom{1024}{d} \cdot 0.5^{1024}$.
3. Finally, the probability to receive a Hamming distance smaller than t for bb_k is

$$p_1 := P(hd_{k,i} < t) = 0.5^{l_{bit}} \cdot \sum_{i=0}^{t-1} \binom{l_{bit}}{i} = 0.5^{1024} \cdot \sum_{i=0}^{t-1} \binom{1024}{i} . \tag{5.1}$$

The binomial coefficients in Eq. 5.1 are large integers which were processed using the computer algebra system `LiDIA`[1] (`LiDIA` is a C++-library maintained by the Technical University of Darmstadt).

Let p_t denote the probability that at least one of the N buildings blocks satisfies Eq. 5.1, i.e., we trigger our input file and find a contribution to our `bbHash`. This is easily computed by the opposite probability that none of the building blocks triggers, that is $p_t = 1 - (1 - p_1)^N$. As explained above we aim for $p_t = 0.01$. Thus, we have to find a threshold t with

$$0.01 = 1 - (1 - p_1)^N \quad \Longleftrightarrow \quad p_1 = 1 - 0.99^{\frac{1}{N}} = 1 - \sqrt[16]{0.99} = 0.00062795 \, . \tag{5.2}$$

According to our `LiDIA`-computations for Eq. 5.1, we identify a threshold of $t = 461$.

---

[1] `http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/` (last accessed 2013-12-12).

5.1.3. Assessment and experimental results

Similarity digest length. The similarity digest length depends on three different properties: the file size L_f, the threshold t and the building block length l. If we expect that both other parameters are fixed, then

- a larger L_f will increase the similarity digest length as the input is supposed to have more trigger sequences.
- a higher t will increase the similarity digest length as more BS_i will have a Hamming distance lower than the threshold t.
- a large l will decrease the performance and the similarity digest length.

Due to performance reasons, we have decided on a building block length of $l = 128$ bytes. As we set the threshold to $t = 461$, the similarity digest length nearly results in approximately $L_f/100$ digits whereby every digit has half a byte length. Thus, the length is approximately 0.5% of the original file size.

In addition, the file type may also influence the length. Compressed file formats like ZIP, JPG or PDF consist of 'random' (i.e., high entropy) byte strings and therefore `bbHash` should yield digests of 0.5% with a high probability. This is in contrast to TXT, BMP or DOC formats where the sequences are less random.

Generation efficiency. The runtime is the main drawback of `bbHash` which is quite slow compared to other approaches. `ssdeep` processes a 10 MiB file in 0.15 seconds while `bbHash` needs about 117 seconds for the same file. This is due to the use of building blocks as external comparison data structures and the computation of their Hamming distance to the currently processed input byte sequence. Recall that at each position i we have to build the Hamming distance of 16 building blocks each with a length of 1024 bits.

To receive the HD we XOR both sequences and count the amount of remaining ones $(\text{bitcount}(bb_k \oplus BS_i))$. To speed up the counting process, we precomputed the amount of ones for all sequences from 0 to $2^{16} - 1$ bits. Thus, we can lookup each 16-bit-sequence with a complexity of O(1). However, since there are $N \cdot l_{bit}/16 = 16 \cdot 1024/16 = 1024$ lookups at each processed byte, it is quite slow.

This problem is often discussed in literature and there are several issues for improvements. For instance, [2] states that their algorithm finds all locations where the pattern has at most t errors in time $O(L_f\sqrt{t \text{ log t}})$. Compared against our algorithm which needs $O(L_f \cdot l)$ time that is a great improvement. As we make use of N building blocks, we have to multiply both runtime estimations by N.

Applicability of `bbHash` depending on the file size. Our first prototype is very restricted in terms of the input file size and will not work reliably for small files. Files smaller than the building blocks' length l cannot have a trigger sequence and cannot be hashed. To receive sufficient trigger sequences for reliable results the input file should be at least 5000 bytes, which results in $5000 - 1 - l$ possible trigger sequences. On the other hand large files result in very long hashes wherefore we recommend to process files smaller than a couple of megabytes. By changing the threshold t, it is possible to influence the similarity digest length.

Detection of fragments. A file fragment is a piece of a file which could be the result of fragmentation and deletion. By design `bbHash` allows users to compare similarity digests of those pieces against similarity digests of complete files.

Attacks. This section focuses on attacks with respect to both types of forensic issues: blacklisting and whitelisting. From an attacker's point of view anti-blacklisting/anti-whitelisting can be used to hide information.

Anti-blacklisting means that an active adversary manipulates a file in a way that approximate matching will not identify the files as similar–the similarity digests are too different. We rate an attack as successful if a human observer cannot see a change between the original and manipulated version (the file looks/works as expected). If a file was manipulated successfully then it would not be identified as a suspicious file and will be categorized as unknown file.

The most obvious method is to change the triggering whereby the scope of each change depends on the HD. For instance, assuming a HD of 450 at position i, an active adversary has two possibilities:

1. He needs to change at least 11 bits in this segment to overcome the threshold t and kick out this trigger sequence from the final similarity digest.
2. He needs to manipulate it in a way that another bb has a closer HD.

From the perspective of an active adversary: in the best case each building block has a HD of 460 and a '1-bit-change' is enough to manipulate the triggering. In this case an active adversary approximately needs to change $\frac{L_f}{100}$ bits, one bit for each position i. Actually a lot more changes need to be made as there are also positions where the HD is much lower than 460.

Anti-whitelisting means that an active adversary has similarity digests from a whitelist (digests from good files) and manipulates a file (normally a suspicious one) that its digest matches to one on the whitelist. Again, we rate an attack as successful if a human observer cannot see a change between the original and manipulated version.

In general this approach is not preimage-resistance as it is possible to create files for a given similarity digest: generate valid trigger sequences for each building block and add some zero-strings in between.

The manipulation of a specific file to a given similarity digest should also be possible but will result in a corrupted file with a high probability. In a first step an active adversary has to remove all existing trigger sequences (result in approximately $\frac{L_f}{100}$). Second, he needs to imitate the triggering behavior of the white-listed file which will cause a lot more changes.

5.2. Majority vote hashing (`mvhash`)

This section explains the underlying idea of our second new approximate matching algorithm called `mvhash` which was published in [11, 3]. The main contribution was to design an approximate matching algorithm that is based on the popular compression technique called run length encoding [7, Sec. 1.10].

The algorithm consists of three phases: First, majority vote is done on the bit level in order to transform any byte sequence into long runs of 0s and 1s. Second, run length encoding (RLE) is applied to represent these sequence of 0s and 1s by its length (in bytes). And third, the similarity digest is created. Due to the majority vote, small local changes of the input byte sequence do not affect the majority vote as the majority of bits remains unchanged. Thus, the similarity digest does not change and similarity is preserved.

Our implementation consists of two branches, `mvhash-L` and `mvhash-B`, which differ by their similarity digest representation. The former outputs Base64 encoded similarity digests which are comparable by applying the Levensthein distance. All details about `mvhash-L` are given in [3]. On the other hand, `mvhash-B` utilizes Bloom filters for the similarity digests which is the focus of the upcoming sections.

5.2.1. Algorithm design and proceeding

Phase 1 - majority vote. The core idea of the first phase is to transform an input into long runs of equal bits, which may easily be compressed during the subsequent phases. The transformation is based on a majority vote.

Before explaining the majority vote phase, we have to introduce some notations. Let BS denote a byte sequence (e.g., a file) of length L. The byte at position i of the input ($0 \leq i < L$) is written as B_i. Furthermore, N_i denotes the n-neighborhood of B_i, i.e., a byte sequence of length $(n+1)$ (n has to be even):

$$N_i := B_{i-\frac{n}{2}}, B_{i-\frac{n}{2}+1}, \ldots, B_{i-1}, B_i, B_{i+1}, \ldots, B_{i+\frac{n}{2}-1}, B_{i+\frac{n}{2}} \ .$$

At the beginning or the end of the input the length of the n-neighborhood is smaller than $n+1$, that is we do not perform any padding. For instance, we have

$$\begin{aligned} N_0 &= B_0, B_1, B_2, \ldots, B_{n/2} \text{ (of length } \frac{n}{2}+1) \\ N_1 &= B_0, B_1, B_2, \ldots, B_{1+n/2} \text{ (of length } \frac{n}{2}+2) \end{aligned}$$

The function $bitcount(N_i)$ returns the amount of bits set to 1 for N_i. If $bitcount(N_i)$ is greater or equal to a certain threshold t ($bitcount(N_i) \geq t$), the majority vote of B_i is $0xFF$ and $0x00$ otherwise.

An example of the majority vote step is given in Fig. 5.2, where the 2-neighborhood of the byte B_5 is considered. We have $N_5 = 11001100.01110101.00111000$, which results in $bitcount(N_5) = 12$. As we use the threshold $t = 12$, the majority vote yields $0xFF$ for B_5.

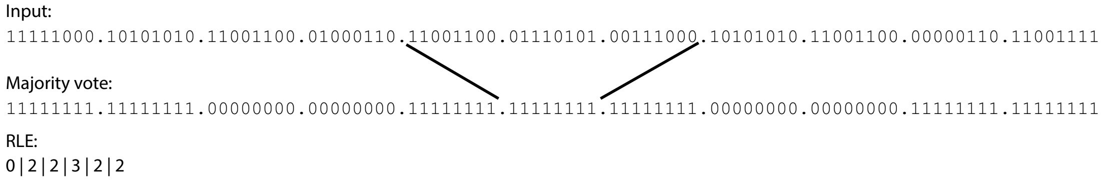


Figure 5.2.: A sample majority vote of N_5 (upper part) and subsequent RLE (lower part) with parameters $n = 2$, $t = 12$.

Finally, we have to explain how to set the threshold t to a reasonable value. By intuition one could assume that if at least half of the bits within N_i are 1, then the majority vote of B_i is $0xFF$. Thus, $t = {}^{(n+1)\cdot 8}\!/_2$ seems to be a canonical threshold. As we discuss later, there are byte sequences where the most significant bit (MSB) is 0 most of the time which would distort the results of the majority vote if the canonical threshold is used. Therefore, we revise the canonical choice of t and use

$$t = \frac{(n+1)\cdot ib}{2} \tag{5.3}$$

where ib is the average amount of influencing bits for a byte ($1 \leq ib \leq 8$). For instance, if MSB is always zero, then $ib = 7$ (the default value is $ib = 8$).

Note, we have to adjust the factor $n + 1$ in Eq. 5.3 to the actual length of the n-neighborhood, if the neighborhood does not comprise $n + 1$ bytes. As an example, we have $t = \frac{(\frac{n}{2}+1)\cdot ib}{2}$ in case of the n-neighborhood of B_0.

Phase 2 - encoding the majority vote bit sequence with RLE. In order to reduce the length of the majority vote bit sequence, we adjust the run length encoding (RLE) technique. RLE simply counts the amount of identical consecutive bytes and returns this number. Our implementation assumes that the majority vote bit sequence starts with a 0-run. Thus, if there is a 1-run in the beginning, the first RLE element is set to 0. An example is shown in the lower part of Fig. 5.2.

Phase 3 - Similarity digest generation. As mentioned in the beginning, there are two branches of `mvhash` which have a Bloom filter similarity digest and a Base64 encoded similarity digest, respectively. The problem with the Base64 encoded one is the runtime of comparing two digests which is solved using the Levensthein distance–quadratic complexity. Hence, this paragraph shows how to insert a RLE sequence into a Bloom filter of size $m = 2048$ bits.

In contrast to other existing approaches, `mvhash` does not make use of a hash function, but uses the RLE sequence directly. To insert the RLE sequence, it is divided in *groups*. Each group consists of $\log_2(m)$ consecutive RLE elements, that is $\log_2(2048) = 11$. Next, the RLE elements are reduced modulo 2. Due to the modulo 2 operation, the result is a bit sequence $b_{10}b_9b_8 \ldots b_0$. To insert this bit sequence into a Bloom filter, it is divided into two parts:

$v_1 = b_{10}b_9b_8b_7b_6b_5b_4b_3$ is used to identify the byte within the Bloom filter and

$v_2 = b_2b_1b_0$ is used to identify the bit within the byte.

To identify the bit which should be set, go to byte number v_1 simply by counting from left to right and set bit number v_2 within this byte, where $v_2 = 0$ means that the least significant bit is set.

In order to be alignment robust, groups cannot be consecutive and need to have an overlap. The only two possibilities are an overlap by 9 or 10 elements whereby 9 obviously has a better compression. Thus, when sliding through the RLE sequence, two subsequent groups need to share 9 RLE elements, that is we shift the group by 2 elements when turning to the next one.

An example is illustrated in Fig. 5.3 where we marked the first three groups in the RLE sequence. Group 1 sets one bit within the Bloom filter, group 2 sets the next bit within the Bloom filter and so on. The position of the bit of group 1 is determined as follows (bit positions for group 2 and group 3 are illustrated only):

1. Identify the first group of 11 RLE elements: 0.2.2.3.2.2.2.1.4.13.14
2. Use the modulo 2 operation for each element: 0.0.0.1.0.0.0.1.0.1.0
3. Divide the bit sequence in 2 parts: $v_1 = 00010001 = 0x11$ and $v_2 = 010 = 0x2$.
4. Go to byte $v_1 = 17$ of the Bloom filter and set bit $v_2 = 2$ to one.

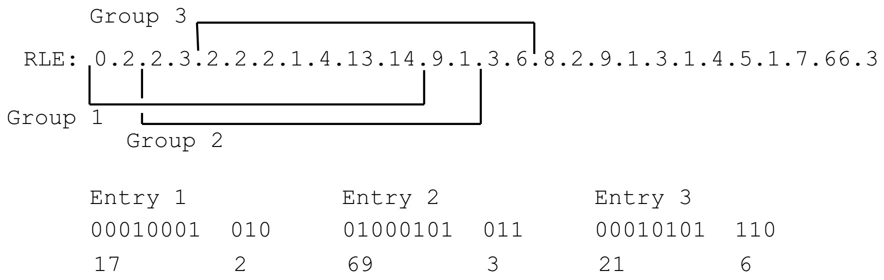


Figure 5.3.: An illustration of how the RLE encoded string is processed.

One may argue that overlapping groups by 9 is redundant. In fact, there are two reasons why it is this way. First, if only one RLE element changes, this affects up to 6 groups. Thus, there is a high probability that bits flip within the Bloom filter. Second is the more important fact, an overlapping by 9 makes our algorithm alignment robust which is best explained by example. Taking the marked three groups from Fig. 5.3 there are two options in the case of a manipulation. First, it is possible that only the value of a RLE element itself changes, e.g., the second byte which is 2 turns to X. This would influence `Group 1` only. Second, it is possible that a manipulation causes a 'deletion' of the RLE element, e.g., there are lots of changes within the underlying byte sequence which changes the 'polarity' from a 1-run to a 0-run.

Assuming the RLE sequence given in Fig. 5.3, a heavy *change* could cause a polarity flip of the second RLE element. Due to the fact that we only changed bytes, the RLE sequenced `0.2.2.` turned into a single RLE element `4` $(0+2+2)$. The exact example is given in the following where we can see that `G3` and `G2` are equal.

```
Original sequence                       manipulated sequence
Input 0.2.2.3.2.2.2.1.4.13.14.9.1.3.6   Input 4.3.2.2.2.1.4.13.14.9.1.3.6
G1    0.2.2.3.2.2.2.1.4.13.14           G1    4.3.2.2.2.1.4.13.14.9.1
G2        2.3.2.2.2.1.4.13.14.9.1       G2        2.2.2.1.4.13.14.9.1.3.6
G3            2.2.2.1.4.13.14.9.1.3.6
```

To conclude, a RLE polarity flip affects three RLE elements and result in a new one. Thus, we need a group shifting by two.

Similarity digest comparison. The comparison algorithm is self created and based on the Jaccard index [53]. As our similarity digest is composed of a sequence of Bloom filters, we first explain how to determine a distance score of two Bloom filters. In a second step, the computation of the final match score is presented.

The distance score di_{score} of two Bloom filters is based on the Hamming distance (HD). Hence, the lower the score, the more similar are two Bloom filters. The number of groups which are entered into a Bloom filter is fix, except that the last one can have fewer entries. Note, small files may only have one Bloom filter. Therefore, the HD is considered relative to the number of entries in the Bloom filters.

Let bf_1, bf_2 be two Bloom filters and let $|bf|$ denote the number of 1-bits within a filter. Furthermore, $hd(bf_1, bf_2)$ computes the Hamming distance between two filters. Then, the distance score di_{score} of two filters is

$$di_{score}(bf_1, bf_2) = \frac{hd(bf_1, bf_2)}{|bf_1| + |bf_2|} \cdot 100 \ . \tag{5.4}$$

Obviously we have $0 \leq di_{score} \leq 100$ where 0 indicates a perfect match. A distance score close to 0 means a small distance of the two Bloom filters and Thus, a high similarity of the underlying RLE sequences.

As a similarity digest consists of a list of Bloom filters, we build the average over all best-matching filters. Let $SD = \{bf_1, bf_2, \ldots bf_s\}$ and $SD' = \{bf'_1, bf'_2, \ldots bf'_t\}$ be two similarity digests (Bloom filter sequences), which shall be compared. We assume $s \leq t$. Then the final similarity score SC-function is

$$SC(SD, SD') = \begin{cases} -1, & \text{if } t - s > 4 \\ 100 - \frac{1}{s} \sum_{i=1}^{s} \min\limits_{1 \leq j \leq t} di_{score}(bf_i, bf'_j), & \text{otherwise} \ . \end{cases} \tag{5.5}$$

In other words, to receive a final similarity score between two similarity digests we do an all-against-all comparison of Bloom filters and average the lowest distance scores. As a high similarity shall be represented by a large final score, we subtract this value in a last step from 100 (recall that a small distance score means a high similarity). If the number of Bloom filters in SD, SD' is too different ($t - s > 4$), a comparison is not possible and the algorithm returns -1.

5.2.2. Design decisions

`mvhash` depends on multiple parameters and therefore this section explains their default values. The main aim is to have a similarity digest length of approximately 0.5% of the input file length, which is adjustable by 3 settings. For tuning the configuration of `mvhash` we build a *c-corpus* (configuration corpus) by downloading files from a search engine using various key words. Table 5.1 provides an overview of the file types and their average file size in our c-corpus.

Table 5.1.: Statistics of the c-corpus.

	JPG	DOC	PDF	TXT
Number of files	2066	2000	1723	2000
Average file size (kB)	251.4	293.0	1022.1	53.2

`mvhash` needs the following three parameters:

ib is the amount of influencing bits per byte and 8 by default, $1 \leq ib \leq 8$. This parameter is used to adjust the threshold t of the majority vote as defined by Eq. 5.3. There are file types where the most significant bit (MSB) is mostly zero. For instance, if we analyze ASCII-encoded[2] text documents, we realize that the MSB is hardly ever used, which influences the majority vote. Thus, for text files the *ib* parameter should be adapted to $ib = 7$.

gr is the amount of groups which are inserted into a Bloom filter. After *gr* groups are inserted, a new Bloom filter is created. Thus, the similarity digest of an input is a sequence of one or more Bloom filters each with a size of 256 bytes.

In order to find a suitable value for *gr*, we have to consider that the more groups are inserted in a Bloom filter, the shorter the similarity digest and vice versa. However, if *gr* increases the possibility for collisions rises, too. The difficulty was finding a good trade-off between compression and detection rate.

Table 5.2.: Statistics of `mvhash` for JPG files.

gr	512	1024	2048	4096
$p(bit = 1)$	22.12%	39.35%	63.22%	86.47%
Relative similarity digest length	1.26%	0.78%	0.59%	0.52%
similarity digest sensitivity (in bits)	3.89	3.03	1.84	0.68

Table 5.2 summarizes the main results of our analysis. Row $p(bit = 1)$ is according to Eq. 5.6 the probability that a certain bit is 1 after inserting *gr* groups. The formula of Eq. 5.6 may easily be derived, if a uniform probability distribution is

---

[2] `http://www.asciitable.com/` (last accessed 2014-03-13).

assumed:

$$p(bit = 1) = 1 - \left(1 - \frac{1}{2048}\right)^{gr} \tag{5.6}$$

The relative similarity digest length is the ratio between input length (original file size) and the corresponding output. Row 4 *similarity digest sensitivity* denotes the expected amount of bits which will be influenced in the Bloom filter if one RLE element changes.

Recall, a group starts at every second RLE element and covers 11 RLE elements. Thus, changing one RLE element effects 5 to 6 groups except in the beginning or at the end of the RLE sequence. Assuming $gr = 2048$, approximately 63.22% of the Bloom filter bits are set to one. As a consequence, the manipulation of one RLE element changes approximately $(1 - 0.6322) \cdot 5 = 1.84$ bits within a filter.

By default we set $gr = 2048$ as 1.84 seems plausible for such a small change and 0.59% is a good compression.

n is the size of the neighborhood to perform the majority vote. Our tests show that a large neighborhood produces long runs, but loses accuracy and vice versa. Table 5.3 presents the average length of runs for different file types. We denote this quantity by arl. As expected, the average length for compressed file types like JPG/PDF is shorter than for uncompressed file types like DOC/TXT. Thus, besides the neighborhood size, the run lengths also depend on the file type, more specifically the entropy of the byte sequences.

Table 5.3.: Average length of runs (arl) for different file types ($ib = 8$).

	n=20	n=50	n=100	n=200
JPG	13.75	24.97	41.91	73.40
PDF	14.31	29.49	47.15	81.89
DOC	26.86	55.88	96.69	172.39
TXT	37.41	275.39	915.62	1886.59

Finally, we explain our approach to set n for different file types. We chose one compressed file type (JPG) and one uncompressed file type (DOC). We neglect TXT and PDF as they show similar behavior than DOC and JPG, respectively. If the RLE sequence of the second phase of `mvhash` is shorter than 11 elements, it is not possible to generate an entry for the Bloom filter. Using our c-corpus and the neighborhoods of Table 5.3, we search for the largest neighborhood which creates an RLE encoded string of at least 11 elements for all files in the c-corpus. The result is $n = 20$ for DOC-files and $n = 50$ for JPG-files.

5.2.3. Assessment and experimental results

To assess `mvhash` we used the t5-corpus and the c-corpus. Overall `mvhash` yielded similar results for both corpuses. We decided on the following settings based on our analyzes from the previous section:

JPG-Files		DOC-Files	
n:	50	n:	20
ib:	8 (default)	ib:	7
gr:	2048	gr:	2048

Similarity digest length. We agreed on inserting 2048 groups in each Bloom filter. As each group consists of 11 RLE elements but overlap by 9 RLE elements, each group effectively represents 2 RLE elements. Accordingly, each Bloom filter is able to compress approximately $m \cdot 2 \cdot arl$ bytes and thus, we obtain a compression ratio of $\frac{m/8}{gr \cdot 2 \cdot arl}$.

For instance, let $n = 50$ and the average run length for JPG is $arl = 24.97$. This results in a compression ratio of $\frac{2048/8}{2048 \cdot 2 \cdot 24.97} = 0.0025 = 0.25\%$ for JPG files.

Practical tests on the c-corpus showed that the actual similarity digest length differs. Having $n = 50$ for JPG files and $n = 20$ for DOC files, `mvhash` results in 0.59% and 0.47%, respectively. This is due to lots of small files which have a similarity digest length of 256 bytes (= one Bloom filter) at least. For instance, imagine all files are small and the similarity digest always consists of only one Bloom filter. Then, the absolute similarity digest length is constant.

Accuracy. Accuracy is the ability to detect similar files with low costs in terms of false positive and false negative results. Therefore, we first need to identify a threshold which may distinguish between similar and non-similar files.

As said, the c-corpus is a collection of randomly collected files, therefore most files should be non-similar. Performing an all-against-all comparison of all JPG-files in the c-corpus, `mvhash` outputs multiple scores up to 70 with none above. Hence, we claim that no non-similar pair gets a score above 70 and define this as the threshold.

Since no JPG file pair had a score above 70, we state that `mvhash` does not produce false positive results. To identify false negative results, we performed the all-against-all comparison by using `sdhash`. If `sdhash` detects similar files that `mvhash` does not, it is a false negative. Using `sdhash`, no file pairs got a score above 21 which is `sdhash`' threshold for non-TXT files [81]. This means that for JPG, `mvhash` produces neither false positive or false negative results.

In addition, we performed similar tests for DOC-files. It turned out that there were lots of file pairs for each value up to 70, but only some few scores above. Thus, also for DOC-files 70 seems a suitable threshold. Manually analyzing the 21 file pairs with a score above 70, we found 8 non-similar and 13 similar file pairs. To conclude, `mvhash` produces some false positive results for DOC-files. To analyze false negatives, we again run `sdhash` with a threshold of 5 (see [81]). Out of the nearly 2 million comparisons, thousands were identified to be similar, e.g., DOC-files share some common syntax due to the file format. However, in the case of `mvhash` we rate them as false positives. In addition, as there are some thousands file pairs, it is not feasible to manually determine the similarity. To summarize, we do not know if false negatives are an issue for `mvhash`.

Finally, we decided to include a real-life scenario. Our basis is a 20 page DOC-file named Barker.doc[3] with text only, based on the original file we made two different versions. In the first version we included a table in the middle of the file, in the second we included a picture in the middle. Comparing these files showed true positives in both cases, where the detailed results are given in Table 5.4.

Table 5.4.: Real-life example of `mvhash`.

file	size	`mvhash` score	`sdhash` score
Barker.doc	115 kB		
Barker.doc with table	125 kB	99	65
Barker.doc with picture	149 kB	95	65

5.3. Statistical analysis hashing (`saHash`)

This section presents another new idea for approximate matching called **s**tatistical **a**nalysis **hash** abbreviated `saHash` which was published in [27, 111]. This approach is based on k sub-hash functions where each of them creates its own sub-hash value. In order to receive the final digest, all k sub-hash values are concatenated. `saHash` estimates the byte level similarity based on the Levenshtein distance and therefore, we have an exact measure of similarity which differentiates this algorithm from all existing ones.

Note, `saHash` allows to detect near-duplicates only, where near means that the underlying byte sequences are essentially the same but vary by some hundreds Levenshtein operations. In this case, `saHash` provides a lower bound for the Levenshtein distance between both byte sequences as its similarity score.

5.3.1. Algorithm design and proceeding

The algorithm is based on k independent sub-hash functions also called *statistical approaches*. Each results in an own hash value whereby the final fingerprint is created by concatenating all k sub-hash values. In order to compare the final fingerprint, it is split into its k sub-hash values and k comparison functions are used to estimate the Levenshtein distance. We denote this by similarity score.

The procedure to identify k sub-hash functions was simple. We started using trivial sub-hash functions such as the byte sequence length or byte frequency. We stopped

---

[3] It is included in the *mvhash (v3.0)* ZIP-file and available at `https://www.dasec.h-da.de/staff/breitinger-frank/#downloads` (last accessed 2014-04-01).

adding new sub-hash functions when we couldn't find any attack to overcome our approach. Attack in this case means the following. Let BS, BS' be two different byte sequences. Furthermore let $LD(BS, BS')$ the Levenshtein distance of BS and BS' and $saH(BS, BS')$ the Levensthein assumption of `saHash`. An attack is valid if an attacker can find BS, BS' such that $LD(BS, BS') \leq saH(BS, BS')$.

At the end we tested to see if all approaches were still necessary, as it might be possible that an approach is already covered by another one. In case an active adversary finds an exploit, the modular design allows to add a new sub-hash function and comparison function.

Currently `saHash` uses $k = 4$, all described in the following section. The current prototype is available online[4]. Our implementation was tested with Microsoft visual studio C++ and GCC on 32 Bit systems. Thus, there might be problems running it in 64 Bit mode.

Statistical approaches. This section describes all four sub-hash functions of `saHash`. Let BS denote a byte sequence (i.e., a file) of bytes $B_0, B_1, \dots B_{l-1}$ of length l ($|BS| = l$; parameters are associated with the file) which serves as input. In order to reduce some sub-hash values we use a modulus mo determined as follows

$$mo = 2^{\max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)} . \tag{5.7}$$

The value of mo depends on the exponent $\max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)$ which results in a minimum modulus of $mo = 256$. We decided on this due to uniformity so that the range of values is utilized well. For instance, in [13] we observed that the average file size for a Windows XP installation is 250 KB $\approx 250 \cdot 10^3$ bytes. Assuming an equal distribution of bytes, every byte will appear $\approx 10^3$ times. Thus, the modulo counters overflow and the probability distribution for all modulo counters approach a uniform distribution. As a consequence, the lower bound ensures a sufficient magnitude of mo. If we handle large files we need to store more information to uncover more Levenshtein operations therefore mo increases.

Let h_k denote the k-th statistical approach a.k.a. sub-hash function for $0 \leq k \leq 3$ and BS a byte sequence of length l.

h_0 returns the byte sequence length equal to l as an `unsigned integer`. The bit size of the integer depends on the compilation mode and is 32 bit or 64 bit. h_0 is an order criteria and used to drop unnecessary comparisons: If the lengths of two byte sequences deviate too widely (depending on the thresholds), the comparison is canceled.

Trivial attack: Any BS' with the same length l will result in the same hash value $h_0(BS')$.

h_1 returns the byte frequency abbreviated $freq$ (it is a histogram showing bytes `0x00` to `0xff`, i.e., `0` to `255`). For instance, to receive the byte frequency of `0xaa`, the

---

[4] `https://www.dasec.h-da.de/staff/breitinger-frank/#downloads` (last accessed 2014-04-01).

algorithm counts the occurrences of `0xaa` within BS. The result of $h_1(BS)$ is an array containing the frequencies of all 256 bytes ($freq_0$ to $freq_{255}$) congruent *mo*. In order to reduce the fingerprint further, we only store values from $freq_0$ to $freq_{254}$, as $freq_{255}$ is then predictable by

$$freq_{255}(BS) = (h_0(BS) - \sum_{i=0}^{254} freq_i(bs)) \mod mo.$$

The byte length is $|h_1(BS)| = \frac{255 \cdot \log_2(mo)}{8}$.

Trivial attack: Any BS' of length l containing the same amount of bytes in any order will overcome the combination of h_0 and h_1.

h_2 returns the transition frequency, abbreviated $tfreq$. First we do an initial left circular bit shifting[5] by 4 and then proceed as in h_1. In other words it is the amount of one specific transition within BS where transition means the 4 lowest bits of B_x and the 4 highest bits of B_{x+1}.

As above we only store $tfreq_0$ to $tfreq_{254}$ congruent to mo and therefore $|h_1(BS)| = |h_2(BS)|$.

Trivial attack: Any BS' of length l containing the same amount of bytes in any order where we switch blocks having the same transition to overcome the combination of h_0, h_1 and h_2.

h_3 returns the unevenness array, abbreviated *uneva*, which is an *own* creation and inspired by the question of which statistical features are relevant. Besides the frequency of occurrences, we identified the frequency of *repeated occurrence*. The unevenness of a byte B within BS is a measure of how evenly the occurrences of B in BS are spread.

The result is an ordered array of 256 bytes starting with the less 'uneven'. As the last uneven byte is predictable (it is the one we have not found within *uneva*), the byte length is $|h_3(BS)| = 255$.

Algorithm 5 shows how to compute the unevenness of all 256 bytes for a given byte sequence BS.

In the following we describe the three parts of the algorithm, each `for`-loop.

1: mean-value calculates the average distance between the same B. Thus, it is the sum of all distances between the same B divided by the amount of occurrences $freq$. The additional +1 is due to the last occurrence of a B until EOF. As the sum of all distances is equal to the input length, we can use the fix value *length*(BS).

2: deviations for each B is equal to the square deviation between each occurrence of a B.

---

[5] `http://en.wikipedia.org/wiki/Circular_shift` (last accessed 2013-02-18).

Algorithm 5 Generating *uneva* array containing the unevenness of BS and byte i at the i-th array position.

```
**array** uneva[256] = means[256] = {0, 0, ...}

for i do 0 to 255                                         ▷ Create mean values.
    means[i] = $\frac{\texttt{length}(BS) + 1}{\texttt{freq}(BS, i) + 1}$
end for

array lastOcc[256] = {-1, -1, ...}
for i do 0 to length(BS) - 1                              ▷ Create deviations for each byte.
    byte = BS[i]
    dev = means[byte] - (i - lastOcc[byte])
    uneva[byte] += dev^2
    lastOcc[byte] = i
end for

for i do 0 to 255                          ▷ Deviation from last occurrence until end.
    dev = means[i] - (length(BS) - lastOcc[i])
    uneva[i] += dev^2
    uneva[i] *= (freq(BS, i) + 1)
end for
```

3: deviation until end of a B describes last occurrences until EOF.

The actual saHash is denoted by H and comprises the concatenation of all four sub-hashes, i.e., $H(BS) = h_1(BS) \parallel h_2(BS) \parallel h_3(BS) \parallel h_4(BS)$.

Sub-digest comparison. Let BS_1, BS_2 be two byte sequences and let d_k denote a distance function that returns a measure of the distance of the sub-digests $h_k(BS_1)$, $h_k(BS_2)$. It is important to note that $d_k\left(h_k(BS_1), h_k(BS_2)\right)$ is a parameter of the 'inverse similarity' of both sub-digests, i.e., the more distant the sub-digests the less similar they are.

sub-hash function for h_0: An obvious distance function for d_0 which represents the byte sequence length is defined as $d_0 = |h_0(BS_1) - h_0(BS_2)|$.

sub-hash function for h_1: In order to define the measure for h_1 which is an array containing the frequencies of all bytes, we use a function $sub_i(h_1(BS_1), h_1(BS_2))$

that subtracts the i-th element from $h_1(BS_1)$ from the i-th one in $h_1(BS_2)$:

$$\begin{aligned} tmp &= \sum_{i=0}^{255} |sub_i(h_1(BS_1), h_1(BS_2))| \\ d_1 &= \left\lceil \frac{tmp - d_0}{2} \right\rceil + d_0. \end{aligned}$$

First, we sum the absolute values for all position-differences for all frequencies. Thus, we know how many byte frequencies are different. In general there are two possibilities. If the $|BS_1| = |BS_2|$, we need $\left\lceil \frac{tmp-d_0}{2} \right\rceil$ substitutions. In case of an unequal-length we need to add d_0.

For instance, `AAAAA` and `AABA` result in $d_0 = 1$ and $tmp = 3$. Thus, $d_1 = \lceil(3 - 1)/2\rceil + 1 = 2$. The difference in length is considered by $d_0 = 1$ while all other differences can be corrected due to a substitution (`B` into `A`).

sub-hash function for h_2: h_2 is similar to h_1 wherefore we again use the same auxiliary function *sub*. One difference is the division by 4 instead of 2 which is caused by the initial left circular bit shifting operation: one deletion/insertion can influence up to 4 positions within the $tfreq$ array.

$$\begin{aligned} tmp &= \sum_{i=0}^{255} |sub_i(h_2(BS_1), h_2(BS_2))| \\ d_2 &= \left\lceil \frac{tmp - d_0}{4} \right\rceil + d_0. \end{aligned}$$

sub-hash function for h_3: To define the similarity of two `uneveness` lists, we first introduce a function $pos_b(h_3(BS))$ that returns the position of byte b inside the array. The maximum distance is $256 \cdot 128$ which happens if the array is shifted by 128.

$$\begin{aligned} tmp &= \sum_{b=0}^{255} |pos_b(h_3(BS_1)) - pos_b(h_3(BS_2))| \\ d_3 &= \left(1 - \frac{\mathrm{tmp}}{256 \cdot 128}\right) \cdot 100 \end{aligned}$$

Final similarity decision. We decided to have a binary decision and therefore `saHash` outputs either *yes* two byte sequences are considered to be similar or *no*. To receive this result `saHash` expects two thresholds t_{LB} and t_{CS}. The first one is a lower bound of Levenshtein operations and the latter one is called *certainty score* and allows assumptions of the quality to be made.

In order to make an assumption about the lower bound of Levenshtein operations LB we set $LB = \max(d_0, d_1, d_2)$. CS is simply set to d_3. If $LB \leq t_{LB}$ and $CS \geq t_{CS}$ then two byte sequences are considered as similar.

The following shows how the default thresholds are defined. `saHash` identifies a match with default settings if two different byte sequences yield $LB \leq 282$ and $CS \geq 97$. Of course, a user can adjust these default values but then the probability of obtaining false positives is raising.

To define the default thresholds t_{LB} and t_{CS}, a set of 12,935 files was used. First, we did an *all-against-all-other* comparison and entered the LB values into a matrix. Next, we compared all file pairs by hand starting at the lowest LB within the matrix and stopped at the first false positive which had a LB of 283. Thus, we set $t_{LB} = 282$. As all true positives exceed 97 for CS we set $t_{CS} = 97$.

Due to the large test corpus we claim that $t_{LB} = 282$ is a realistic threshold to avoid false positives. We have to keep in mind that this is a lower bound, the real amount of Levenshtein operations might be higher.

Adjusting sub-digests to modulus. This paragraph demonstrates the special case of having two different moduli. Let BS_1 and BS_2 be two inputs yielding the moduli mo and mo', respectively, where $mo < mo'$. Due to the fact that `saHash` is designed to only identify small changes between two inputs, the moduli will at most differ by a factor of 2 and therefore $mo' = 2 \cdot mo$.

In order to build the distances d_1 and d_2, we have to adjust $h_1(BS_2)$ and $h_2(BS_2)$. As $(x \bmod 2 \cdot mo) \bmod mo = x \bmod mo$, we simply go through the array for $h_1(BS_2)$ and $h_2(BS_2)$ and use mo to recompute the frequency values. Afterwards we are able to compare the sub-hash values.

5.3.2. Assessment and experimental results

This section aims at evaluating the correctness and performance of our approach `saHash` with respect to efficiency. Furthermore, we compare `saHash` to the existing approaches `ssdeep` and `sdhash` to demonstrate its benefits and drawbacks.

Our working environment was a Dell Laptop using Windows 7 64 Bit with 4 GB RAM and Intel Core 2 Duo 2x2 GHz. All tests are based on a file corpus of 12,935 files[6] which have been gathered through a python script using Google.

Remark: In the following we often talk about random byte sequences. These have been generated using the function `int rand()` with seed `srand(time(NULL))`.

Generation efficiency. One property of approximate matching is *ease of computation* where we gave a high priority to generation efficiency during the `saHash` development process. As a result, the time complexity to compute $H(BS)$ for a byte sequence BS is $O(|BS|)$ and therefore optimal.

In order to compare the run time performance we generated a 100 MiB file from `/dev/urandom/` and used it for all algorithms. C++ provides a `clock()` function, which has the benefit that times of parallel processes do not influence the timing.

---

[6] 2,631 DOC, 67 GIF, 362 JPG, 1729 PDF and 8146 TXT.

As shown in Table 5.9 `saHash` is almost as fast as SHA-1 and therefore faster than both other approaches.

Table 5.5.: Ease of computation for approximate matching and SHA-1.

	`SHA-1`	`saHash`	`ssdeep 2.9`	`sdhash 3.2`
runtime	1.23 s	1.76 s	2.28 s	4.48 s

Space efficiency. The overall hash value length $|H(BS)|$ in bytes for a byte sequence BS of length l is the sum of all k sub-hashes bit length. Thus, the bit length is

$$\left.\begin{aligned} |h_0(bs)| &= 4 \text{ bytes (32 bit)}, \\ |h_1(bs)| &= \frac{255 \cdot \max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)}{8} \text{ bytes}, \\ |h_2(bs)| &= \frac{255 \cdot \max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)}{8} \text{ bytes}, \\ |h_3(bs)| &= 255 \text{ bytes}. \end{aligned}\right\} \left\lceil 4 + \frac{2 \cdot 255 \cdot \max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)}{8} + 255 \right\rceil .$$

For instance the fingerprint for a 1 MiB file ($=2^{20}$) is 897 bytes and thus, has a compression ratio of 0.08554%.

The compression property is best fulfilled by `ssdeep` which generates hash values up to a maximum of 104 Base64 characters regardless of the input length.

[80] stated that `sdhash` approximately creates hash values of 2.6% of the input. Our test with several thousand real-world files showed that `sdhash` creates fingerprints of a length of approximately 2.5%. Therefore, `saHash` has a better compression for all files larger than approximately $\frac{897 \cdot 100}{2.5} \approx 35,880$ bytes. Recall, [13] observed that the average file size of a Windows XP installation is 250 KiB.

Modulus. In this section we discuss the impact of a variable modulus. According to Eq. 5.7, the modulus is fix for all byte sequences with less than 2^{16} bytes. Following this, the factor rises by two if the input size increases by factor of four. We assume that a larger modulus is able to determine more modifications and thus, has a better detection rate.

As small files do not provide enough information and do not exhaust the counters, our test scenario only used files out of the corpus that are larger than 1 MiB which resulted in 749 files. In order to analyze the consequences of an increasing modulus, we manually changed the modulus to 7, 8, 9 and 10 bits. Based on the methodology, we searched for the smallest LB yielding a false positive and received $2,527$, $4,304$, $7,207$ and $13,503$ LB, respectively. To conclude, the higher the modulus, the more robust `saHash` is.

One thing we would like to point out is the transition from 9 to 10 bit, where it was possible to identify more true positives which were quashed using smaller moduli. Thus, we argue it makes sense to use an increasing modulus.

Correctness. This section presents a randomly driven test case and mainly focuses on false positives. Randomly in this case means that we make changes all over the file by chance.

First, we analyzed the impact of random changes for a byte sequence BS of length 250 kB (average file size for Win XP). A random change is one of the typical edit operations; deletion, insertion and substitution, each with a probability of $1/3$ whereby each byte in BS is equiprobable to be changed.

We generated a test corpus T_{700} of 700 files each of size 250 kB using the `rand()` function. Then, for each $n \in \mathbb{N}$ where $1 \leq n \leq 700$ and for each $BS \in T_{700}$ we then applied n random changes and received the file BS'. Next, all pairs (BS, BS') have been used as inputs for `saHash` to decide about similarity. We expect that as n increases the dissimilarity of the corresponding files is growing, too.

`saHash` outputs two different values LB and CS. Fig. 5.4 presents the behavior of up to $n = 700$ edit operations (x-axis) and the number of detected edit operations by `saHash` (y-axis). Even $n = 700$ yields a 'number of operations' lower 282 wherefore we only had true positives. Thus, although the default value for LB is 282, `saHash` is able to detect a lot more modifications without any false positives. Remember, the output is a lower bound for the amount of Levensthein operations.

According to our findings, we set the default threshold for CS to 97. Fig. 5.5 shows the certainty score in relation to the number of random edit operations. With a threshold of 97, `saHash` is able to detect reliably similarity for $n < 600$.

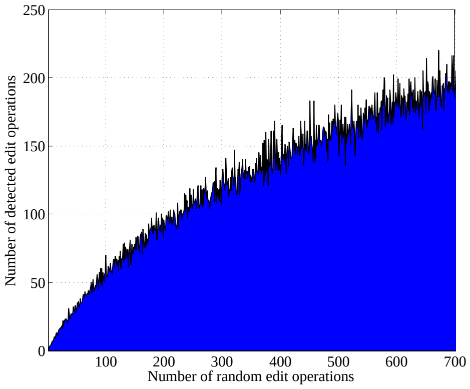


Figure 5.4.: Number of edit operations vs. detected edit operation by `saHash`.

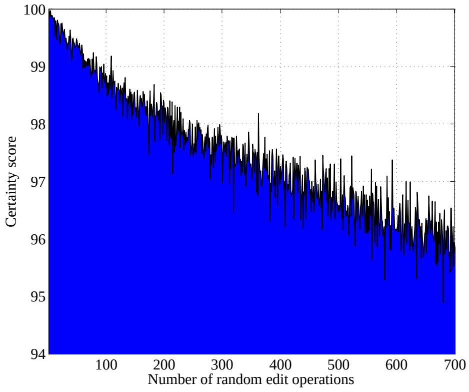


Figure 5.5.: Ratio between the certainty score and the number of edit operations.

5.4. A concept for an efficient similarity digest lookup

The main drawback of Bloom filter based similarity digests is that it is not possible to order / index them. As a consequence, looking for a single digest in a database containing z digests, comes with an 'against-all' comparison (brute-force) and thus a complexity of $O(z)$.

For instance, we measured the time to do an against-all comparison of the t5-corpus

(1.78 GiB = 4457 files) which took 1281 s. In other words, comparing $1.78 \times 1.78 = 3.17$ GiB of data takes about 20 minutes. According to this, we estimated the runtime for having 256 GiB of files in the database and a hard drive with 200 GiB of data. Overall, we have to compare $200 \times 256 = 51,200$ GiB of data which comes to $^{51,200}/_{3.17} \cdot 1281$ s ≈ 20689968 s ≈ 240 days.

In contrast, for cryptographic hashes the complexity is $O(\log_2(z))$ or even $O(1)$ depending on the storing technique: binary tree data structures or hash tables, respectively. Recently, Base64-based approaches such as `ssdeep` have also been improved (see Sec. 3.1.2 on page 23) and now operate with practical speed. However, it is not possible to adapt the n-gram technique from F2S2, since inserting features into a Bloom filter 'randomly' sets bits all over in the filter.

In the upcoming section we present an divide & conquer approach for Bloom filter based approaches (not published but submitted [21]). The rest of this section is structured as follows: Sec. 5.4.1 gives an overview of the basic operation mode of the proposed concept. Subsequently, we present the design decisions in Sec. 5.4.2 followed by a workflow for trees. A theoretical assessment for our approach is given in Sec. 5.4.4. The last section shows the special use case of a single Bloom filter.

Hierarchical tree data structure. A tree data structure $\mathbf{\Psi}$ of degree x is a tree where each node has x references to nodes (the 'children') or leaves. Let $|\mathbf{\Psi}|$ define the total number of leaves in $\mathbf{\Psi}$, then the height, i.e., the number of nodes on the longest path from the root to a leaf is estimated as

$$h(\mathbf{\Psi}) \quad = \quad \lceil \log_x(|\mathbf{\Psi}|) \rceil, \tag{5.8}$$

where $\mathbf{\Psi}$ has exactly $h(\mathbf{\Psi})$ levels. Fig. 5.6 shows examples of a binary and a tertiary tree containing a total number of $|\mathbf{\Psi}| = 7$ leaves. Obviously, for fixed values of $|\mathbf{\Psi}|$ increasing x decreases the height $h(\mathbf{\Psi})$, and vice versa.

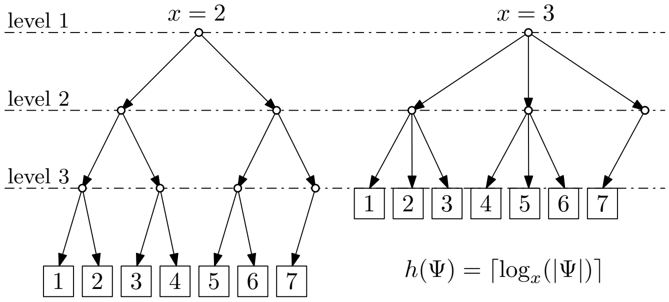


Figure 5.6.: Hierarchical tree data structures: examples of a binary tree (left) and a tertiary tree (right) containing seven leaves.

5.4.1. Proceeding overview

The key idea is to build a tree data structure of Bloom filters over all similarity digests where the leaves are 'file identifiers' (*FI*). A *FI* is a link to a database which contains

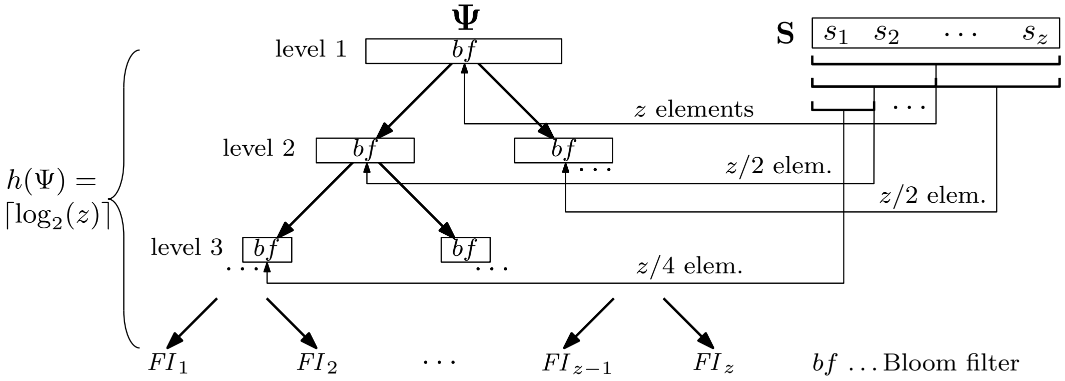


Figure 5.7.: Construction of Bloom filter-based tree data structure: schematic illustration using a binary tree.

at least the similarity digest, but may contain additional information, too. Thus, this section first shows the generation process followed by the lookup.

Tree generation. For a given set $\mathbf{S}$ containing z elements, each element $s \in \mathbf{S}$ is inserted to the root node of the tree; a huge Bloom filter. Subsequently, depending on the degree x of the tree data structure, $^{z}/_{x}$ consecutive elements are inserted to x Bloom filters stored in the children nodes of the root node. This procedure is applied recursively, i.e., in level L, $^{z}/_{x^{L-1}}$ elements are inserted into x^{L-1} different Bloom filters, while $^{z}/_{x^{L-1}} > 1$. Finally, *FI*s are stored at the corresponding leaves. The procedure is summarized in Fig. 5.8, an example for a binary tree is illustrated in Fig. 5.7.

Input:

- Set of elements, $\mathbf{S}$

Output:

- Hierarchical tree data structure, Ψ

Procedure:

1. Insert each element $s \in \mathbf{S}$ into the root Bloom filter $\mathbf{b}$
2. In level L insert x^{L-1} consecutive sequences of z/x^{L-1} elements into Bloom filters of according x^{L-1} nodes
3. Repeat 2. while $z/x^{L-1} > 1$
4. Insert *FI*s at leaves of Ψ

Figure 5.8.: Generalized construction of Bloom filter-based tree data structure.

Lookup strategy. Having a tree means that it is not necessary to compare each digest against all digests in the reference dataset; it only has to be compared against a subset of nodes. Moreover, we claim that most comparisons yield a non-match (i.e., $|good\ files| >> |bad\ files|$) and hence are dropped in the first step. Once a match is found, we trace a path down the tree data structure, ending at a leaf. If an exact match score is needed, the *FI* can be used to fetch the similarity digest from the database. This improves the lookup complexity for one digest to $O(\log_x(z))$.

The final lookup procedure is different since we do not compare files to the nodes, but to fragments of files. The impact and final workflow is discussed in Sec. 5.4.3.

Note, if we only have a single node (the root node), this works as an easy filter for the all-against-all comparison which will be discussed in Sec. 5.4.5. Only files that are found in the node need to be compared against the database. We denote this special case by *file-against-set* comparison as it outputs ‘yes, the file is in the set’ or ‘no, it is not’.

Terminology & definitions. This section gives the notations which are necessary to understand all improvements and design decisions described in the following section. For $m, k, n, z \in \mathbb{N}$, abbreviation and according descriptions are defined as,

$z\ldots$ denotes the number of files which is equal to the number of similarity digests.

$feature\ldots$ describes a byte sequence which is hashed and inserted into the Bloom filter.

$m\ldots$ denotes the Bloom filter size in bits.

$k\ldots$ number of sub-hashes where each one sets a bit in the Bloom filter.

$n\ldots$ number of features inserted into a Bloom filter.

$p\ldots$ false positive probability for an element / feature to be in the Bloom filter.

In case of `mrsh-v2` $feature$ equals a chunk of approximately 160 bytes and regarding `sdhash`, $feature$ is a sequence of exactly 64 bytes.

5.4.2. Design decisions

This section explains our design decisions for the various parameters needed to implement our concept. The first paragraph shows the correlation between the input file size and the number of features which are inserted into the Bloom filter. Next, the size and height of the binary tree is discussed. Subsequently, we introduce our match decision approach and the false positive rate. Based on all these findings, we explain the procedure of how to calculate the best Bloom filter size. Finally, the relevance of the feature hash function is discussed.

Correlation between elements (n) and file set size. In Eq. 2.1 on page 17, n denotes the number of elements that are inserted into a Bloom filter. However, the number of elements is different to the amount of files in the set ($= z$) but equal to the number of features. Hence, this section analyzes the relation between n and file set size. Let μ denote the file set size in MiB.

- `sdhash` inserts 192 features into a Bloom filter for approximately every 10 KiB of the input file. Thus, n is calculated by $n = \mu \cdot 2^{20} \cdot 192/(10 \cdot 2^{10}) \approx \mu \cdot 2^{14}$, where 2^{20} and 2^{10} is needed to change from MiB and KiB to bytes, respectively.
- In case of `mrsh-v2`, the implementation splits the input in 160-byte features. Thus, n is calculated by $n = \mu \cdot 2^{20}/160 \approx \mu \cdot 2^{13}$ where 2^{20} converts MiB into bytes.

We use $n = \mu \cdot 2^{14}$.

Seize and height of the tree. Eq. 5.8 on page 70 shows how to calculate the height of the tree if every leaf only contains one FI. If we assume a total size of Ω GiB and an average file size of ω KiB, then our dataset consists of ${}^{\Omega \cdot 2^{30}}/_{(\omega \cdot 2^{10})} = {}^{\Omega}/_{\omega} \cdot 2^{20}$ files which results in a maximum height of $h(\mathbf{\Psi}) = \lceil \log_x({}^{\Omega}/_{\omega} \cdot 2^{20}) \rceil$ which each bucket has exactly on entry.

As this is very memory consuming, there is also the possibility that a leaf is a bucket and contains multiple FIs instead of a single one. In that case the queried similarity digest must be compared to all files in the bucket. However, by storing a total number of l FIs at each leaf, we reduce the height to:

$$h(\mathbf{\Psi}) = \lceil \log_x(|\mathbf{\Psi}|/l) \rceil.$$

The actual size of the each Bloom filter (each node) is variable. However, if we require that k, p, n are fix, then the size is ${}^{1}/_{x}$ between two consecutive levels. More precisely, let $L \in \mathbb{N}$ denote the level in the binary tree ($L = 1$ is the root level) and let m_L be the Bloom filter size at the corresponding level. Then, $m_{L+1} = {}^{m_L}/_{x}$. On a subsequent level we have an x-fold number of Bloom filters yielding a constant size of all filters per level. Accordingly, the overall size of the tree is $(h(\mathbf{\Psi}) - 1) \cdot m_1$.

Remark: we reallocated the elements but the overall number is constant at each level. If the number of inserted elements reduces and k, p are fix, then the size of the filter reduces, too.

Match decision and false positives. Traditionally approximate matching compares two similarity digests against each other and returns the score. Our procedure works differently. Instead of searching for the complete files, we focus on fragments to identify potential buckets. A fragment of a file matches a Bloom filter, if r subsequent features are found in the node. Once a leaf is identified, we might perform the conventional comparison.

More precisely, Eq. 2.1 relates the false positive probability p for a single feature to the different parameters k, n, m. In fact, we are less interested in the false positive rate for a single feature but more for a fragment of a whole file. Let p_f denote the desired false positive probability for a fragment. If we require $r \in \mathbb{N}$ consecutive false positive features to be a false positive fragment, the false positive probability for a fragment is $p_f = p^r$.

For instance, if we request a false positive probability for a file (which is equal to a fragment) of 10^{-8} and set $r = 8$, then $p = 10^{8/-8} = 0.1$.

Note, $r = 8$ corresponds an approximate overlapping of $8 \cdot 160 = 1024$ bytes in case of `mrsh-v2`. We argue that this is a sufficient lower bound to talk about similarity. Of course, one may change these settings to their personal requirements.

Root Bloom filter size. Before discussing the Bloom filter size, recall probability p for a false positive from Eq. 2.1:

$$p \quad \approx \quad \left(1 - \mathrm{e}^{-kn/m}\right)^k .$$

Let n and p be given and k be the optimal value from Eq. 2.2. Then aforementioned equation becomes

$$p = \left(1 - \mathrm{e}^{-\ln(2)}\right)^{m/n \cdot \ln(2)} = 2^{-m/n \cdot \ln(2)} = e^{-m/n \cdot (\ln(2))^2} .$$

According to this, the root Bloom filter size is estimated as

$$m_1 = n \cdot {}^{-\ln p}/_{(\ln 2)^2} . \tag{5.9}$$

Based on the findings from the previous paragraph, $p = \sqrt[r]{p_f}$, we use $r = 8$ and the false positive probability for a fragment $p_f = 10^{-8}$. Then we have $p = 10^{8/-8} = 0.1$ and $m_1 = n \cdot {}^{-\ln 0.1}/_{(\ln 2)^2} = \mu \cdot 2^{14} \cdot 4.7925 = \mu \cdot 2^{16.26}$. Depending on μ, each level of the binary tree could easily have Bloom filters of 1-2 GiB (the size of the Bloom filter has to be of type 2^a where $a \in \mathbb{N}$).

Feature hash function. As known from Eq. 2.2, if n and m are given, the value k minimizing the false positive probability is

$$k = {}^{m}/_{n} \cdot \ln 2. \tag{5.10}$$

Note, k is independent from the considered level as m and n are both multiplied by ${}^{1}/_{x}$.

According to this, the best value for $k = {}^{\mu \cdot 2^{16.26}}/_{\mu \cdot 2^{14}} \cdot \ln 2 = 2^{1.73} = 3.34$. Since $k \in \mathbb{N}$ and it may vary depending on the further parameters, we propose $3 \leq k \leq 7$.

Generally speaking, to set k bits in a Bloom filter of m bits length, it requires a feature hash function of at least $k \cdot \log_2(m)$ bits. More formally, having a feature hash function of b bits, $b \geq k \cdot \log_2(m)$. Having a Bloom filter of 1 GiB therefore requires $b \geq 4 \cdot \log_2(2^{30}) = 120$ bits.

Since, the default implementations of `sdhash` and `mrsh-v2` run 160-bit SHA-1 and the 64-bit FNV hash, respectively, both algorithms have to be adapted. In order to allow larger reference sets like 256 GiB or 512 GiB which need Bloom filters of 4 or 8 GiB, we recommend implementing 256-bit versions of the hashing algorithms. For instance, set $k = 5$ and using a 256-bit hash function allows us to handle Bloom filter sizes of $2^{256/5} = 2^{51.2}$ bits $\approx 2^{18}$ GiB.

5.4.3. Workflow for trees

Traditionally the reference data is hashed and stored in a database. Next, for the comparison, the file is processed and the similarity digest is compared against all entries in the database. Due to the usage of a tree and fragments (i.e., r consecutive features) the overall procedure changes. In the following we discuss diverse design decisions including their impact. We decided for binary trees as they are very common and easy to understand.

Input:

- Bloom filter $\mathbf{b}$ and file A

Output:

- Binary match decision 1...match, 0...no match

Procedure:

1. Identify and hash feature of A
2. Add it to the similarity digest
3. Compare the feature to $\mathbf{b}$
4. If it matches, then increase counter τ and save feature hash, otherwise $\tau = 0$
5. If $\tau = r$, return '1', otherwise apply 1. to the next feature
6. Return '0'

Figure 5.9.: Generalized overview of the proposed matching function ξ.

One vs. all fragment comparisons. In general there are two possibilities of when to stop. First, one can stop when one fragment is found or second, one may stop after comparing all fragments. The difference is that in the former case probability for a false positive is higher. In the latter case the output is a list of possible leaves, i.e., file identifiers, and if several fragments point to the same leaf, the result is considered more reliable.

Depending on the personal preference, one may fetch the real similarity digests from the database based on the FI and perform the conventional comparison. Moreover, we recommend to sort the FI based on match frequency, e.g., if FI of A was matched 100 times and FI of B only twice, A is considered more important.

Analysis of subsequent level (for binary trees). If a fragment is found in the root node, we jump to the Bloom filters on level 2. Let us always start with the left node. If the fragment matches there too, we continue on the next level. In case of a negative, there are two possibilities:

1. *Hard decision*: we trust that this fragment is not a false positive and jump to the next level of the right node.

2. *Soft decision*: we verify the result by comparing the fragment against the right node.

In the former case we obtain a logarithmic lookup efficiency where in the latter case we approximately have an additional 50% comparisons. However, we reduce the false positive rate. The entire procedure is depicted in Fig. 5.10.

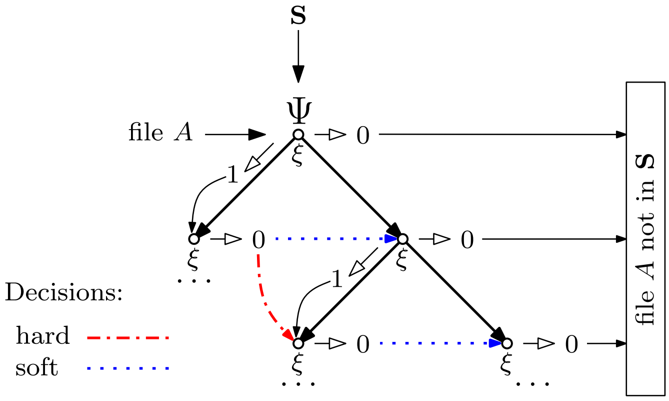


Figure 5.10.: Generalized overview of the proposed workflow.

Nearest neighbor vs. all neighbors If we are interested in having 'all similar files' instead of only 'one match', we always compare the fragment to all nodes of the next level and stop when we arrive at a leaf or if the fragment is not found. In this case complexity is upper bounded by $O(z)$, i.e., in the rather unlikely case that all FIs are similar to the given query element all nodes have to be processed, which will then require $2^{\log_2(z)} = z$ comparisons for a binary tree.

Insertion & deletion of file identifiers Within any tree data structure the insertion of an FI can be handled in $O(\log(z))$ steps by inserting according features to $\log(z)$ Bloom filters and one FI at the resulting leaf. To construct $\mathbf{\Psi}$ we recommend to start from the top and go to the bottom as we can reuse the feature hashes from higher levels.

In contrast, the deletion of an FI can not be performed on an existing tree data structure. Since features are hashed into Bloom filters in an irreversible manner, i.e., we do not know which ones result from which features and the number of times these occur, deletion has to be performed off-line. In order to delete a single FI from the proposed Bloom filter-based tree data structure $\mathbf{\Psi}$, the according FI has to be deleted from the original set $\mathbf{S}$ and $\mathbf{\Psi}$ has to constructed from scratch. However, we claim that blacklisted files will always remain blacklisted and therefore deleting a FI never happens.

Recommend setting for blacklisting With respect to blacklisting we argue that identifying one match is sufficient. Therefore, we think that 'one fragment comparison' and 'nearest neighbor' are sufficient. For more reliability and to obtain a match score, we recommend to compare the final similarity digests with each other to obtain a match score. As we stop after matching one fragment we would always check against both Bloom filters when jumping to the subsequent level.

5.4.4. Theoretical assessment for Bloom filter digests in trees

This section presents a theoretical assessment of the proposed concept. Therefore, we briefly discuss the average file sizes and used settings in the rest of this section. Next, we present an overview of the required memory for different scenarios followed by some calculations about the estimated efficiency improvement.

As we are interested in a functional evaluation, the actual type of the database (blacklist vs. whitelist) does not matter. Since no blacklisted files are available, we studied whitelisted files. To get an overview of the average file size of large datasets, we analyzed the sizes of almost 1,000,000 files in the `govdoc`-corpus. "These documents were obtained by performing searches for words randomly chosen from the Unix dictionary, numbers randomly chosen between 1 and 1 million, and randomized combinations of the two, for documents of specified file types that resided on web servers in the .gov domain using the Yahoo and Google search engines"[7].

The average file size is 494 KiB with a distribution as shown in Table 6.10 – nearly 91% of all files are smaller than 1 MiB. Additionally, we checked the average file size on our working stations which is 510 KiB and 611 KiB, respectively.

Table 5.6.: File sizes distribution in the `govdoc`-corpus (min size is 1 KiB).

File size (KiB)	≤ 4	≤ 16	≤ 64	≤ 256	≤ 1024
Amount (%)	5.40	20.71	52.54	75.82	90.60

Any tree vs. binary tree. The size and the height of a tree depends on their degree x. For instance, if we assume a database $\mathbf{S}$ with 10,000 entries, then for choosing a binary tree, $x = 2$, we get a total number of $h(\mathbf{\Psi}) - 1 = \lceil \log_2(10,000) \rceil - 1 = 13$ levels. The complexity of looking up a single digest reduces from 10,000 comparisons to 14 (at most). If the digest is not found in the root node, it is dropped immediately. In case there is not enough memory to handle a binary tree of 13 levels, we can set an upper limit. Thus, the leaves do not contain a single FI but a bucket with multiple FIs, e.g., setting an upper limit of 10 levels results in $\lceil 10,000/x^{L-1} \rceil = \lceil 10,000/2^{L-1} \rceil = 20$ FIs per bucket. In addition, it is possible to increase x, e.g., setting $x = 4$ reduces the number of levels to $\lceil \log_4(10,000) \rceil - 1 = 6$.

In order to handle such a tree data structure and its huge Bloom filters, we require powerful hardware. However, we argue that nowadays hardware is not as expensive as it

---

[7] `http://digitalcorpora.org/corpora/files` (last accessed 2014-03-07).

was and that companies working in prosecution have servers with hundreds of gigabytes or even a terabyte of RAM.

Setup. Based on these findings, we decided for an underlying dataset of 256 GiB and an average file size of 512 KiB. Recall, we apply approximate matching in the context of blacklisting. We claim that blacklists are far smaller than whitelists and thus 256 GiB is reasonable (e.g., the whitelist of NSRL contains about 2 TiB of unique files). While emphasis is put on this single setting, which we consider as representative, the concept can be easily adapted to any other setting. In addition, it is important to note that the binary tree is precomputed and, hence, its generation time is irrelevant.

We restrict our analysis to focus on binary trees where leaves correspond to single *FIs*. Assuming an average file size of 512 KiB, the number of files is $^{256 \cdot 2^{30}}/_{(512 \cdot 2^{10})} = 2^{19} = 524,288$. Thus, the binary tree has 19 levels and the size of the Bloom filter at the root m_1 is 2 GiB. Overall, the tree requires $19 \cdot 2 = 38$ GiB of memory. An overview of the relation between average file size and tree height is given in Table 5.7.

Table 5.7.: Height of the binary tree with respect to the average file size ω.

ω in KiB	32	64	128	256	512	1024
$h(\mathbf{\Psi})$	22	21	20	19	18	17

Further, we presume an HDD having around 450,000 files with a total file size of 200 GiB which corresponds to an average file size of 466 KiB.

Memory requirement. As mentioned earlier, see Sec. 5.4.1, the required amount of memory depends on the amount of data and the number of files, z. Based on the overall size of the data and the configuration of the employed tree data structure $\mathbf{\Psi}$, we can define the size and number of required Bloom filter nodes, σ. The number of files defines the height of the tree, $h(\mathbf{\Psi})$. For the considered setting, i.e., the application of a binary tree data structure where each leaf points at a single *FI*, diverse sample configurations are summarized in Table 5.8.

As already mentioned, any restriction to memory can be handled by utilizing buckets of *FI*s at leaves or extending the degree x of the tree data structure. Both of these actions will reduce the amount of required Bloom filters and, hence, the overall memory requirement.

Comparison efficiency. This subsection studies the comparison efficiency of the traditional approach and the new one. We decided to focus on the bit comparisons of each proceeding.

Traditional lookup: Recall, a similarity digest may consist of several 'small' Bloom filters (mostly each 256 bytes). In order to compare two digests, all filters of digest A have to be compared against all filters of digests B. Consequently, doing an all-against-all

Table 5.8.: Correlation between data size and average file size. First value is $h(\boldsymbol{\Psi})$, second value is total size of the binary tree in GiB.

Avg. file size	**Data size**				
	128 GiB $m_1 = 1$	256 GiB $m_1 = 2$	512 GiB $m_1 = 4$	1024 GiB $m_1 = 8$	2048 GiB $m_1 = 16$
32	22 / 22	23 / 46	24 / 96	25 / 200	26 / 416
64	21 / 21	22 / 44	23 / 92	24 / 192	25 / 400
128	20 / 20	21 / 42	22 / 88	23 / 184	24 / 384
256	19 / 19	20 / 40	21 / 84	22 / 176	23 / 368
512	18 / 18	19 / 38	20 / 80	21 / 168	22 / 352
1024	17 / 17	18 / 36	19 / 76	20 / 160	21 / 336

comparison of similarity digests equals an all-against-all comparison of all Bloom filters and thus of all bits. Hence, we need to know how many Bloom filters of 256 bytes exist.

`sdhash` roughly compresses 10 KiB of the input into a 256 bytes Bloom filter. Therefore, the 256 GiB reference dataset results in $256 \cdot 2^{20}/10 = 2^{24.68}$ filters and the analyzed data corresponds $200 \cdot 2^{20}/10 = 2^{24.32}$ filters. To sum it up, we have to compare around $2^{49.00}$ Bloom filters each consisting of 2^{11} bits (256 bytes). In all we have to compare 2^{60} bits in total.

`mrsh-v2` approximately compresses 25 KiB of the input into a 256 bytes Bloom filter. Thus, we have $256 \cdot 2^{20}/25 = 2^{23.36}$ filters for the reference dataset and $200 \cdot 2^{20}/25 = 2^{23.00}$ for the HDD. Overall, it requires to compare $2^{46.36}$ Bloom filters or $2^{57.36}$ bits.

Proposed approach: On page 72 we correlated the input size and the amount of elements or features by $n = \mu \cdot 2^{14}$. This means that 200 GiB of data equal $n = 200 \cdot 2^{10} \cdot 2^{14} = 2^{31.64}$ features. If we use $k = 5$, then each feature requires to compare 5 bits: $2^{31.64} \cdot 5 = 2^{33.96}$ bits. If we perform an 'all fragments comparison' where all of these exist in the database, these features have to be compared at all $19 = 2^{4.25}$ levels which comes to a total of $2^{38.21}$ bits that have to be compared. This represents the worst case where all fragments are found. In the best case (no fragment is found), we only require $2^{(31.64+33.96)/2} = 2^{32.80}$ bit comparisons (i.e., on average we compare 2.5 bits out of the 5 to the Bloom filter). Additionally, the similarity digest comparison is more expensive as after comparing two similarity digests, the amount of 1s needs to be counted. If we assume a uniform distribution of matches (which is rather unlikely as blacklisted files will occur less frequent), we have to compare $(2^{38.21} - 2^{33.96})/2 = 2^{37.13}$ in the average case.

5.4.5. Special use case: only a single Bloom filter

In this section we present and evaluate the special case were we only have a single Bloom filter which allows a *file-against-set* comparison with a lookup complexity of $O(1)$ for a single digest (published in [17]). In contrast to general approximate matching, our approach can only answer the question "does this set contain a similar file to *file A*?" by

- **yes**, there is a similar file (but it cannot say which one), or
- **no**, there is no similar file,

which is sufficient in case of blacklisting. We obtain this benefit either at the cost of more hashing operations or requiring a lot of main memory.

Proceeding. The basic idea is to insert all features into a single Bloom filter instead of having multiple filters as the lookup complexity per filter is $O(1)$. More precisely, let S_B and S_D be two sets of digests. Traditionally (using cryptographic hash functions) an investigator possesses a database containing the elements of S_B (e.g., the blacklist). When he receives D (e.g., a seized device), he hashes all files to S_D and compares them against S_B. Note, the database S_B can be precomputed and hence its generation time is irrelevant.

Regarding our concept, there are basically two alternatives depending on the underlying hardware:

1. Alternative one is identical to the traditional procedure. That is, the Bloom filter is filled with the features of S_B in advance, meaning that we can neglect its generation time.
2. The second possibility assumes that S_B does not fit into the Bloom filter, but S_D does. In that case we turn the work flow upside down by filling the Bloom filter with S_D and compare S_B against it.

The difference between these two procedures is the overall time. While in traditional procedure (alternative (1)) only S_D needs to be processed, the second possibility also has to hash S_B as a precomputation step. In the following (1) is denoted by *best-case* and (2) by *worst-case*.

The reason why alternative (2) might be necessary is that it is not possible to load the Bloom filter for S_B into main memory. Hashing all files of a set into a single Bloom filter requires a large Bloom filter which has to fit into main memory due to efficiency reasons. Thus, the limiting source is the physically available RAM.

For instance, let us assume that $|S_B| = 1500\,\text{GiB}$ and $|S_D| = 200\,\text{GiB}$. As shown later, an everyday working station with 8 GiB RAM cannot handle a Bloom filter of S_B but can of S_D. Therefore, we suggest creating a Bloom filter out of S_D and comparing all files of S_B in a second step. It is obvious that both sets have to be hashed – it is not possible to create the database in advance.

To optimize (2), one may store a list of hash values of S_B instead of the files. Thus, the files are already hashed and the overall proceeding is almost as fast as (1). In addition, the compression is better as we only store a 256-bit (32 byte) hash for each 64-byte chunk.

Implementation details. To verify our findings, we released a tool called `mrsh-net` which is basically a modification of the latest `mrsh-v2` version. The implementation is very simple and only has two options:

-g generates the database and prints it to stdout. Usage:

```
./mrsh-net -g t5-corpus > dbFile
```

-i reads DB-FILE `dbFile` and compares DIR/FILE against it. Usage:

```
./mrsh-net -i dbFile t5-corpus
```

The final step is to compile it and run `make mrsh-net`.

The main change was the implementation of the FNV-1a 256-bit function which only consists of an XOR and the multiplication with the prime $2^{168}+2^{8}+0\text{x}63$. As the runtime efficiency is very important, the implementation of the multiplication is 'hardcoded', i.e., it is not trivial to change the prime number or extend it to 512-bit.

In order to speed up the implementation, one may manipulate the FNV implementation in `src/fnv.c`. The function `mulWithPrime2` is responsible for the multiplication with the 256-bit prime. However, in case of a small Bloom filter, we do not need the most significant bits and can remove them. For instance, setting the Bloom filter to 32 MiB and $k = 5$, we only need $\log_2(32 \cdot 2^{20} \cdot 8) \cdot 5 = 140$ bits. Thus, we can comment out lines 108-112, which then ignores the bits 160 to 255.

To adapt our prototype for a specific use case, the user can change the following configuration in `header/config.h`:

`SUBHASHES` - amount of sub-hashes, parameter k (default: 5).
`MIN_RUN` - minimal longest run, parameter r (default: 6).
`BF_SIZE_IN_BYTES` - Bloom filter size in bytes (default: $33\,554\,432 = 2^{25} = 32\,\text{MiB}$). It has to be a power of 2.

Experimental results for special use case

First, we analyze the general efficiency of different approaches. The second part relates `mrsh-net` with the longest common substring. The final part compares `mrsh-net` and `mrsh-v2` with each other. All the presented results are based on the *t5*-corpus.

For our testing, we used the default configuration of `mrsh-net`, with $k = 5, r_{min} = 6$ and a Bloom filter size of 32 MiB. The blocksize, i.e., the approximate length of a feature, is set to 64 bytes.

Table 5.9.: Database size and runtime efficiency of different algorithms.

	`sdhash`	`mrsh-v2`	`mrsh-net` worst	`mrsh-net` worst	`mrsh-net` best	F2S2	SHA-1
Database size	61.18 MiB	27.33 MiB	1.78 GiB	1.78 GiB	32.00 MiB	3.69 MiB	0.24 MiB
Generation eff.	178 s	53 s	123 s	77 s	123 s	221 s	24 s
Comparing	1281 s	1259 s	< 1 s*	< 1s*	< 1 s	< 1 s	< 1s
Total	1459 s	1312 s	246 s	154 s	123 s	221 s	24 s

*The comparison itself need less than a second, however, in the worst case DS needs the be hashed.

Efficiency in general. Let S_D denote the hashes of files from a device and let S_B denote database set (i.e., the blacklist). Traditionally the proceeding requires to hash all files in S_D and to compare the hashes against an 'existing database' of S_B files. Thus, this section focuses the general properties of the different approaches with respect to runtime efficiency and database size (compression).

The results are given in Table 5.9 whereby the details are discussed in the upcoming subsections. The first subparagraph analyzes the compression (row 1). Next, we explain the runtime of the algorithms (rows 2-4). The last subparagraph is an estimation for a large scale scenario to clarify the impact of non-indexing.

Columns 1 and 2 present the results for the original implementations of `sdhash` and `mrsh-v2`, respectively. In column 3 we show the results for the worst case which means that we do not have an underlying database. The following column also presents the worst case but we modified `mrsh-net` based on the default settings (less bits of the FNV hash are considered). For completeness we included the results for F2S2 and SHA-1 in the last two columns.

Database size. Let S_B be the t5-corpus. Then, this section shows the size of the corresponding database. In case `sdhash`, `mrsh-v2`, F2S2 and SHA-1 the database is trivial as the database is equal to the hashes.

Regarding `mrsh-net` there are two possibilities: worst vs. best case. The worst case describes the scenario where the database does not fit in RAM and hence a hash-database as such does not exist. The investigator needs to have the whole dataset available. In contrast, for the best case where sufficient RAM is available, the database is simply the Bloom filter.

To conclude, the size of the databases of `sdhash`, `mrsh-v2` and `mrsh-net` (best) are in the same order of magnitude and therefore only a weak assessment criterion.

Experimental generation efficiency. This section focuses on the time of hashing S_D and comparing it against the database of S_B (generating the database is neglected as it can be done in advance). As we are interested in the runtime only, we use t5-corpus as both S_B and S_D. Note, this results in 4457 × 4457 comparisons[8].

---

[8] We run the tools by `./tool -c D B` which compares both lists also there are duplicate comparisons, i.e., A against B and B against A.

The times are given in Table 5.9. Row 2 states that all algorithms perform well in hashing but are still slow compared to SHA-1. The problem is shown in row 3 where both Bloom filter approaches need an extremely long time for the all-against-all comparison. The last row only sums rows 2 and 3. Note, the 'worst-columns' constitute an exception. Admittedly the comparison takes less then a second, however there is no underlying database and thus S_B has to be processed. Therefore, row 4 contains two times the hashing time (as $D = B$).

One should keep in mind that the comparison has quadratic complexity and thus increases enormously when the number of files increases. In contrast, our new concept has a linear runtime as it only needs to hash the files.

Impact on runtime in large scale forensics. Based on the findings from before, this section estimates the efficiency for a real life scenario. We used the numbers from Table 5.9 and calculated the upcoming ones for a larger use case where we pick up the example from the beginning of this section and assume 200 GiB of seized data and a database of 1500 GiB[9].

The results are given in Table 5.10. The estimated database size is given in row 1 where in the best case `mrsh-net` needs the less space, however, it should be kept in RAM. Row 2 calculates the approximate hashing time by multiplying $^{200}/_{1.78}$ (we have to process 200 GiB instead of 1.78 GiB). The last row assesses the comparison time. For instance, `sdhash` needed 1281 s for comparing $1.78 \times 1.78 = 3.17$ GiB of data. As this sample requires to compare $200 \times 1500 = 300,000$ GiB of data, we estimate the overall time by $^{300,000}/_{3.17} \cdot 1281$ s.

Again, there are two possible scenarios with `mrsh-net`. In the worst case, we hash the 200 GiB to the Bloom filter and then process the 1500 GiB. As the comparison 'costs nothing', `mrsh-net` has to hash 1700 GiB which is $^{1700}/_{1.78} \cdot 123\,\mathrm{s} = 117,471\,\mathrm{s}$ (approx 32 h). In the best case we have a powerful station that can hold the Bloom filter for 1500 GiB data in RAM (approximately 16 GiB of RAM are needed). Thus, we only need to process the 200 GiB which takes 227 min.

Table 5.10.: Estimated runtime for a sample use case.

	`sdhash`	`mrsh-v2`	`mrsh-net` worst	`mrsh-net` best
Database size	49.79 GiB	22.22 GiB	1500 GiB	16 GiB
Hashing	329 min	98 min	227 min	227 min
Comparing	3.84 years	3.77 years	32.63 h	< 1 min

Precision & recall on base of the longest common substring. The current version of `mrsh-net` decides between match and non-match based on the longest run. Hence, this section focuses on the relation between `mrsh-net` and the longest common substring.

---

[9] The current NSRL of NIST contains about 2 TiB of unique data, hence this a realistic size.

Due to the complexity, we build our ground truth on the approximate longest common substring which is briefly described in Sec. 6.2.4. The basic idea of the approximate longest common substring metric (aLCS) is not to compare files byte by byte but rather block by block.

To calculate the precision and recall rates, we perform an all-against-all-other comparison and use the following simplified notation. $mrsn(f, BF)$ compares file f against the Bloom filter bf and returns the longest run. $aLCS(f, GT)$ returns the longest aLCS score for f in the ground truth GT.

According to this, we define the true positive (TP), false positive (FP), true negative (TN) and false negative (FN) as follows:

TP: $mrsn(f, bf) \geq r_{min}$ and $aLCS(f, GT) \geq r_{min} \cdot bs$.
FP: $mrsn(f, bf) \geq r_{min}$ and $aLCS(f, GT) < r_{min} \cdot bs$.
TN: $mrsn(f, bf) < r_{min}$ and $aLCS(f, GT) < r_{min} \cdot bs$.
FN: $mrsn(f, bf) < r_{min}$ and $aLCS(f, GT) \geq r_{min} \cdot bs$.

where $r_{min} \cdot bs = 6 \cdot 64 = 384$ bytes.

Positives. Our comparison returned 2555 positive matches with a true positive rate of 99.3% and a false positive of 0.7%. Reviewing the false positives, all but one of the longest run lr do not exceed 9 which means that they are very close to our threshold.

In addition, we studied the distribution of the aLCS scores d_{aLCS} relative to $r_{min} \cdot bs$ for the false positives where

$$d_{aLCS} = \left\lceil 100 \times \left(1 - \frac{aLCS(f, GT)}{384}\right)\right\rceil, d_{aLCS} \in \mathbb{N}.$$

This shows how close the false positives are to the threshold of 384. The results are given in Table 5.11. For instance, over 60% have an aLCS score above 30% (=269 bytes). To sum it up, although these are false positive, they are close to the thresholds.

Table 5.11.: Empirical probability distribution function (*pdf*) and cumulative distribution function (*cdf*) for d_{aLCS}.

X	10	20	30	50	70
$P\{d_{aLCS} = X\}$	0.1111	0.2778	0.2222	0.1111	0.0556
$P\{d_{aLCS} \leq X\}$	0.1111	0.3889	0.6111	0.8889	0.9444

Next, we consider the relation between the longest run and the aLCS score. In other words, we expect that the longest run lr multiplied by the blocksize bs is greater or equal the aLCS score, i.e., $lr \cdot bs \geq aLCS$. According to that, we adapt the configuration from the beginning of this section and changed $r_{min} \cdot bs$ to $lr \cdot bs$. Thus, the new true positive setting is:

TP: $mrsn(f, bf) \geq r_{min}$ and $aLCS(f, GT) \geq lr \cdot bs$.
...

In this case, the detection rates worsen and fall down to a true positive rate of 92.3% and a false positive rate of 7.7%. Again, we consider the distribution for the aLCS scores in Table 5.12. As we can see, over 75% vary by less than or equal to 30% and we rate these results as still acceptable.

Table 5.12.: Empirical *pdf* & *cdf* for d_{aLCS} for the relation between longest run and aLCS score.

X	10	30	50	70	100
$P\{d_{aLCS} = X\}$	0.3214	0.2296	0.0714	0.0051	0.0051
$P\{d_{aLCS} \leq X\}$	0.3214	0.7551	0.9235	0.9796	1.0000

Negatives. Obviously the negatives are $4457 - 2555 = 1902$ which can be broken down into 77.1% true negatives and 22.9% false negatives. Having a closer look at these very high false negatives, we observe that most aLCS matches are based on low entropy sequences. In other words, the high aLCS scores between some files are based on long runs of zeros only, i.e., the entropy of the substring is $e = 0$, or runs of with a lot of zeros, e.g., the entropy of the substring is $0 < e < 3$. Thus, Table 5.13 shows the impact of considering aLCS sequences with a higher entropy.

Nevertheless, false negatives are not so much relevant. For instance, with respect to blacklisting, these files remain unclassified and an investigator has to analyze them manually. Hence, false negatives are considered during a further investigation.

Table 5.13.: Distribution of false negative with respect to entropy.

entropy	> 0	> 1	> 2	> 3
TN	78.5%	82.3%	86.4%	91.2%
FN	21.5%	17.7%	13.6%	8.8%

Precision & recall rates compared to mrsh-v2. This section compares the relation between `mrsh-v2` and `mrsh-net`. As both are based on the same procedure, we expect that both implementations yield similar results. In other words, comparing a file f against database DB, both algorithms should either output a match or a non-match. Thus, we define the following rates:

TP: $mrsn(f, bf) \geq r_{min}$ and $mrsh(f, DB) \geq 1$.
FP: $mrsn(f, bf) \geq r_{min}$ and $mrsh(f, DB) = 0$.
TN: $mrsn(f, bf) < r_{min}$ and $mrsh(f, DB) = 0$.
FN: $mrsn(f, bf) < r_{min}$ and $mrsh(f, DB) \geq 1$.

Positives. Regarding the 2555 positive matches from `mrsh-net`, 92.1% are true positives and also identified by `mrsh-v2`. The false positive rate is therefore at 7.9%. Comparing the false positives against the aLCS showed that in fact only 3.6% (out of the 7.9%) are really false positive. On the other side, Table 5.14 shows the distribution of the longest run for the false positives. Most of them are close to the longest run threshold 8.

To conclude, the results are slightly different, however, the `mrsh-net` shows a finer granularity as in fact these are not false positives but true positives.

Table 5.14.: Empirical *pdf* & *cdf* for longest run lr.

X	10	15	20	30	50
$P\{lr = X\}$	0.1095	0.0200	0.0200	0.0200	0.0050
$P\{lr \leq X\}$	0.4925	0.7363	0.7960	0.9665	0.9950

Negatives. The negatives yield a 61.8% true negative rate and a 38.2% false negative rate. Recall, false negative means that `mrsh-net` does not identify a match while `mrsh-v2` outputs a score greater 0. In other words, `mrsh-v2` identifies a positive.

Thus, we first compared the `mrsh-v2` results against aLCS. In fact, almost 70% percent of these matches are based on files that share less than 384 bytes which have no false negatives for `mrsh-net` (our setting aims at having more than 384 bytes). Regarding the remaining 30%, most of the matches are again based on a low entropy, e.g., over 75% have $e < 3$.

To conclude, the algorithms do not coincide very much with respect to negatives.

5.5. File fragment detection on network traffic

In this section we show how to use approximate matching to detect similar files in network traffic. Compared to existing techniques, our approach is straightforward and does not need a comprehensive configuration. It can be easily deployed and maintained as only fingerprints (a.k.a. similarity digest) are required, unlike providing verbose rules or machine learning. The results are not published yet but submitted [10].

There are two challenges in order to apply approximate matching approaches to network traffic. First, algorithms have to be optimized to handle fragments of MTU size. In contrast to their original purpose (handling inputs of kilobytes, megabytes or gigabytes), they have to be optimized to handle fragments where each has less approximately 1460 bytes. Second, we need a fast lookup strategy for the similarity digests which is solved by the findings from Sec. 5.4.5.

5.5.1. Foundations

This section explains the foundations and presents related literature. First, we briefly present the network basics. Next, we discuss the different levels of network packet analysis. The current techniques for data leakage prevention systems are explained in the following.

Network basics. Network traffic can be analyzed on different levels which result from the network architecture, the TCP/IP reference model. This model was proposed by Cerf and Kahn in 1974 [31] and divides the network into four layers named application, transport, Internet (IP) and link layer.

In order to send information over the Internet, the payload is split into chunks of 1460 bytes and each layer has a specific header prefixed to it [97]. This size results from the maximum transmission unit (MTU, [71]). MTU is the maximum packet size that can be handled by the IP-layer and is 1500 bytes for Ethernet traffic.

Literature often uses the terms 'segment' for TCP traffic units on transport layer, 'datagram' for units on the internet layer, and 'frame' for the unit of a link layer. The term packet is a more generic term and can be used for units on all layers. We will use the term 'network packet' as a synonym for datagram.

Packet inspection. There are several approaches for firewalls to filter and analyze packets on the network which will be discussed in the following.

The simplest one is *static packet inspection* from 1988 which treats each packet as 'stand-alone' and decides on information contained in the packet headers. Common rules are based on destination IP, source IP, ports and protocol. An improvement is *stateful packet inspection* (SPI) which "shares many of the inherent limitations of the static packet filter with one important difference: state awareness. [...] The typical dynamic packet filter is aware of the difference between a new and an established connection" [98, p77++]. Hence, the firewall maintains a table to be aware of any connection.

The next step in the evolution is *deep packet inspection* (DPI). Besides analyzing packet headers, it examines the actual payload. "DPI engines parse the entire IP packet, and make forwarding decisions by means of a rule-based logic that is based upon signature or regular expression matching. That is, they compare the data within a packet payload to a database of predefined attack signatures (a string of bytes). [... However,] searching through the payload for multiple string patterns within the datastream is a computationally expensive task" [72].

Techniques for data leakage prevention. According to [58], organization's data can be classified into three states: a) *Data in Motion* (DIM): data in the process of being transmitted over the network, b) *Data at Rest* (DAR): data in file systems, FTP server, and c) *Data in Use* (DIU): data at an network endpoint, like desktop or USB device. *Data Loss Prevention Systems* (DLPS) is a mechanism that identifies sensitive information by content in DIM, DAR or DIU, and prevent its leakage to outside of the organization

The main idea behind these products is to use deep packet inspection (DPI) for automatic network analysis. In other words, it tries to detect protected information or files within the network traffic. According to [88], there are the following approaches:

Regular expressions are effective in case of structured data like credit card numbers or social security numbers, however, they fail in case of file identification.

Database fingerprint analyzes network packets for exact strings. Hence, instead of looking for all credit card numbers, one may only look for specifics ones. In addition, it is very common to identify documents that are tagged with buzzwords like ‘confidential’ or ‘secret’, however, this approach fails if buzzwords are omitted or in case of binary data.

Exact file matching uses hash functions to find exact matches which is file type independent. However, it is trivial to evade (alter at any position). Another drawback is that all packets needs to be captured, the files need to be rebuilt, then hashed.

Statistical analysis is based on machine learning approaches. This approach includes a comprehensive training in the beginning and only works reliably for a large corpus. In addition, if new protected data is added, the training needs to be re-started. Furthermore, this approach is prone to false positive and false negatives.

Besides these four mentioned techniques, there is *conceptual/lexicon* which is a combination of rules, directories and other analyzes[10] and *categories* which is also based on rules and dictionaries.

The most promising approach for automatic file identification is *partial document matching* which looks for complete or partial matches (e.g., a few sentences of a document) on protected content. This technique often uses a rolling hash to compare documents against network packets payload. Unfortunately we could not find any information as most DLPS are commercial and therefore closed source.

5.5.2. Solution and proceeding overview

In order to handle network packets, algorithms need a finer granularity and we reduce the blocksize from `mrsh-v2` from the original 160 bytes to 64 bytes which results in more features. However, a finer granularity increases the chance for false positives as the decision is based on less data. That is why we deploy a filter mechanism that eliminates non-relevant chunks from consideration, e.g., long runs of zeros.

To overcome the lookup complexity, we decide to have only one single Bloom filter which overcomes the drawback of existing approaches (all details are given in Sec. 5.4.5).

According to that, the overall process requires two phases:

1. *Database generation:* Divide file into chunks, filter out non-relevant chunks, hash chunks and fill Bloom filter.
2. *Network packet analysis:* Divide packet into chunks, hash chunks and compare against Bloom filter.

Note, the filter mechanism is only necessary for creating the Bloom filter. During the network analysis phase the packets are hashed and compared only. To sum it up, no matter how complex the filter mechanisms are, the performance of the network analysis is not influenced.

Having only one Bloom filter comes with two downsides. The first of these is that we can only monitor a finite set of files. For the sake of efficiency, the Bloom filter has to be completely ‘in memory’ and therefore the limiting source is the physically available

---

[10] As this technique is very complex, we like to refer to [88, p9].

memory. However, as shown later, our approach only requires 32 MiB of memory to monitor 2 GiB of data; 100 GiB of data requires about 2 GiB of memory. We claim that in the case of most mid-size business the data will not exceed 100 GiB when dealing with office documents, source-code and blueprints (images).

The second disadvantage is that our approach is a *packet-against-set* comparison. That is, the answer of a packet-query is either yes (there is a similar file to that packet in the set) or no. In order to receive the exact file, further analysis is needed. Nevertheless, we claim that in case of data leakage prevention or virus identification this is sufficient. It is not relevant to know the file name, it is more important to have a yes or a no and cancel the connection.

Additional benefits. One aspect which might not be immediately obvious is privacy. Due to the usage of Bloom filters, the monitored data is stored in a preimage resistant format (although we use the non-cryptographic hash function FNV). Hence, it is possible to maintain the sensitive data at a central point, fill the Bloom filter and distribute it while no information leaks. For instance, anti virus vendors may provide a database containing the newest malware.

Another point is that Bloom filters can be easily combined with each other by ORing both filters which allows do update filters. In case of counting Bloom filters one may even remove elements.

Feature filter. As mentioned, not all features are of the same quality. In order to decide if a chunk is important or not, we consider two values called ‘entropy’ and ‘randomness’. While the former is based on the well-known Shannon entropy [89], randomness is a value developed for this project and considers two neighboring bytes. If two consecutive bytes are equal or differ by one, the anti-randomness is increased. More formally, let a chunk of length L be given where B_i denotes the byte at position i. Then anti-randomness R is calculated as follows:

$$R = \sum_{i=0}^{L-2} ar(B_i) \text{ where } ar(B_i) = \begin{cases} 1, & \text{if } |B_i - B_{i+1}| \leq 1 \\ 0, & \text{else} \end{cases}$$

A chunk has sufficient randomness if $R \cdot 2 < L$.

Implementation details. To verify our findings, we released a prototype called `mrsh-net` which is basically a modification of the latest `mrsh-v2` version. Currently there is only one branch and thus a lot of testing-code-pieces are included which makes the code harder to understand as it is. For instance, we implemented counting Bloom filters which were necessary for testing purposes, and research into these is ongoing research.

The upcoming subsections briefly explain the functionality and the some implementation details. The prototype can be downloaded on our website[11].

Commandline arguments

---

[11] `http://www.dasec.h-da.de/staff/breitinger-frank/#downloads` (last accessed 2014-03-13).

-g generates a Bloom filter from DIR and prints it to std. Usage:
`./mrshnet -g DIR/* > dbFile`.

-i reads Bloom filter BF-FILE and compares DIR/FILE against it. Usage:
`./mrshnet -i BF-FILE DIR/FILE`.

-f generates the ‘false positive’ matches for a list of files.
`./mrshnet -f DIR/*` .

-e sets the minimum entropy a chunks needs to have. Usage:
`./mrshnet -e 2.8 -g DIR/* > dbFile`.

-t excludes a file type for the -f option.

‘False positive’ means that all files are added to the Bloom filter, file f is removed (we implemented counting Bloom filters for testing purposes) and finally f is compared against the filter. If combined with -t, the filter excludes all files of -t type and compares them against it. In both cases matches are printed to stdout.

5.5.3. Test methodology

All tests run on ‘simulated’ network traffic where simulated means that all files are split into 1460 byte sequences. This size results from the MTU size of 1500 bytes minus 20 bytes IP header and minus 20 bytes TCP header. On real network traffic it is easily possible to skip 40 bytes in the beginning and thus we claim this simulates real life circumstances. Our experiments are divided into two parts based on the input – synthetic and real world data.

The rest of this section is divided as follows. First, throughput is discussed which is an essential property for all kind of algorithms. Next, we explain our test-setup using synthetic data. In the last section, we describe our opinion on similarity and explain the testing of real world data.

Throughput. A fundamental property of network analysis tools is throughput – huge delays are not acceptable. According to Sec. 5.4.1, the throughput considers phase 2. We neglect the database generation process as it is independent from the network analysis.

Our assessment is based on our C-implementation on a usual workstation. We capture the time for processing a given set of files, i.e., reading, hashing and comparing. Note, there is neither a hardware implementation of our algorithm nor are we programing experts. However, there exist ideas of how to build high-throughput hardware implementations for Bloom filters e.g., [38].

Random data. This controlled test utilizes `/dev/urandom` to create a pseudo random file set. Thus, it is possible to distinguish between true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). In order to have a real life distribution of file sizes, we took the t5-corpus as a model and created 4457 files with identical sizes which is in total 1.78 GiB.

To determine the detection rates, we did the following tests:

TP: Fill all files from the corpus to the Bloom filter. Next, we compare all packets against the Bloom filter. All identified packets are true positive. (This can be solved using the `-g` and `-i` options).

TN: All files are added to the Bloom filter. Next, we remove file f from the Bloom filter and compare f against it (solved by `-f` option).

FP: 1-TN.

FN: 1-TP.

As approximate matching algorithms are working perfectly on random data [24], we expect a TP rate close to 100% and a TN rate of exactly 100%. The TP rate will most certainly differ from 100 as the last packet of a file might be too small and does not produce a longest run of $r \geq r_{min}$.

Similarity. Before describing the test methodology for real world data, we describe our understanding of similar and dissimilar. According to our point of view, a

TP: means that two files share a significant and interesting amount of data, e.g., same text passages, pictures or copyright information.

FP: means that two files have nothing in common or only unrelated information, e.g., file headers in common, but they are matched.

TN: means that two files have nothing in common.

FN: means that two files are not matched although they share significant and interesting data.

According to this, it is necessary to analyze the general communality between files, i.e., which data is file type specific and not related to the actual content. Possible samples are common headers, long runs of zeros, or file structure information. This is the input for creating useful filters.

Since there is no ground truth available, it is hard to categorize a match. To classify the positives (true positive + false negatives) it is necessary to assess all obtained matches. In order to verify the negatvies (true negative + false negative) we have to evaluate the whole corpus. Thus, we mainly focus on positives.

Recall, the output of our procedure is not an exact match but a statement that there is a similar file in the underlying set. Thus, it is necessary to compare all positive matches against the underlying file set. In order to handle this, we apply `sdhash` to identify ‘possible true positives’ which then are partly inspected manually.

Result presentation. When assessing our results we consider the packet and the file level. The *packet level* describes the relation between all sent packets and matched / non-matched ones which is especially important for false positives. However, with respect to true positives, we suggest the *file level* which requires that at least one packet matches. For instance, two large office documents that share only one small table will not have many packets in common but are similar and the graphic is monitored content.

Cross matching file types. The first test examines the detection behavior among different file types. Let Y denote the file type, and let SET_Y be all files of type Y. Then, DB_{-Y} is a database that contains all files of $t5\backslash SET_Y$, i.e., all files except the files of type Y. Furthermore, let our concept be denoted by $mrshnet(SET, DB)$ which is a function that returns all files in SET that matches the database DB. According to that, cross matching runs $S = mrshnet(SET_Y, DB_{-Y})$ for all file types Y. The output of a run is a set S which contains all cross matched files.

To distinguish between TP and FP, we compared SET_Y and $t5\backslash SET_Y$ by `sdhash` and received a list of possible matches. Next, we manually proofed the similarity starting with the best matches, i.e., if a file is matched with two files, we consider the higher score first.

For instance, set $Y = doc$. Then, all *.doc files are compared against the DB_{-doc}. The result could be a set like $S = \{f1.doc, f2.doc, f3.doc\}$ which serves as input for `sdhash`. Here, the output is a list like

```
file1.doc matches fileA.ppt (50)
file2.doc matches fileB.xls (10)
...
```

According to this list, we picked out matches and compared them manually, i.e., open file1.doc and fileA.ppt and compare them. The selection process mostly focuses on borderline matches. For instance, two files yielding a `sdhash` score of 50 are most likely similar whereas two files having a score of 5 require manual inspection.

The motivation for this test is the expected number of positive matches should be small. We mostly assume hits between office documents or between HTML and TXT.

True positives. We have introduced techniques to minimize the chance of false positives called entropy and randomness. In general, these approaches filter out chunks in advance so that we come closer to random data and do not make decisions based on long runs of zeros.

Within this test we study the impact of different configurations with respect to the true positives. Although chunks are filtered out, there should be a high true positive rate on both levels, file level and packet level.

Mixed file types. For this test we randomly selected 100 files out of the t5-corpus which serve as ‘monitored files’ and hashed to DB. The remaining 4357 files are used to produce the traffic. In contrast to cross matching, there are also comparisons among equal file types.

5.5.4. Experimental results & assessment

For the real world data test we choose the t5-corpus as described in Sec. 2.6 on page 17. Moreover, we used the following configuration: $k = 6, r_{min} = 8$ and $m = 2^{28}$ bits (32 MiB). The blocksize is set to 64 bytes, i.e., the approximate length of a chunk.

Throughput. To determine the throughput we processed the t5-corpus which has exactly 1823.11 MiB (= 1.78 GiB) . The measured result includes the time for reading the input from disk, splitting it into packets, processing (hashing) it and performing the Bloom filter comparison. The time was measured by the Linux `time`-command where we selected the user time: 23.474 s. Overall, this comes to $^{1823.11\,\mathrm{MiB}}/_{23.474\,\mathrm{s}} = 77.67\,\mathrm{MiB/s}$ which corresponds $77.67\,\mathrm{MiB} \cdot 2^{20} \cdot 8 = 651,501,914$ bits/s; approximately 650 Mbit/s.

The test was performed on a 2 GHz Intel Core i7 CPU, single threaded. However, the concept allows an easy parallelism* of the complete approach without any synchronization. There are only requests if specific bits are set to one; the underlying Bloom filter does not change.

Analyzing the bottlenecks of the code showed that there are two time consuming parts which are accessed for each byte: the rolling hash and FNV hash calculation[12].

The rolling hash is based on 2 subtractions, 5 additions, 1 multiplication, 1 shift and 1 XOR. Thus, on the one hand it could be possible to create a more efficient assembler version. On the other hand, it might be possible to run this on dedicated hardware as there are only a few operations.

The FNV hash function consists of one multiplication and one XOR. We implemented this using 8 32-bit integers and stored the result of each multiplication in a 64-bit integer. Of course, we considered the simplicity of the prime to have only few multiplications. Nevertheless, we are sure this can be optimized.

Detection rates on random traffic. This test runs on synthetic data which was captured from `/dev/urandom` and imitates the t5-corpus – we created 4457 synthetic sequences having the same sizes as the files in the corpus.

The results are as expected. On the packet level the true positive rate is at 99.6% and the true negative rate is at 100.0%. The few false negatives result from the length of the last packet which could be very small and thus contain less than 8 runs.

File analysis and similarity. This section explores the kind of basic-communality between files based on the statements from the paragraph titled *Similarity* from page 91. According to this, we rate a match as a false positive if two underlying files only share common headers, long runs of zeros, or file structure information.

For this test we randomly selected files from the t5-corpus, correlated them by hand to ensure they are dissimilar and then compared them automatically with respect to the

---

[12] Note, the entropy calculation is also very time consuming. However, this is only necessary when creating the Bloom filter and not while the comparison process.

longest common subsequence. We decided upon the longest common sequence as it is very similar to our approach – we require a minimum run length.

Basically we made two observations which yield long runs of non-significant data / information. In other words, possible false positives according to our definition.

The first kind of strings are low entropy sequences which are very common throughout most types. Especially Microsoft Office documents share long common low entropy sequences, but also types like GIF or JPG. Note, some files are more like container which can embed other types, e.g., office documents can have embedded pictures. This hex-dump is a sample from a GIF.

```
0000010: ffff ccff ff99 ffff 66ff ff33 ffff 00ff
0000020: ccff ffcc ccff cc99 ffcc 66ff cc33 ffcc
...
0000200: 6633 3366 0033 33ff 3333 cc33 3399 3333
0000210: 6633 3333 3333 0033 00ff 3300 cc33 0099
```

Besides low entropy, we also discovered long runs of 'non-random' sequences which have a normal to high entropy. The following hex-dump is a sample from a PPT.

```
00a1d80: 5657 5859 5a63 6465 6667 6869 6a73 7475
00a1d90: 7677 7879 7a83 8485 8687 8889 8a92 9394
...
00a1dc0: cad2 d3d4 d5d6 d7d8 d9da e1e2 e3e4 e5e6
00a1dd0: e7e8 e9ea f1f2 f3f4 f5f6 f7f8 f9fa ffc4
```

Finally, we studied office documents to check to see if they share another 'base-similarity' e.g., an empty DOC-file has a size of ≈ 21 KiB. An example of a common chunk is the following hex-dump which was fund in many office documents. However, this would be filtered out due to too less entropy and randomness.

```
001a600: 0100 0000 0200 0000 0300 0000 0400 0000
001a610: 0500 0000 0600 0000 0700 0000 0800 0000
...
001a8f0: bd00 0000 be00 0000 bf00 0000 c000 0000
001a900: c100 0000 feff ffff c300 0000 c400 0000
```

We argue that these kinds of chunks represent non-relevant information for an investigator and thus packets comprised of these chunks can be safely neglected. To sum it up, it is reasonable to apply filter mechanism like those described in paragraph *Feature filter* on page 89.

Detection rates for cross matches. This section analyzes the cross matches and thus the behavior of $mrshnet(FILES_Y, DB_{-Y})$ where in total 1,311,576 packets have been sent.

A summary of our findings is given in Table 5.15 which shows the matches on the packet and the file level. For instance, row 1 states that in total 10,288 packets matched the database and belonged to 809 different files. Since the entropy is 0.0, all packets are

Table 5.15.: Impact of introducing an entropy.

	JPG	GIF	DOC	XLS	PPT	PDF	TXT	HTML	Packet level		File level	
e			(matches on packet level / matches on file level)						matches	SDM	matches	SDM
0.0	133/53	17/17	2386/233	1249/117	5251/335	1118/29	48/8	86/17	10288	66.8%	809	21.5%
0.0*	0/0	5/5	1147/31	504/29	2984/227	932/14	38/6	77/12	5687	87.9%	324	31.8%
2.4	131/53	14/14	1573/118	674/60	3753/242	1018/53	40/8	85/17	7288	80.8%	538	32.3%
2.4*	0/0	0/0	1017/26	191/25	2654/73	930/14	38/6	77/12	4907	94.8%	156	47.4%
2.8	131/53	0/0	1136/35	148/5	2856/72	968/20	39/7	83/17	5361	96.7%	209	74.2%
2.8*	0/0	0/0	961/17	10/6	2497/22	927/14	38/6	77/12	4510	99.0%	77	67.5%
3.2	130/53	0/0	1124/35	148/5	2837/72	960/20	39/7	82/17	5320	96.7%	209	74.2%
3.2*	0/0	0/0	955/17	8/6	2479/22	925/14	38/6	76/12	4481	99.1%	75	67.5%

* Here the randomness check is turned on.

considered. SDM is the `sdhash` match rate which expresses that `sdhash` also identified similarity when comparing $FILES_Y$ against the monitored set, e.g., 66.8% of 10,288 packets have a high probability to be a true positive. Rows highlighted with a star are outputs were randomness mode is on, i.e., packets marked as non-random are filtered out.

When rating the settings we mostly considered the packet-file-ratio (abbreviated pf-ratio) which is relation between packet matches and file matches. For instance, a high pf-ratio (close to 1) like for GIF indicates that only single packets matched the database which is an indicator for less interesting results and vice versa. In the specific case of GIF, all files included the same, low entropy sequence that matched the database.

Regarding the table, an entropy over 2.8 with the randomness test seems to be promising. In total there are 4510 packets where the SDM rate is 99.0%. It is obvious that the higher the required entropy, the less overall matches.

In the following we consider the matches for row *2.8**. In total, 4510 packets matched which belong to 77 different files. Out of these 4510 packets 99.0% coincide with the results of `sdhash`. If we consider all `sdhash` hits as true positives, there are 45 false positives (out of the total 1,311,576 packets). Thus, our approach yields a false alert rate of ${}^{4510 \cdot 0.01}/_{1,311,576} = 3.44 \cdot 10^{-5}$. In other words, every $\approx 10^5$ packet there is a false alert (Note, this is not the false positive rate).

To verify the `sdhash` results, we manually proofed some of the 52 file pairs (62.7% of the 77). Most matches where between TEXT and HTML or between the office documents. For instance, `mrsh-net` returned a match for 003344.doc which was correlated by `sdhash` to 003358.ppt. Examining these two files showed that both contain equal graphics which are listed in Fig. 5.11. We found true positives only.

Next, we reviewed those files where `sdhash` could not find a similar file. First, it was conspicuous that almost all hits were caused by a single packet which means files have only small communality. Since our implementation works deterministically (i.e., if A matches B, then also B matches A), we compared all remaining files with each other. In fact, we could detect file pairs that are similar. For instance, some HTML files where falsely decelerated as TXT files and had the cascading style sheet (css) included.

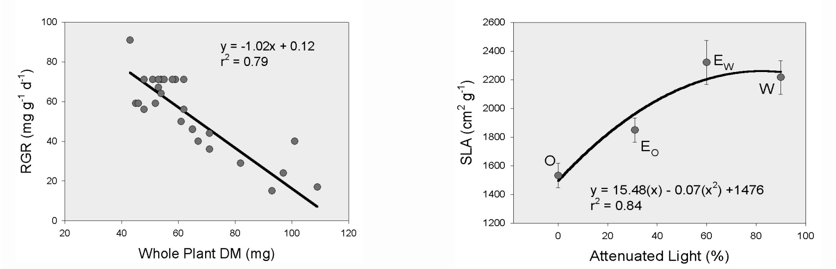


Figure 5.11.: Both files, 003344.doc and 003358.ppt, contained these two graphs and thus they are classified as a true positive.

Thus, these files have a similar layout and tables. It is questionable if this is relevant or non-relevant information. Assume an internal webpage which contains secret business numbers. Then, an employee could download it, update it with the latest numbers and mail it to someone. Other examples are translated documents which are still in the same layout. Actually, this depends on the scenario, and an admin needs to consider this when creating the database.

Nevertheless, even if we rate all these matches as false positives we obtain a false alert rate of $3.44 \cdot 10^{-5}$ (otherwise 0) for cross matching which is acceptable for a first prototype.

True positive analysis. Deploying filter mechanisms to reduce the false positive matches implies that true positives will also be reduced. Therefore, this section studies the impact of different settings on the true positive rate. The results are given in Table 5.16.

Table 5.16.: True positive rates for different settings sending 4457 files / 1,311,576 packets in total.

	0.0	0.0*	2.4	2.4*	2.8	2.8*	3.2	3.2*
pkt. (%)	91.9	82.0	81.2	80.1	76.4	75.9	69.4	69.2
files (%)	99.9	98.9	99.3	98.9	98.8	98.7	98.0	98.0

* Here the randomness check is turned on.

Equal to the random-test, there is no 100% true positive rate if both filtering mechanisms are turned off. The reason is the rolling hash. For instance, one undetected file was an almost empty Word document. The byte structure was composed of many zero runs and thus most packets contained zeros. The rolling hash is not able to determine enough trigger sequences and the run will not exceed the minimum.

Regarding our chosen setting 2.8*, about 1/4 of all packets are filtered out due to less entropy and randomness but we still detect 98.7% of all files. To conclude this section: monitored files should consist of a few kilobytes of data having an entropy over 2.8.

Table 5.17.: SDM rates for all file types.

Packet level		File level		File types					
matches	SDM	matches	SDM	DOC	XLS	PPT	PDF	TXT	HTML
883	97.6%	109	88.1%	10	1	11	59	2	26
731	99.5%	50	92.0%	10	1	11	/	2	26

Note, GIF and JPG are not listed because there were no matches.

Detection rates on mixed file types. This test simulates the case where the database contains mixed content. We randomly selected 100 files out of the t5-corpus which serve as the monitored files. The remaining files are used to produce the traffic. We capture all packets that provoke a match and performed analysis as in the previous sections.

In total we sent 4357 files producing 1,284,738 packets. The settings are based on our previous findings: $e = 2.8*$. A summary of the results is presented in Table 5.17. Note, GIF and JPG are not listed because there were no matches.

Row 1 shows the overall results. Overall 833 packets are positives with a `sdhash` match rate of 97.6%. It is conspicuous that PDF has so many matches. Studying the results shows that there are multiple false positives which seem due to overlapping structure information. In contrast to office documents and images, the structure information has a high entropy and overcomes our filter out techniques. Nevertheless, there are also near duplicates, e.g., 000740.pdf and 000743.pdf.

Row 2 considers the results without PDF. The 731 packets come to a `sdhash` match rate of 99.5%. Our manual review showed that all files are true positives (most of them are similar webpages like 002224.html, 002758.html). Hence, the false positive rate here is 0.

Considering the packet-file-ratio of both rows shows that on average a match of PDF is due to ${}^{883-731}/_{59} \approx 2.5$ packets. Further investigations reveal that most matches are due to the same byte sequence. In other words, the database contained one sequence which a lot of PDFs included. However, we could not find interesting similarity among these matches.

If we class all PDFs as false positives (which is an upper limit as we also had true positives), the overall false positive rate is ${}^{883-731}/_{1,284,738} = 1.1 \cdot 10^{-4}$.

5.6. Summary

This chapter presented algorithms, concepts and applications of approximate matching. The algorithm part is composed of three new algorithms named `bbHash`, `mvhash` and `saHash`. Additionally, we explained a concept for an efficient similarity digest database lookup in Sec. 5.4. The last part of this chapter described a new application for approximate matching based on network traffic.

`bbHash` is the first algorithm using an external data structure to create a similarity digest. Its settings aim at having a similarity digest length of 0.5% of the original file size. However, the performance is too slow for practical use which is future work. `mvhash` was the second algorithm. Due to three trivial phases majority vote, run-length encoding and Bloom filters we have several advantages compared to existing algorithms. A strong advantage of `mvhash` is high generation efficiency. Our approach is almost as fast as the SHA-1. Equal to `bbHash`, the compression ratio of 0.5%. The drawback is that the current version has some configuration options; each file type requires its own configuration, no standard configuration works for all file types. This is future work e.g., distinguish between file types automatically using the entropy. The last presented approach is called `saHash` which is very modular by design and is very robust against changes all over the file. The main unique feature is that the similarity measure is based on the Levensthein metric. However, it does not work for file fragments or embedded objects which is future work.

In Sec. 5.4 we presented a possibility for a faster lookup for Bloom filter based similarity digests which was one of the main drawbacks. We demonstrated a prototype which yielded promising results. However, this implementation is only able to perform a file-against-set. In a next step we want to extend this prototype to verify our theoretical concept of the whole tree.

In the last section we considered the challenge of similar file identification on network traffic using approximate matching. We demonstrated that with some minor changes approximate matching can be used on network traffic. Our tests showed that random data can be detected perfectly while real-world data has a false alert rate between 10^{-4} and 10^{-5}. The challenge of real-world data is the filtering of ‘common substrings’ which was solved using entropy and anti-randomness. However, this needs further research so that the considered packets are akin to random data. Compared to existing algorithms, our approach is very simple and straightforward: hash the file and add it to the database. In addition, we only use open source technologies.

6

Testing bytewise approximate matching

Currently it is very hard to assess and classify the different approximate matching algorithms. Most tools were briefly compared to other existing algorithms with respect to compression and runtime efficiency. However, there are additional features that are important for approximate matching. For instance, Roussev pointed out the following scenarios [81]:

1. Document similarity detection: identify related documents, e.g., different versions of a Word document.
2. Embedded object detection: identify a given object inside a container, e.g., a JPG within a Word document.
3. Fragment detection: identify an original input based on a fragment, e.g., analyzing a device on the byte level (HDD sectors).
4. Clustering files: group files that share similar content, e.g., a Word document and an e-mail.

Based on previous research, we designed and implemented the test framework `FRASH` (published in [23]) which addresses these different challenges. Currently, `FRASH` v1.0 is a collection of scripts used to analyze the *efficiency*, *sensitivity & robustness* and *precision & recall*. In the latter case, we perform tests on both synthetic (published in [24, 25]) and real world data (published in [22]).

We divided this chapter into five main parts. In Sec. 6.1 we describe the different test categories which motivate our tests. The exact realization of the tests is explained in Sec. 6.2 followed by the results in Sec. 6.3. The details about the implementation of `FRASH` 1.0 are given in Sec. 6.4. In Sec. 6.5 we summarize this chapter.

6.1. Test categories

This section gives an overview of the different test categories. First, we present the efficiency tests which are equal to an overall benchmark for approximate matching algorithms. Sec. 6.1.2 explains the sensitivity & robustness tests which determine the operating conditions. The last section describes a possibility to define the precision & recall rates.

6.1.1. Efficiency

According to the definition in Sec. 2.4 on page 13, efficiency is an essential characteristic of hash functions and approximate matching algorithms. In order to test compression and ease of computation, we designed an efficiency test which is composed of three sub-tests called generation efficiency, comparison efficiency and space efficiency (compression).

Generation efficiency. Runtime is one of the fundamental properties of most algorithms and in case of hashing often denoted by *ease of computation.* Due to large amounts of data during investigations, it is obvious that approximate matching has to be fast.

Generation efficiency measures the time which the algorithm needs to process the input. Processing in this case means that we measure the time for picking the features and forming the similarity digest. We included SHA-1 [41] as a reference point.

Comparison efficiency. The comparison efficiency completes the runtime efficiency test. First, an overall statement should be given by the theoretical complexity (O-notation). Since there are currently rarely ways to order/index similarity digests in a database, most have a quadratic lookup complexity.

Besides the theoretical O-notation, the actual time is important as the ‘similarity function’ might be based on different algorithms, e.g., Levensthein vs Hamming distance.

Thus, the absolute time of an all-against-all comparison for a given set should be given. Of course, algorithms equipped with an indexing mechanism have an advantage. Note, this time excludes the digest generation mentioned from the previous paragraph.

Space efficiency. Compression is the second essential property of hash functions. Traditional hash functions return a fixed length fingerprint, which is different to approximate matching, where we often deal with a variable length. As similarity digests are typically stored within databases a short digest is desirable.

Space efficiency measures the ratio between input and output of an algorithm and returns a percentage value. More precisely,

$$\text{space efficiency} = \frac{\text{output length}}{\text{input length}} \cdot 100 \ . \tag{6.1}$$

6.1.2. Sensitivity & robustness

The definition requires a sensitivity & robustness test (see Sec. 2.4.4 on page 16) to determine the operating conditions, the performance envelope. In contrast to precision & recall, these are no detection rates but absolute values of the robustness. Based on the four challenges from the introduction, we created the following test categories (later called modifications):

Fragment detection: An important property of approximate matching is the minimum size of a fragment that can be matched back to an original file. Thus, this test can answer the question: what is the smallest piece/fragment, for which the tool reliably correlates the fragment and the original file? Possible use cases are HDD block level or network packet analysis. For instance, analyzing a device on the sector level (e.g., 4 KiB), is it possible to find the original file – does this sector belong to this 800 MiB movie?

Single-common-block correlation: This test "simulates a situation where two files have a single common object" [81]. Considering two files f_1 and f_2 that are completely different, but share a common object O, "what is the smallest O for which the similarity tool reliably correlates the two targets?".

Alignment robustness: Some approaches are vulnerable to inserting content at the beginning of the file. Especially Bloom filter based approximate matching approaches split an input into features which are then inserted into the filters. In the case content is added at the beginning, these features shift and reduce the similarity score. Typical real life scenarios are logfiles, email traffic or office documents.

White-noise-resistance: The origin of this test goes back to the security analysis of `ssdeep` [4] where we showed that a few changes all over the input are sufficient to obtain a non-match. The intention of white-noise-resistance is to have a randomly driven test trying to produce false negatives. This allows an estimation of how many bytes need to be changed all over the input to receive a non-match. An example could be source code where a variable name is changed.

A random change is one of the typical edit operations deletion, insertion, and substitution, where each edit operation is chosen with a probability of $1/3$. Additionally, each byte in the input is equiprobable to be changed.

Knowing these test categories, the sensitivity part is guaranteed by fragment detection and single-common-block correlation. These modifications measure the amount of commonality which is detectable by the algorithm. The tool that detects smaller levels of commonality is more sensitive. On the other hand, alignment robustness and white-noise-resistance examine the robustness.

6.1.3. Precision & recall tests

Besides the performance envelope from the previous section, the precision & recall rates are important when classifying algorithms. Critical in any evaluation process is the establishment of the ground truth. Here, this means establishing whether or not two digital objects (files) are similar. Prior work, such as [81], has approached this problem from two perspectives: automated controlled tests based on synthetic data, and manual user evaluation of positive results.

The main advantage of controlled experiments is that the ground truth is constructed and therefore, precisely known. This allows randomized tests to be run completely automatically and the results to be interpreted with standard statistical measures. The obvious down side is that the majority of real data is far from random and so the applicability of the result to the general case remains uncertain. Nevertheless, running controlled tests in this manner is quite useful in characterizing baseline capabilities of algorithms.

The main advantage of user evaluation is that it provides results on real data as it would be experienced by an investigator. The downside is that the process is manual and not suitable for large-scale testing. Also, the results include a degree of subjective judgment on whether two objects are, in fact, similar. Finally, there is also the problem of how to treat objects that exhibit non-trivial commonality that is not normally observable by the user (the significance of such findings is inherently case-specific).

Based on this brief discussion, we decided on the following procedure:

Synthetic data: First is a test scenario based on controlled (pseudo-)random data. This returns exact results since we precisely know the ground truth. An obvious proceeding is to create byte sequences based on a random source (e.g., `/dev/urandom`), create mutations of them using different kinds of modification methods, run the actual comparison and summarize the results.

Real world data: Compared to synthetic data, real world data yields more realistic results and allows a better characterization of the behavior of approximate matching algorithms. To work on real world data, we have to create a ground truth and next analyze the behavior of algorithms to it. Since we have to handle huge amounts of data, it is not possible to manually create the ground truth. In other words, to set the ground truth, the similarity of objects should be defined based on a well-known metric.

Thus, we seek to bridge the gap between the two approaches by providing the means to perform *fully automated testing on real data.* In order to solve this challenge, we need a practical algorithm that can establish whether two (arbitrary) data objects are similar, or not.

6.2. Realization of tests

In the following we describe the realization of our different tests. Since the efficiency part is trivial, we skip this part and explain the sensitivity & robustness tests followed by the precision & recall tests.

6.2.1. Sensitivity & robustness

Each modification consists of different options where an option is a specific setting/test, e.g., 'fragment and 99%' or 'alignment and 4 KiB'. As one may be interested in absolute and percentage values, each modification supports two test series. However, in the default configuration, only the alignment test uses this possibility, all further tests focus on percentage values only.

Alignment robustness. This test analyzes the impact of inserting byte sequences of size X at the beginning of an input where we have the following two proceedings:

1. *Fixed blocks:* $X \in \{1, 2, 3, 4, 8, 16, 32, 64\}$ KiB. We decided on these fixed numbers as we like to analyze the impact of small changes (i.e., 1, 2 and 3) and multiples of four as they are very common in computer sciences (e.g., 4 KiB is the typical sector size of a HDD).
2. *Percentage blocks:* $X \in \{10\%, 25\%, 50\%, 75\%, 100\%, 200\%, 400\%\}$. We decided on these numbers in order to analyze the impact of large changes. Especially as logfiles may grow very rapidly.

Fragment detection. Fragment detection identifies the minimum correlation between an input and a fragment. Therefore, it sequentially cuts $X \in \{25\%, 50\%, 60\%, 70\%, 75\%, 80\%, 85\%, 90\%, 95\%, 96\%, 97\%, 98\%, 99\%\}$ of the original input and compares both inputs.

We decided to have two different cutting modes:

1. *Random cutting:* The test randomly decides at each step whether to cut at the beginning or the end of an input.
2. *End side cutting:* The test only cuts blocks at the end of an input.

The reason why we do not cut in the beginning is that this shows a similar behavior than the alignment test.

Single-common-block correlation. First, two random files f_1 and f_2 of size $S \in \{512, 2048, 8192\}$ KiB are created followed by the common block $O \in \{75\%, 50\%, 40\%, 30\%, 20\%, 10\%, 5\%, 4\%, 3\%, 2\%, 1\%\}$ of S. Next, O overwrites f_1 and f_2 at different and randomly chosen offsets – the size of f_1 and f_2 remains S all over the time. Finally, we perform a comparison of f_1 and f_2. If we obtain a match score > 0, we reduce O further and restart. Due to the fact that we choose the offset randomly, we perform multiple runs for each file size and average the values.

White-noise-resistance. First, the original file f_1 is copied to have f_2. Next, the test obfuscates f_2, i.e., X of f_2's bytes are manipulated where $X \in \{0.1\%, 0.25\%, 0.5\%, 0.75\%, 1.0\%, 1.5\%, 2.0\%, 2.5\%\}$. The percentage boundaries were determined experimentally, as no algorithm did detect two files as similar when more than 2.5% of the bytes were manipulated.

A random change is one of the typical edit operations deletion, insertion, and substitution, where each edit operation is chosen with a probability of $1/3$. Additionally, each byte in the input is equiprobable to be changed.

6.2.2. Precision & recall on synthetic data

Depending on the desired test scope, there are two main configurations: *file-count* and *runs*. The former parameter is the amount of files in the test set; the latter specifies the number of independent test runs to be executed. Each run creates its own new test set.

In terms of execution time, having a set of *file-count* files results in *file-count*^{2} comparisons. Hence, the total number of comparisons per algorithm is calculated by *file-count*$^2 \cdot runs \cdot o$ where o is the number of all options.

The mutated set is created by applying the four generic modification techniques from Sec. 6.2.1. We reduce the scope in order to decrease the overall runtime. The following settings are used:

Alignment robustness: f_2 is a copy of f_1, prefixed with a random byte string of length $X = \{25\%, 50\%, 100\%, 200\%\}$.

Fragment detection: f_2 is a copy of f_1 where $X = \{50\%, 60\%, 70\%, 80\%, 90\%, 95\%, 97\%, 99\%\}$ is cut off.

Single-common-block correlation: f_1 and f_2 have equal size and share a common byte string (block) of size $X = \{50\%, 40\%, 30\%, 20\%, 10\%, 5\%, 3\%, 1\%\}$. The position of the block is chosen randomly for each file.

White-noise resistance: f_2 is an obfuscated version of f_1, i.e., X of f_2's bytes are edited, where $X = \{0.5\%, 1.0\%, 1.5\%, 2.0\%, 2.5\%\}$ of the file size.

Recall, the term option is the combination of a modification and a specific setting/test, i.e., valid options are alignment 1% or fragment 50%. To sum it up, there are 25 different options for the pseudo random data test.

6.2.3. Precision & recall on real world data

We decided to characterize the behavior of approximate matching algorithms with regards to the longest common substring (LCS) metric. The reason for this metric is that we are most interested in shared byte sequences between objects (and actually this is what bytewise approximate matching is designed for). For instance, we like to answer the question: do these two documents contain a common picture or text-paragraph?

We are less interested of single words or translations where one needs special semantic approaches.

Recall that we are interested in byte-level similarity, which means that we do not consider any synthetic, or semantic features in our analysis. In other words, the artifacts are being compared as strings and similarity is defined as the presence of common substrings.

Considering the strings ABABBCADEF and ABAECADEBF, for example, their longest common substring is CADE. Since its length is 40% of the length of the strings, we could use that as a baseline measure of similarity. In general, it is clear that there could be additional sources of commonality, so LCS should be considered as a lower bound.

One problem with using LCS is that the algorithm has quadratic time complexity–$O(mn)$, where m and n are the string lengths in bytes. Given that files could be quite large, and the number of test cases grows quadratically as a function of the number of files in the test set, the use of an exact algorithm quickly becomes infeasible. Due to this, we created a tool which outputs a good approximation of the longest common substring and, by design, provides a lower bound on LCS (details are given in Sec. 6.2.4).

Testing methodology. First, we created a ground truth and identified the similarity of objects based on our own metric called *approximate longest common substring* (aLCS). We validate our aLCS results by comparing them against the traditional *longest common substring* (LCS). In a second step, we analyzed the false positive and false negative rates of the approximate matching algorithms with respect to the ground truth.

Our approximate ground truth is a text file of unordered pairs of files (f_1, f_2) structured as follows:

```
filename1 | size1 | filename2 | size2 | L_a | entropy | L_r
1.pdf | 98781 | 2.pdf | 185271 | 2500 | 4.66 | 0.03
3.pdf | 16661 | 4.pdf | 18530 | 2077 | 1.75 | 0.12
```

where L_a is the absolute result (a lower bound on the length of the longest common substring), entropy is the information content of the substring and L_r is the relative result. More precisely,

$$\begin{aligned} L_a &= alcs_a(f_1, f_2), \text{ where } 0 \leq L_a \leq min(|f_1|, |f_2|). \\ L_r &= \lceil 100 \times {}^{L_a}/_{min(|f_1|,|f_2|)} \rceil \text{ , where } 0 \leq L_r \leq 100. \end{aligned}$$

where $|f|$ denotes the file size in bytes.

For instance, the first line states that 1.pdf and 2.pdf have an absolute length of $L_a = 2500$ bytes which corresponds a relative length $L_r = 0.03 = 3\%$ with an entropy of 4.66. The second line shows a special case with a very low entropy which could be an indicator that both files share mostly zeros.

Although, in theory, any two strings sharing a substring are related, we place a more practical lower bound on the minimum amount of commonality to declare two files related. For our testing, we decided to study the following two perspectives:

Relative: We require that the absolute size L_a is at least 100 *and* that the relative result L_r exceeds 0.5% of the size of the smaller file. According to this, the true positive function $TP_{alcs}(f_1, f_2)$ is defined as

$$TP_{alcs}(f_1, f_2) \equiv L_a \geq 100 \quad \wedge \quad L_r \geq 1 \;.^{1}$$

Clearly, the *true negative* function $TN_{alcs}(f_1, f_2) = \neg TP_{alcs}(f_1, f_2)$.

Absolute: We require that the absolute size L_a is at least 2048 bytes (2 KiB) regardless the file size.

$$TP_{alcs}(f_1, f_2) \equiv L_a \geq 2048 \;.$$

6.2.4. Approximate longest common substring

The basic idea of the approximate longest common substring metric (aLCS) is not to compare files byte by byte but rather in variable sized chunks. To pick the chunks, we use a derivative of the standard approach to data fingerprinting by random polynomials pioneered by Rabin. Specifically, we borrow the rolling hash from `ssdeep` and adjust the parameters such it produces chunks of 40 bytes, on average. Each chunk is hashed with the FNV-1a hash [69] and the sequence of all hash values forms the basis for the *alcs-digest*. Besides the hash values, we also store the entropy and length for each chunk.

Given two digests, it is straight forward to construct an estimation of the LCS; a reference implementation is publicly available on our website[2].

Implementation details. The tool is implemented in *C* and separated into three steps: reading, hashing and comparing, which are declared in the main class. It has a command line interface and is run against all files in a target directory: `./aLCS <dir>` .

First, all files in `dir` are read. Out of the file names, we create 'hash-tasks' which are added to a thread pool. A hash-task contains the path to a file and denotes 'hash file x'. Depending on the amount of threads, these tasks are processed. Once all alcs-digests are created, we perform an all-against-all comparison. Therefore, we create compare-tasks (compare $file_1$ against $file_2$) which are again added to the thread pool. The output is printed to the standard output.

The reference implementation has three main settings configurable in `header/config.h`. `MIN_LCS` is the minimum L_a length which is printed to stdout and is by default 0 (all comparison are printed). The `THREAD_POOL_QUEUE_SIZE` is the length of the queue and should be ${}^{fileamount \cdot (fileamount - 1)}/_{2}$. `NUMTHREADS` is the amount of threads which should be equal to the amount of cores.

Verification of ground truth. To verify the correctness of our approximate longest common substring, we compared the results against LCS for a subset of t5. In order to solve this challenge, we implemented a parallelized LCS tool written in *C*. The output

---

[1] Note: result of L_r is rounded and thus 0.5 is equal to 1.

[2] `http://www.dasec.h-da.de/staff/breitinger-frank/#downloads` (last accessed 2014-03-13).

is a summary file similar structured then our aLCS output: `file1 | file2 | LCS`. A small ruby script is used to compare LCS-summary and aLCS-summary.

Our subset consists of 201 randomly selected files. We compared these files using aLCS as well as LCS and finally compared both summaries. All 20100 comparisons yield a true positive, i.e., $0 \leq alcs \leq lcs$. We also consider the distribution of the differences between the LCS and aLCS scores. Specifically, we define d_r for files f_1 and f_2 as follows:

$$d_r = \left\lceil 100 \times \frac{lcs(f_1, f_2) - alcs(f_1, f_2)}{min(|f_1|, |f_2|)} \right\rceil, d_r \in 0, 1, \ldots, 100.$$

In other words, we consider the score difference relative to the size of the smaller of the two files, and build the probability (pdf) and cumulative distribution function (cdf) in Table 6.1. As we can see, upwards of 95% of the observed differences do not exceed 3% of the size of the smaller files – we consider this a reasonable starting point for our purposes (further research may refine this). If anything, this should give tools a slight boost as the available commonality would be underestimated.

Table 6.1.: Empirical *pdf* & *cdf* for d_r.

X	0	1	2	3	4	5	10	15	20
$Pr\{d_r = X\}$	0.8869	0.0449	0.0155	0.0040	0.0047	0.0116	0.0062	0.0001	0.0000
$Pr\{d_r \leq X\}$	0.8869	0.9318	0.9473	0.9513	0.9561	0.9677	0.9834	0.9992	0.9999

6.3. Testing results

The past sections explained the different test categories including their realization. Based on these decisions, this section presents all test results. Sec. 6.3.1 shows the efficiency results. In Sec. 6.3.2 we discuss the results for the sensitivity & robustness test. The last section summarizes our findings for the precision & recall. All test are based on the t5-corpus except the precision & recall on synthetic data.

The results are based on `ssdeep` 2.9 and `sdhash` 3.4 with their default settings. `mrsh-v2` uses the default settings with a blocksize of 160. For `mvhash` we decided to change the default settings to `-t 7 -w 20 -e 1024`. The reason is that we have to deal with a lot of text based files and therefore we set `-t 7`. Files are rather small which means we can reduce the neighborhood (`-w 20`) and the amount of groups per filter (`-e 1024`).

Definitions and terminology. We follow a fairly standard information retrieval framework for evaluating the quality of the results produced by approximate matching tools.

Genuines: Two, or more files, defined to be similar.

synthetic data: In this case we know by default if the files are similar or not.

real data: In this case we use our ground truth as discussed in Sec. 6.2.3.

Impostors: Two, or more files, defined to be non-similar.

Approximate matching score: $S(f_1, f_2)$ is the result of comparing two files using an approximate matching function.

Threshold (t) of significance: A score parameter, used in approximate matching to separate matches from non-matches.

Match: Two files, f_1 and f_2, are matched using approximate matching algorithm $h \iff S(f_1, f_2) > t$.

True positive TP:

$$TrP(f_1, f_2, t) \equiv TrP_{alcs}(f_1, f_2) = true \ \wedge \ S(f_1, f_2) > t$$

True negative TN:

$$TrP(f_1, f_2, t) \equiv TrN_{alcs}(f_1, f_2) = true \ \wedge \ S(f_1, f_2) \leq t$$

False positive FP:

$$TrP(f_1, f_2, t) \equiv TrN_{alcs}(f_1, f_2) = true \ \wedge \ S(f_1, f_2) > t$$

False negative FN:

$$TrP(f_1, f_2, t) \equiv TrP_{alcs}(f_1, f_2) = true \ \wedge \ S(f_1, f_2) \leq t$$

Precision P, true positive rate TPR:

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

Recall R:

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

True negative rate TNR:

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

False positive rate FPR:

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

Negative predictive value NPV:

$$NPV = \frac{TN}{TN + FN}$$

Accuracy A:

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

F-score F_β:

$$F_\beta = (1 + \beta^2) \times \frac{precision \times recall}{\beta^2 \times precision + recall}$$

The F-score is a generic measure combining precision and recall into a single number. We use three different versions based on the β parameter: $F_1, F_2, F_{0.5}$. The first one weights precision and recall equally, the second favors precision over recall, and the last one favors recall over precision.

Matthews correlation coefficient MCC:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}, \; MCC \in [-1, 1]$$

The MCC is a correlation coefficient between observed and predicted binary classifications; it is included here as it is considered a balanced measure even for classes of substantially different sizes (as is our case). A result of +1 represents perfect prediction, whereas −1 indicates perfect disagreement; 0 indicates that the classifier offers no advantage over a random guess.

6.3.1. Efficiency

The test environment for the efficiency was a private laptop having the following benchmark data:

```
CPU : 4x Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz (2 Cores, 4 Threads)
HDD : Mushkin Chronos SSD 120GB (SATA3)
RAM : 2x4GB DDR3 SODIMM 1333 MHz
```

Generation and comparison efficiency. The results for both tests are given in Table 6.2. *Total* and *fingerprint comparison* measure the time for hashing all files and run an all-against-all comparison, respectively. The last column shows the relationship of all algorithms compared to SHA-1.

`mvhash` is the fastest algorithms followed by `mrsh-v2` which results from the straightforward proceeding. The slowest algorithm is `sdhash` (`-p 1` indicates a single thread) but outperforms at least `ssdeep` when it is parallelized. Obviously all algorithms are worse than SHA-1.

The fingerprint comparison shows that `ssdeep` and `mvhash` overcome both other algorithms. This results from the pre-condition: both algorithms only compare similarity digests in the same range. For instance, if the block sizes between two `ssdeep` similarity digests differ too much, they are not compared.

Table 6.2.: Runtime efficiency and fingerprint comparison.

	Total	Fingerprint comparison	$\frac{algorithm}{SHA-1}$
sha1sum	5.128 s	-	1.00
`ssdeep` -s	39.925 s	18.943 s	7.79
`sdhash` -p 1	67.733 s	289.077 s	13.21
`sdhash`	28.880 s	187.238 s	5.63
`mrsh-v2`	13.300 s	244.430 s	2.59
`mvhash`	9.219 s	31.502 s	1.80

Space efficiency. Table 6.3 shows the results for the compression test. The framework outputs the average similarity digest length, average compression ratio in percent, the maximum similarity digest including the corresponding file (not included in the table) and the size of all digests.

For the 'digest file size' test we run the algorithm with the t5-corpus and store the resulting similarity digests (including all further information) in a text file. The 'avg. digest length' test also considers the digest representation and ignores additional information like file name. For instance, `ssdeep` uses a Base64 encoding and always outputs the file name.

To conclude, with respect to compression `ssdeep` outperforms all other algorithms while `sdhash` produces the longest similarity digests.

Table 6.3.: Compression test overview.

	Avg. digest length	Digest file size	compression ratio
`ssdeep` -s	63.4 B	0.52 MiB	0.015%
`sdhash`	10812.2 B	61.28 MiB	2.520%
`mrsh-v2`	3215.5 B	27.45 MiB	0.750%
`mvhash`	775.9 B	4.55 MiB	0.181%

6.3.2. Sensitivity & robustness

This section is divided into four paragraphs, based on the different tests: alignment robustness, fragment detection, single-common-block correlation and white-noise-resistance. All tests within this category have a very comprehensive output as the tables consist of 10 or more columns. Therefore, we decided to pick out some selected columns which are presented in the following.

'Matches' is the amount of valid scores, i.e., how many percent of all files were matched. 'Avg. score' is the averaged match score output by the algorithm. The last row is the standard deviation of the match score.

Alignment robustness. The results are Table 6.4 and Table 6.5 showing the impact of *percentage blocks* and *fixed blocks*, respectively.

All algorithms perform well for small changes where small is either 4 KiB or 50%. With respect to larger alignments especially `sdhash` and `mrsh-v2` perform almost perfect having matches between 100.0% and 99.9%. Having a closer look at these two approaches, it is eye-catching that the average match score is very similar for ‘percentage changes’ but goes up and down for ‘fixed blocks’. This behavior comes from the usage of Bloom filters. If `sdhash`, for example, uses 16 KiB to fill the Bloom filter, than an alignment of 8 KiB reduces the match score to a minimum. Although `mvhash` also uses Bloom filter, it behaves differently which comes from the run-length coding.

As mentioned in Sec. 3.1, `ssdeep` can by design only detect similarities between two inputs if their size differs by less than $^{1}/_{3}$ (with some exceptions). This is also the conclusion from Table 6.4, `ssdeep` can hardly find similarities if the modification is too large. For instance, the amount of matches reduces down to 22.7% if we add 200%. Similar to `ssdeep`, `mvhash` has problems matching files that differ to much in size. Therefore, the matches reduce for large changes like 200% or 64 KiB.

Table 6.4.: Results of the alignment robustness test with percentage block sizes.

		added block size (%)						
		10	25	50	75	100	200	400
`ssdeep`	Matches (%)	99.6	98.2	92.1	82.2	70.6	22.7	0.0
	Avg. score	90.80	81.44	70.89	63.98	59.14	44.28	37.5
	Std. deviation	6.48	9.75	11.09	11.03	10.94	10.64	5.5
`sdhash`	Matches (%)	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	Avg. score	69.81	71.44	72.29	71.68	70.91	71.49	70.93
	Std. deviation	20.53	20.57	21.00	21.26	20.81	21.05	21.39
`mrsh-v2`	Matches (%)	99.9	99.9	99.9	99.9	99.9	99.9	99.9
	Avg. score	77.40	77.77	78.71	79.04	79.45	79.78	79.81
	Std. deviation	20.88	20.44	20.65	20.62	20.27	20.26	20.25
`mvhash`	Matches (%)	96.5	91.0	83.2	77.5	73.0	59.1	43.7
	Avg. score	75.70	67.14	59.98	56.77	53.92	48.13	46.82
	Std. deviation	22.33	23.47	23.45	22.91	2.54	24.22	28.77

Fragment detection. Table 6.6 and Table 6.7 show the results for *random cutting* and *end side cutting*, respectively. Actually each table comprises 23 columns containing the results for all cuts $5\%, 10\%, 15\%, \ldots, 95\%, 96\%, 97\%, 98\%, 99\%$. We reduced this down to 9 in order to provide a suitable overview. Note, there is a difference between the cut size and fragment. If we cut 10%, then the fragment size is 90%.

As already mentioned in the alignment paragraph, `ssdeep` can by design only detect file fragments between 50% and 25% (i.e., cut sizes 50% to 75%) which is also the conclusion considering both tables. The algorithm works with a high precision until cuts of 60% then the ‘matches’ reduce rapidly. Table 6.7 shows that `ssdeep` also uncovers 0.5% of 5%-fragments which is one of the mentioned special cases.

Table 6.5.: Results of the alignment robustness test with fixed block sizes.

		added block size (KiB)							
		1	2	3	4	8	16	32	64
`ssdeep`	Matches (%)	100.0	99.9	98.9	93.8	84.4	71.8	57.3	44.1
	Avg. score	96.46	92.66	88.35	85.52	82.29	79.19	76.27	72.83
	Std. deviation	3.87	8.04	12.20	14.07	16.28	17.71	18.61	19.28
`sdhash`	Matches (%)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	Avg. score	86.82	63.15	59.97	91.67	89.98	56.58	89.55	84.91
	Std. deviation	6.31	15.64	18.16	6.08	8.29	19.48	9.74	13.54
`mrsh-v2`	Matches (%)	99.9	99.9	99.9	99.9	99.9	99.9	99.9	99.9
	Avg. score	95.16	90.04	82.47	72.65	65.39	88.93	85.30	78.70
	Std. deviation	3.56	6.34	12.24	20.21	25.88	9.79	13.66	19.08
`mvhash`	Matches (%)	99.3	98.8	98.5	98.2	96.9	93.5	85.8	71.9
	Avg. score	96.44	93.4	90.38	87.6	78.4	64.8	44.8	33.1
	Std. deviation	8.45	10.08	11.99	13.48	16.76	17.82	14.38	14.33

Table 6.6.: An extract of the fragmentation detection test using random cutting.

		cut size (random)								
		10%	25%	50%	60%	70%	80%	90%	95%	99%
`ssdeep`	Matches (%)	99.9	99.1	91.2	70.2	37.7	8.6	0.3	0.0 (2)	0
	Avg. score	83.25	80.19	65.82	58.59	51.03	45.95	43.94	48.5	0
	Std. deviation	19.43	8.27	9.81	10.26	10.86	12.12	14.42	16.50	0
`sdhash`	Matches (%)	100.0	100.0	100.0	99.9	99.8	99.4	99.6	78.6	42.9
	Avg. score	78.36	70.27	68.89	70.14	71.72	72.84	75.99	77.44	79.77
	Std. deviation	22.5	22.28	22.55	22.86	23.07	22.92	22.44	22.21	22.29
`mrsh-v2`	Matches (%)	99.9	99.9	99.4	98.7	97.3	92.5	82.0	68.3	37.5
	Avg. score	93.38	75.99	74.11	73.47	74.13	73.98	74.25	75.29	76.27
	Std. deviation	5.97	20.65	20.28	20.39	19.5	19.21	18.62	18.09	17.14
`mvhash`	Matches (%)	96.3	86.7	71.8	53.1	29.7	15.4	3.2	0.6	0.0 (3)
	Avg. score	77.46	60.27	37.21	31.63	35.21	34.85	24.26	21.12	52.0
	Std. deviation	20.58	20.1	19.91	21.68	22.83	21.32	20.80	17.07	34.91

In contrast, `sdhash` outputs a very high rate also for 95%-cuts. In case of 99%-cut `sdhash` has 42.9% matches which is still remarkable. `mrsh-v2` is similar to `sdhash` but receives consistently a few less matches. `mvhash` performs similar than `ssdeep` but outperforms it when dealing with cuts of 80% or more.

Comparing both cutting modes, the results for `ssdeep` and `mvhash` are almost the same. However, `sdhash` and `mrsh-v2` show a different behavior which is again due to the similarity digest representation using Bloom filters. End side cutting only influences the last Bloom filters and thus, the beginning is equal which results in high scores with only a little deviation. On the other hand random cutting also cuts the beginning which shifts features to different Bloom filters and reduces the match score.

Table 6.7.: An extract of the fragmentation detection test using end side cutting.

		cut size (end)								
		10%	25%	50%	60%	70%	80%	90%	95%	99%
`ssdeep`	Matches (%)	99.9	99.8	93.1	74.6	44.1	14.2	1.7	0.5	0
	Avg. score	95.90	88.87	71.70	64.38	58.03	55.32	56.55	55.95	0
	Std. deviation	4.23	8.14	11.85	12.87	14.48	15.42	21.28	20.11	0
`sdhash`	Matches (%)	100.0	100.0	100.0	100.0	100.0	100.0	97.9	80.5	41.2
	Avg. score	99.68	99.63	99.42	99.37	99.18	98.81	97.97	97.83	97.63
	Std. deviation	1.09	1.22	1.38	1.55	1.87	2.60	4.184	4.275	3.54
`mrsh-v2`	Matches (%)	99.9	99.9	99.6	99.1	97.7	93.9	83.8	69.7	35.2
	Avg. score	97.82	97.60	97.01	96.62	96.11	95.57	93.61	93.61	92.20
	Std. deviation	1.93	2.33	2.98	3.49	4.04	4.46	5.16	5.98	6.06
`mvhash`	Matches (%)	97.9	93.22	81.3	65.0	45.2	29.9	9.8	3.2	0.5
	Avg. score	90.17	78.23	57.11	54.78	57.39	51.89	44.48	45.46	63.32
	Std. deviation	10.19	16.18	29.23	31.51	28.41	26.73	29.94	33.05	34.28

Single-common-block correlation. Although this test outputs three tables showing the results for files of 512, 2048 and 8196 KiB, we only included a summary for 2048 KiB as the results are very similar. We decided for a test scope of 100 files. The main conclusion of Table 6.8 is that `ssdeep` outputs higher scores while the other approaches detect smaller common blocks. Especially `sdhash` performs well as it obtains match rates of 100% for all blocks $\geq 3\%$.

Table 6.8.: An extract of the single-common-block correlation with a file size of 2048 KiB.

		single common block size								
		1%	2%	3%	4%	10%	20%	30%	40%	50%
`ssdeep`	Matches (%)	0	0	0	0	8	66	90	100	100
	Avg. score	0	0	0	0	28.00	31.95	36.04	47.09	54.12
	Std. deviation	0	0	0	0	2.55	5.56	6.35	8.00	6.74
`sdhash`	Matches (%)	44	97	100	100	100	100	100	100	100
	Avg. score	1	1.26	1.76	2.24	6.16	12.39	18.08	23.66	29.55
	Std. deviation	0	0.44	0.66	0.91	2.05	4.27	6.39	8.70	11.14
`mrsh-v2`	Matches (%)	0	35	73	93	100	100	100	100	100
	Avg. score	0	1.00	1.32	1.81	5.06	11.11	17.24	23.94	28.11
	Std. deviation	0	0	0.46	0.72	2.14	4.64	6.73	9.21	10.51
`mvhash`	Matches (%)	0	67	87	95	100	100	100	100	100
	Avg. score	0	1.06	1.61	2.13	5.27	12.10	17.51	23.55	29.50
	Std. deviation	0	0.21	0.61	0.91	1.76	3.97	5.63	7.60	10.06

White-noise-resistance. The result for the white-noise-resistance is presented by Table 6.9. Recall, this test randomly changes bytes within an input.

The security analysis [4] showed that it is theoretically possible to produce a non-match by changing 7 bytes in an input. In Sec. 5.3 we showed that about 70 random changes are sufficient to produce non-matches. Thus, it is not surprising that `ssdeep` performs worst. Again, `sdhash` and `mrsh-v2` show a very similar behavior with slight advantages for `sdhash`. `mvhash` shows a better behavior than `ssdeep` for small modifications ($< 1.00\%$) but gets a bit worse for larger ones.

Table 6.9.: Results for the white-noise-resistance test.

		modifications							
		0.10%	0.25%	0.50%	0.75%	1.00%	1.50%	2.00%	2.50%
`ssdeep`	Matches (%)	40.7	21.6	10.4	5.4	2.9	1.3	0.7	0.4
	Avg. score	74.72	66.96	60.03	54.41	49.60	43.32	40.07	37.5
	Std. deviation	12.42	12.06	11.01	10.46	9.81	9.68	9.07	9.19
`sdhash`	Matches (%)	100.0	100.0	100.0	99.9	99.7	74.6	6.7	1.5
	Avg. score	82.70	65.69	44.57	28.50	16.2	3.20	3.14	2.63
	Std. deviation	9.29	10.42	9.81	8.59	7.02	3.26	2.83	2.34
`mrsh-v2`	Matches (%)	99.9	99.6	94.1	37.1	7.7	0.7	0.2	0.0
	Avg. score	65.61	37.26	12.23	5.03	4.648	4.13	3.56	2.75
	Std. deviation	8.553	9.59	7.51	5.75	5.16	4.31	2.17	1.78
`mvhash`	Matches (%)	95.17	87.9	63.9	38.5	17.7	1.1	0.1	0.0 (2)
	Avg. score	61.91	34.79	14.62	6.44	3.3	1.7	2.5	2
	Std. deviation	15.88	14.74	9.98	5.54	2.88	1.25	2.06	0

6.3.3. Precision & recall on synthetic data

Our test examines the performance of the algorithms as a function of file size. For that purpose we consider the behavior at six fixed file sizes – 1, 4, 16, 64, 256 and 1024 KiB. We decided for these size boundaries after analyzing the sizes of almost 1,000,000 files in the `govdoc`-corpus[3]. As shown in Table 6.10, nearly 91% of all files are smaller than 1 MiB.

Table 6.10.: File sizes distribution in the `govdoc`-corpus (min size is 1 KB).

File size range (KiB)	≤ 4	≤ 16	≤ 64	≤ 256	≤ 1024
Amount (%)	5.40	20.71	52.54	75.82	90.60

To avoid library-level integration with the tools (which would facilitate efficiency but introduce tight code dependencies) we found it necessary to use the command line interface provided by the tool. Thus, even though each tool invocation completes in a fraction of a second, there is considerable overhead that adds up on a large scale.

---

[3] "These documents were obtained by performing searches for words randomly chosen from the Unix dictionary, numbers randomly chosen between 1 and 1 million, and randomized combinations of the two, for documents of specified file types that resided on web servers in the .gov domain using the Yahoo an Google search engines" (`http://digitalcorpora.org/corpora/files`).

One challenge that may not be immediately obvious is the amount of computation needed to complete the tests. For this test we did 10 different runs on 100 files (vs 100 modified versions) which results in 100,000 comparisons per option which ends up in 2,500,000 (25 options) comparisons per file size and algorithm. Due to 6 different file sizes, we had 15 million comparisons per algorithm, which is in total 60 million comparisons.

The rest of this section is divided into three parts. First, we explain the representation of our results followed by the average results over all file sizes and tests in the next paragraph. As this is very general, we decided to present some more details about specific options in the last paragraph.

Result presentation. A common way to visualize the precision & recall rates are score histograms, error probability distributions, and detection error trade-off curves (DET curves).

Score histograms show the frequency for each score value, i.e., how often a score occurs, for both true and false positives. Ideally, scores for known impostors would be all zero, while genuine scores would be positive. More precisely, we need impostor scores to be generally lower than genuine scores and we could use the threshold parameter as means to clearly separate them. The score histogram is a convenient means to identify suitable threshold values.

Error probability distributions show the *false positive rate (FPR)* and *false negative rate (FNR)* as a function of the chosen threshold value. More formally, let t be the threshold where $0 \leq t \leq 100$ and s denote the comparison score. Then:

- $FPR(t)$ is the number of impostor comparisons with $s \geq t$ divided by the total number of impostor comparisons.
- $FNR(t)$ is the number of genuine comparisons with $s < t$ divided by the total number of genuine comparisons.

Detection error trade-off curve correlates the FPR (x-axis) and the FNR (y-axis). Thus, we can answer the question *if at most x% false matches shall be tolerated, how many false non-matches must be expected?*. Obviously, the more false matches are tolerated, the less false non-matches can be expected and vice versa.

Averaged results over all file sizes. Recall, this test considers the behavior for the file sizes 1, 4, 16, 64, 256 and 1024 KiB. In a first step we decided to provide the average results per algorithms for all sizes, options and tests. Thus, we have a single histogram, error probability distribution and DET-curve for each algorithm.

The Score histograms for the algorithms are given in Fig. 6.1. In case of `ssdeep` we do not have any impostors with a score above 0 (which is perfect). `sdhash` has 15 impostors with a score up to 4 which is almost perfect. `mrsh-v2` and `mvhash` have 8160 and 5,317,189, respectively, with scores up to 30 and thus perform worse.

With regards to the false negatives, `sdhash` (70,280) performs best followed followed by `mrsh-v2` (91,603). `mvhash` (121,060) and `ssdeep` (127,730) have a significantly higher amount of false negatives.

Table 6.11 and 6.12 show the false positives and false negative, respectively, dependent on the file size. Almost all algorithms do not produce false positives for files up to 16 KiB. For larger files, especially `mvhash` outputs huge amounts of false positives while `mrsh-v2` also has a few thousands.

Table 6.11.: False positives with respect to the file size.

KiB	1	4	16	64	256	1024
`ssdeep`	0	0	0	0	0	0
`sdhash`	0	0	0	0	1	14
`mrsh-v2`	0	0	28	3,190	2,517	2,425
`mvhash`	0	0	0	1,673,762	2,272,710	1,370,717

Studying the false negatives in Table 6.12 shows that `sdhash` and `mrsh-v2` have problems, especially to handle small files ($\leq$ 16 KiB) while they improve for larger files (e.g., $\geq$ 64 KiB). The amount of false negatives of `ssdeep` is rather constant and around 22,000 independent of the size. `mvhash` has problems to handle small files and large files but show that it works better than `ssdeep` for 64 and 256 KiB files.

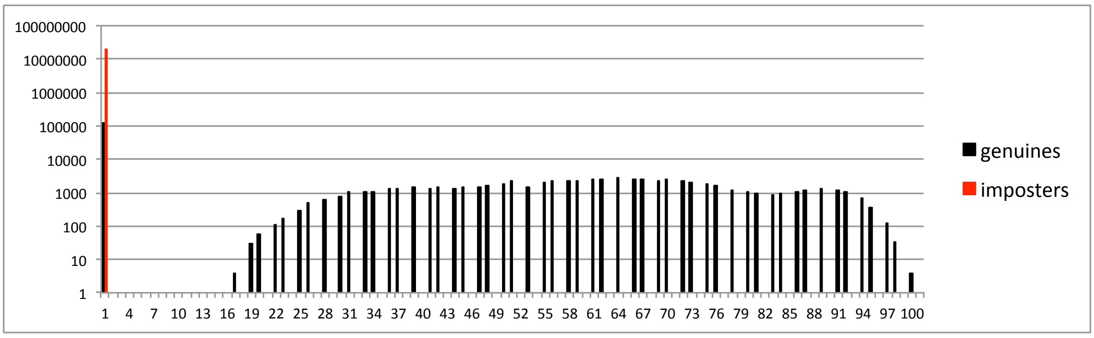
Table 6.12.: False negatives with respect to the file size.

KiB	1	4	16	64	256	1024
`ssdeep`	26,531	21,221	22,020	22,050	21,956	22,023
`sdhash`	18,460	18,007	12,657	7,075	3,216	2,794
`mrsh-v2`	31,517	21,384	16,062	10,412	6,807	5,421
`mvhash`	29,436	27,943	25,273	10,736	8,627	19,045

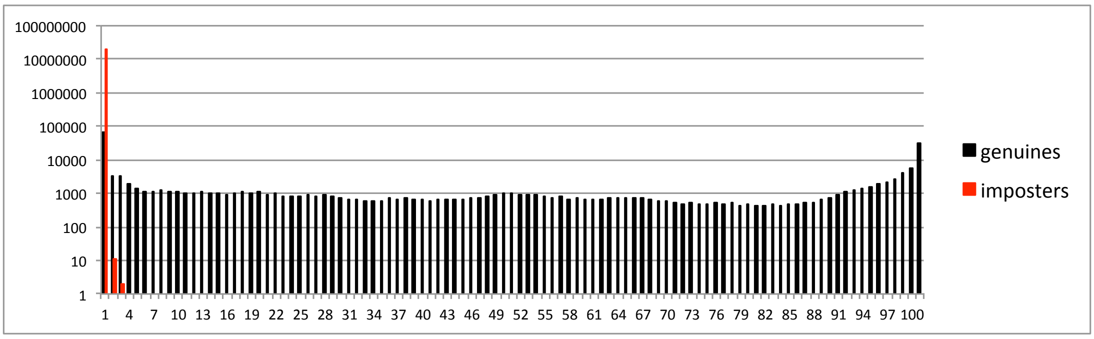
The error probability distributions for the algorithms are given in Fig. 6.2. The false positive rates (black lines) for $t = 1$ of `ssdeep`, `sdhash` and `mrsh-v2` are 0, 10^{-7} and 10^{-4}, respectively. Hence, these three algorithms perform pretty well. `mvhash` has a false positive rate of 27% which gets zero for $t \geq 26$.

With respect to false negatives (red lines), `ssdeep` and `mvhash` have an initial rate of approximately 62% for ($t = 1$) while `sdhash` and `mrsh-v2` perform better with a rate of approximately 40%. As already discussed in the previous paragraph, these high false negative rates mostly result from small files, e.g., `sdhash` drops the comparison if an input has less than 512 KiB.

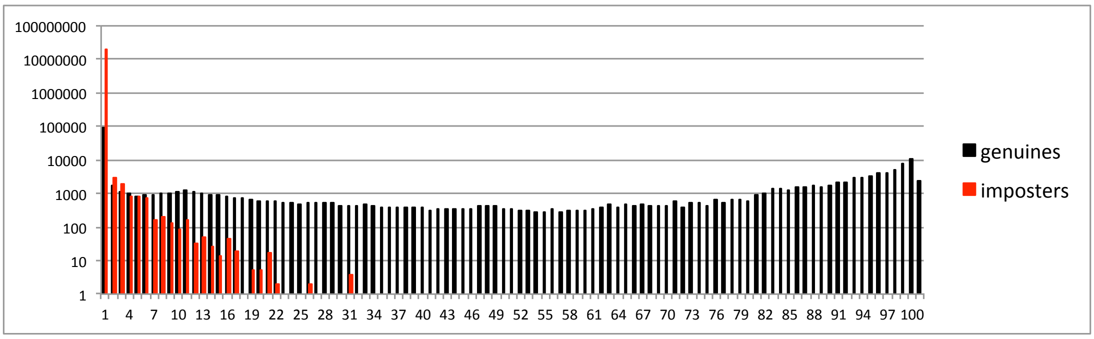
The detection error trade-off curves are given in Fig. 6.3. Note, `ssdeep` and `mvhash` use the regular scale while both others use a logarithmic scale for a better view. In case



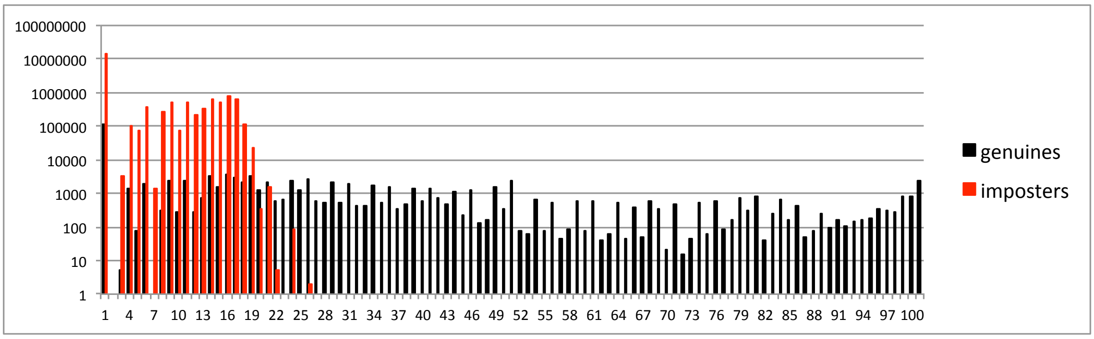
(a) Histogram for `ssdeep`.



(b) Histogram for `sdhash`.

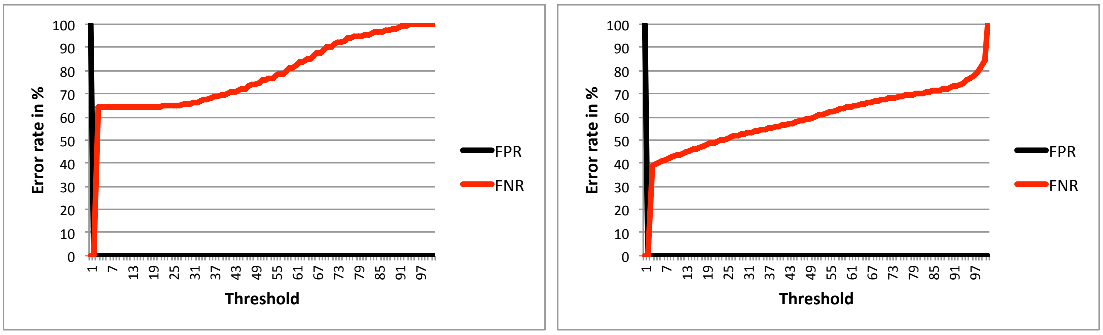


(c) Histogram for `mrsh-v2`.



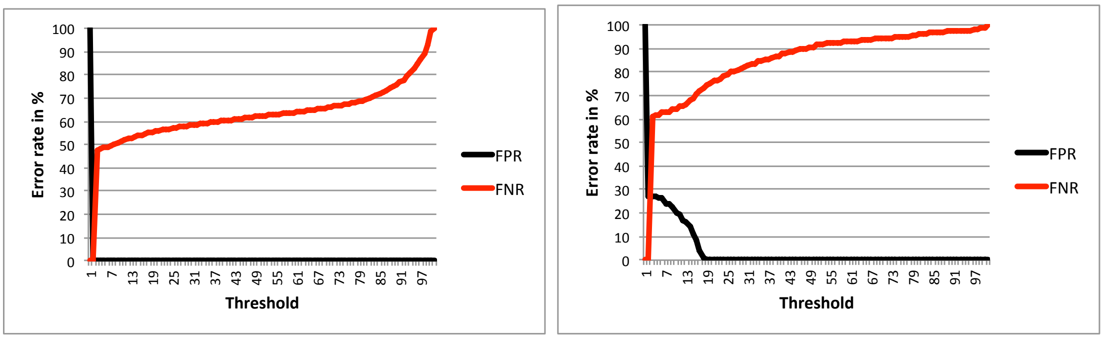
(d) Histogram for `mvhash`.

Figure 6.1.: Histograms for different algorithms.



(a) `ssdeep`.

(b) `sdhash`.



(c) `mrsh-v2`.

(d) `mvhash`.

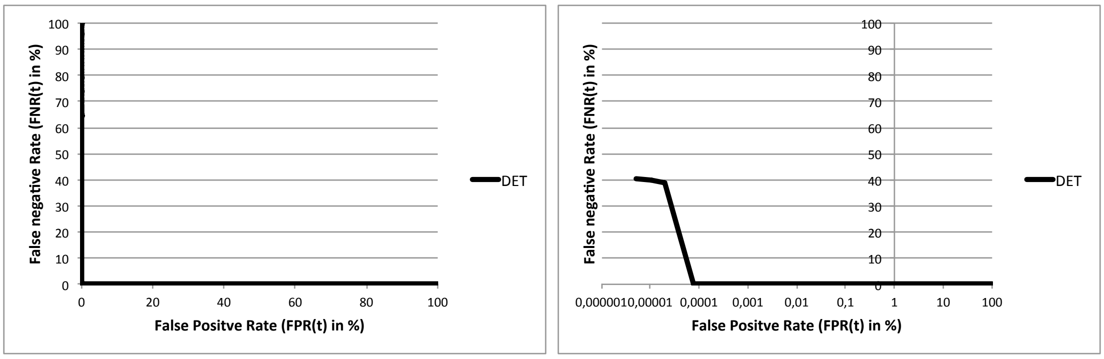
Figure 6.2.: Error probability distribution for different algorithms.

of `ssdeep` there are plenty false negatives but no false positives. Thus, the graph drops straight to the x-axis. `mvhash` has a lot of false positives with a score ≤ 25 and then goes down to zero. `sdhash` and `mrsh-v2` have less false negatives then the aforementioned algorithms. However, they have some false positives.

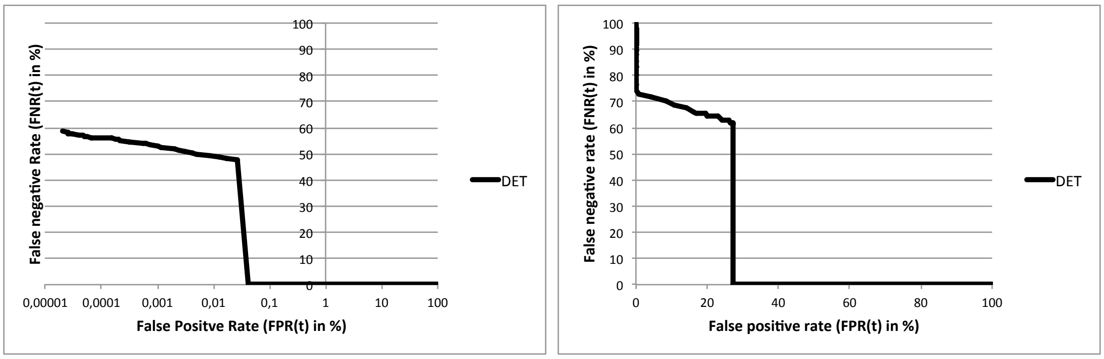
Detailed test results. In this section, we summarize the results for each of the four types of data manipulation and for all file sizes (i.e., 1, 4, 16, 64, 256, and 1024 KiB). All details are given in Appendix B. In addition to the numerical scores, we discuss the relationship between the observed behavior and the design of the algorithms.

Fragment detection. In this test, we evaluate the ability of the algorithms to find a piece of target. Specifically, we compare a random test file f_1 with a randomly sampled cut which results in f_2. The cut size is X of the original, where $X = \{50\%, 60\%, 70\%, 80\%, 90\%, 95\%, 97\%, 99\%\}$. Table 6.13 shows the results and lead to the following observations.

The behavior of `ssdeep` is consistent across all file sizes: above 0.97 for the 50% case, $0.68 - 0.71$ at 60%, and (near) zero in all other cases. Considering the algorithm, these observations make sense. The algorithm produces a fixed size similarity digest, which means that, in *relative* terms, it maintains the same resolution, so the minimum



(a) `ssdeep`. (b) `sdhash`.



(c) `mrsh-v2`. (d) `mvhash`.

Figure 6.3.: Detection error trade-off for different algorithms.

Table 6.13.: True positive rates as a function of file and cut size (random cutting).

		50%	60%	70%	80%	90%	95%	97%	99%
`ssdeep`	1	0.98	0.68	0.04	0.02	0.00	0.00	0.00	0.00
	4	0.98	0.69	0.04	0.02	0.00	0.00	0.00	0.00
	16	0.98	0.71	0.03	0.01	0.00	0.00	0.00	0.00
	64	0.98	0.69	0.03	0.02	0.00	0.00	0.00	0.00
	256	0.97	0.68	0.04	0.02	0.00	0.00	0.00	0.00
	1024	0.97	0.69	0.03	0.02	0.00	0.00	0.00	0.00
`sdhash`	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	4	1.00	1.00	0.99	0.04	0.00	0.00	0.00	0.00
	16	1.00	1.00	1.00	1.00	1.00	0.05	0.00	0.00
	64	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
	256	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	1024	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
`mrsh-v2`	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	4	0.94	0.76	0.44	0.09	0.00	0.00	0.00	0.00
	16	1.00	1.00	1.00	1.00	0.77	0.08	0.00	0.00
	64	1.00	1.00	1.00	1.00	1.00	1.00	0.91	0.02
	256	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	1024	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
`mvhash`	1	0.34	0.08	0.01	0.00	0.00	0.00	0.00	0.00
	4	0.79	0.17	0.00	0.00	0.00	0.00	0.00	0.00
	16	1.00	0.56	0.00	0.00	0.00	0.00	0.00	0.00
	64	1.00	1.00	0.62	0.00	0.00	0.00	0.00	0.00
	256	1.00	1.00	1.00	0.65	0.48	0.00	0.00	0.00
	1024	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

detectable fragment should be defined in relative terms; the tests clearly capture this feature.

The behavior of `sdhash` and `mrsh-v2` is more dynamic; they start with complete failure at 1 KiB but quickly improves as the file size grows: at 4 KiB, `sdhash` needs a 30% sample (a cut of 70%) for near-perfect detection, whereas at 256 KiB and up, even a 1% sample is detected perfectly. Overall, `mrsh-v2` is bit less precise.

This behavior should be expected. The tools use a variable-sized digest so a fragment of any size above the design minimum should be detectable, regardless of the size of the source file. Considering (in *absolute* terms) the size of the fragment at which it becomes perfectly detectable, we can see that it is approximately the same in all cases: $30\% \times 4\,\text{KiB} \approx 10\% \times 16\,\text{KiB} \approx 2\% \times 64\,\text{KiB}$.

The behavior of `mvhash` looks random. It is not consistent across file sizes and only improves until the 256 KiB; then it fails completely. Considering the implementation, this makes sense. `mvhash` can only compare similarity digests if the difference between two similarity digests (i.e., Bloom filter sequences) is less than four which is the reason for the failure at 1024 KiB. On the other hand it works more precise then `ssdeep` for files between 64-256 KiB and almost equal for 16 KiB.

Single-common-block correlation. An extension of the fragment test, the single-common-block test evaluates the ability of an approximate matching algorithm to correlate two files, f_1 and f_2, that are known to have fragment in common. To simplify the analysis and presentation, we choose the files of equal size and vary the amount of commonality as a fraction X of the file size; $X = \{50\%, 40\%, 30\%, 20\%, 10\%, 5\%, 3\%, 1\%\}$. Table 6.14 shows the results and lead to the following observations.

Table 6.14.: True positive rates for as a function of file and common block size.

		50%	40%	30%	20%	10%	5%	3%	1%
`ssdeep`	1	1.00	1.00	0.95	0.59	0.06	0.00	0.00	0.00
	4	1.00	1.00	0.94	0.63	0.07	0.00	0.00	0.00
	16	1.00	0.99	0.94	0.62	0.05	0.00	0.00	0.00
	64	1.00	1.00	0.95	0.63	0.07	0.00	0.00	0.00
	256	1.00	1.00	0.95	0.66	0.07	0.00	0.00	0.00
	1024	1.00	1.00	0.93	0.64	0.07	0.00	0.00	0.00
`sdhash`	1	0.61	0.43	0.06	0.00	0.00	0.00	0.00	0.00
	4	1.00	0.98	0.13	0.00	0.00	0.00	0.00	0.00
	16	1.00	0.98	0.87	0.53	0.00	0.00	0.00	0.00
	64	1.00	1.00	1.00	1.00	0.83	0.11	0.00	0.00
	256	1.00	1.00	1.00	1.00	1.00	1.00	0.76	0.01
	1024	1.00	1.00	0.93	0.64	1.00	1.00	1.00	0.32
`mrsh-v2`	1	0.09	0.04	0.01	0.00	0.00	0.00	0.00	0.00
	4	0.95	0.72	0.28	0.02	0.00	0.00	0.00	0.00
	16	1.00	0.95	0.25	0.00	0.00	0.00	0.00	0.00
	64	1.00	0.96	0.84	0.61	0.05	0.04	0.04	0.00
	256	1.00	1.00	1.00	1.00	0.86	0.24	0.03	0.01
	1024	1.00	1.00	1.00	1.00	1.00	0.99	0.58	0.00
`mvhash`	1	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	4	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	16	0.60	0.01	0.00	0.00	0.00	0.00	0.00	0.00
	64	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	256	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	1024	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

The behavior of the tools on this test is clearly correlated with their performance on the prior test but the relationships are a bit more complex.

For `ssdeep`, we need at least 20% commonality to achieve a better-than-50% TPR, and at least 30% to achieve reliable (93%+) detection. As with the fragment detection, the results are consistent across file sizes and depend on the *relative* size of the common fragment.

For `sdhash` and `mrsh-v2`, the usual problem at 1 KiB persist and, just like in the fragment case, the performance improves as the size of the source file grows. However, considering the *absolute* size of the common fragment, the algorithms need a bigger common block compared to the fragment test. This is likely due to alignment issues as the similarity digests consist of a sequence of Bloom filters, such one representing (on average) 9-10 KiB (`sdhash`) or 25-26 KiB (`mrsh-v2`) of the source data. Depending on the location of the sample, the data could map to two, or three, separate filters and match could be numerically diluted during comparison.

`mvhash` operates worse for 1-16 KiB and perfect for files $\geq$ 64 KiB at a first look. However, studying the detailed results (see Appendix B) actually shows that the algorithm fails this test completely as the false positive rate (FPR) is also 100%. One reason might be the chosen setting `-t 7` which only considers 7 bits per byte[4]. Nevertheless, this behavior needs a more detailed analysis.

Alignment. Recall that the alignment test is designed to evaluate the robustness of the algorithm with respect to data alignment. To perform the test, we compare an original file f_1 to a file f_2, which consists of the entirety of f_1 prefixed by a random byte string of length $X = \{25\%, 50\%, 100\%, 200\%\}$. Table 6.15 shows the results and lead to the following observations.

Table 6.15.: True positive rates as a function of file size and the alignment.

		25%	50%	100%	200%			25%	50%	100%	200%
`ssdeep`	1	1.00	1.00	0.96	0.92	`mrsh-v2`	1	0.24	0.24	0.24	0.24
	4	1.00	1.00	0.97	0.93		4	1.00	1.00	1.00	1.00
	16	1.00	1.00	0.98	0.93		16	1.00	1.00	1.00	1.00
	64	1.00	1.00	0.97	0.93		64	1.00	1.00	1.00	1.00
	256	1.00	1.00	0.98	0.94		256	1.00	1.00	1.00	1.00
	1024	1.00	1.00	0.96	0.93		1024	1.00	1.00	1.00	1.00
`sdhash`	1	0.86	0.87	0.89	0.90	`mvhash`	1	1.00	0.95	0.55	0.01
	4	1.00	1.00	1.00	1.00		4	1.00	1.00	0.96	0.01
	16	1.00	1.00	1.00	1.00		16	1.00	1.00	0.97	0.93
	64	1.00	1.00	1.00	1.00		64	1.00	1.00	1.00	1.00
	256	1.00	1.00	1.00	1.00		256	1.00	1.00	1.00	1.00
	1024	1.00	1.00	1.00	1.00		1024	0.96	0.00	0.00	0.00

`ssdeep` performs almost perfect up to a shift of 100% and a little bit less precise for 200%. The latter case means that the original file is 1/3 of the modified file which is the limit of `ssdeep`. If the alignment increases further, the detection rate will decrease further.

`sdhash` and `mrsh-v2` deal perfect with all sizes except 1 KiB where `mrsh-v2` performs worse than `sdhash`. This is an edge case for the tools, which are optimized for targets with minimum size of about 1400 bytes (full network packet).

Again the problem with the high false positive rate of `mvhash` remains and therefore we do not discuss it.

White-noise resistance. In the random-noise resistance test, we attempt to correlate a file f_1 and a randomly disturbed version of it – f_2. In addition to file size, we vary

---

[4] In order to make all test results comparative, we must have the same configuration for all tests; `-t 7` is needed for the real world assessment.

the fraction X of bytes modified, where $X = \{0.5\%, 1.0\%, 1.5\%, 2.0\%, 2.5\%\}$ of f_1's size. Table 6.16 shows the results and lead to the following observations.

Table 6.16.: True positive rates as a function of file size and the random noise.

		0.5%	1.0%	1.5%	2.0%	2.5%
`ssdeep`	1	1.00	0.94	0.77	0.56	0.39
	4	0.60	0.15	0.03	0.01	0.00
	16	0.02	0.00	0.00	0.00	0.00
	64	0.00	0.00	0.00	0.00	0.00
	256	0.00	0.00	0.00	0.00	0.00
	1024	0.00	0.00	0.00	0.00	0.00
`sdhash`	1	0.64	0.60	0.40	0.17	0.05
	4	1.00	1.00	0.72	0.09	0.07
	16	1.00	1.00	0.81	0.05	0.00
	64	1.00	1.00	0.96	0.03	0.00
	256	1.00	1.00	0.98	0.03	0.00
	1024	1.00	1.00	0.89	0.00	0.00
`mrsh-v2`	1	0.20	0.12	0.05	0.02	0.01
	4	0.92	0.27	0.03	0.01	0.00
	16	0.99	0.04	0.00	0.00	0.00
	64	1.00	0.14	0.03	0.01	0.01
	256	1.00	0.06	0.01	0.00	0.00
	1024	1.00	0.00	0.00	0.00	0.00
`mvhash`	1	0.10	0.01	0.00	0.00	0.00
	4	0.06	0.00	0.00	0.00	0.00
	16	0.25	0.00	0.00	0.00	0.00
	64	1.00	1.00	1.00	1.00	1.00
	256	1.00	1.00	1.00	1.00	1.00
	1024	1.00	1.00	1.00	1.00	1.00

In prior work [4], we have shown that `ssdeep`'s noise resistance is a function of the absolute number of changes made to the source data. The current results confirm this. `ssdeep` performs well for small files and small percentage values. For instance, 0.5-1% of 1024 bytes leads to 5-10 changes; `ssdeep` can clearly tolerate the disturbance as evidenced by the true positive rates of 100% and 94%. Somewhere around 20-22 modifications the TPR drops below 50%, and around 80 it becomes effectively zero.

Apart from the always-problematic 1 KiB case, `sdhash` deals well with random noise of up 1.0% for all file sizes – the true positive rate is 100%. The 1.5% case triggers a fluctuating detection rate between 72% (4 KiB) and 98% (256 KiB) and shows that the tool is starting to fail. At 2.0% and up, the TPR crashes to zero indicating that the tool's ability to tolerate noise has been exhausted.

`mrsh-v2` operates worse than `sdhash` as it only performs well for the 0.5% noise and then rapidly goes down to a TPR of 0. This comes from the feature selection. While `mrsh-v2` divides an input into chunks (all bytes are represented in the similarity digest), `sdhash` selects features of 64 bytes which may have gaps in between (not all bytes influence the similarity digest).

Again the problem with the high false positive rate of `mvhash` remains and therefore we do not discuss it.

6.3.4. Precision & recall on real world data

This section presents the results from applying our test methodology from Sec. 6.2.3 to analyze the performance `ssdeep`, `sdhash`, `mrsh-v2` and `mvhash` on the basis of the t5-corpus.

One challenge that may not be immediately obvious is that a complete, all-pairs comparison run requires a non-trivial number of comparisons – a set of n files results in $\frac{n(n-1)}{2}$ comparisons, which corresponds to $9,930,196$ aLCS-comparisons for t5. Although it takes only $\approx 425\,ms$ per comparison, such work would clearly be impractical without parallel execution. Fortunately, such a workload is readily parallelizable and our implementation takes full advantage of that. In our tests, a 48-core, 2.6 GHz AMD Opteron server needed 1466 min ($\approx$ 24 hours) to generate and compare all aLCS-digests.

The rest of this section is divided into the following parts. First, we give a general overview of the detection rates of the different approaches. The next three sections discuss the false positives, false negatives and true positives, respectively. Finally, the last section shows the differences in performance for the containment and resemblance usage scenarios.

Note, the following results are for the relative case described in the testing methodology in Sec. 6.2.3 on page 104. On page 129 we briefly discuss the baseline results for an absolute alcs L_a.

Baseline results. First, we present the baseline case where $t = 0$. In other words, we define any approximate matching score greater than zero as a positive result, and any zero score as a negative result. This is the lowest barrier for matching algorithms to jump over as they simply need to match the positive/negative behavior of the baseline aLCS measure, with no additional expectations of the exact value of the score.

Using the definitions from the last paragraph, the observed statistics from the experiments are shown in Table 6.17 and lead us to the following initial observations:

- *Precision.* In absolute terms, `sdhash` yields the largest number of true positives, while `mvhash` is with the lowest number of false positives; `mrsh-v2`'s true positives fall right in the middle but the false positives are much higher than the other three. This results in reasonably high precision for `ssdeep`, `sdhash` and `mvhash`, and a relatively low one for `mrsh-v2`.

- *Recall.* Due to the high number of false negatives across the board, the recall rates are quite low. In relative terms, `sdhash` holds a considerable advantage over `ssdeep`, `mrsh-v2` and `mvhash`.

- *TNR & Accuracy.* Due to the very high ratio of negative to positive results, these measures do not provide any meaningful differentiation among the tools.

- *F-scores.* In these combined measures, `sdhash` holds a consistent $1.47-1.50\times$ performance advantage over `mrsh-v2`, $5.50-5.75\times$ over `ssdeep` and $8.85-9.00\times$ over `mvhash`.
- *MCC.* The measure also puts `sdhash`'s performance ahead by a $2.25-4.5\times$ margin; interestingly, `mrsh-v2` and `ssdeep` swap places suggesting that `mrsh-v2`'s lower precision is having a bigger effect on MCC than on F-scores. `mvhash` is in between `ssdeep` and `mrsh-v2`.

Table 6.17.: Baseline approximate matching results for $t = 0$.

	`ssdeep`	`sdhash`	`mrsh-v2`	`mvhash`
TP	951	5,474	1,335	611
FP	15	790	9,011	11
TN	9,472,047	9,471,272	9,463,051	9,472,051
FN	457,183	452,660	456,799	457,523
Precision	0.98447	0.87388	0.12904	0.98232
Recall	0.00010	0.00058	0.00014	0.00006
TNR	1.00000	0.99992	0.99905	1.00000
Accuracy	0.95396	0.95434	0.95309	0.95392
F_1	0.00020	0.00115	0.00028	0.00013
F_2	0.00013	0.00072	0.00018	0.00008
$F_{0.5}$	0.00050	0.00288	0.00070	0.00032
MCC	0.04412	0.09913	0.01276	0.03531

Analysis of false positives. Let us now consider the false positive behavior of the tested tools in detail. Fig. 6.4 shows the empirical probability distribution of the approximate matching scores S_h for which the respective tool has yielded a false positive. Both `mrsh-v2` and `sdhash` show a highly desirable behavior – the FP scores are heavily concentrated close to zero. Indeed, the cumulative probability for scores in the $1-10$ range constitute 99.9% and 96.6% of all FP for `mrsh-v2` and `sdhash`, respectively; `ssdeep`'s results are uniformly distributed throughout the 32-85 range. The scores for the FP for `mvhash` are all smaller than 17.

Analysis of false negatives. The breakdown of false negative results show virtually identical distribution of L_r scores for all four tools. This is clearly due to the overwhelming number of negatives, which render any differences across tools insignificant. The good news is that, although the false negatives are substantial in number, their L_r scores are heavily clustered around zero. This means that we can put a relatively tight and useful bound on what approximate matching tools might miss.

For example, assume that one of the tools, `sdhash`, returns a score of zero ($S_{sdhash} = 0$). Given its *negative predictive value* $NPV = {}^{TN}/_{(FN+TN)} = 0.954$, the result will be TN

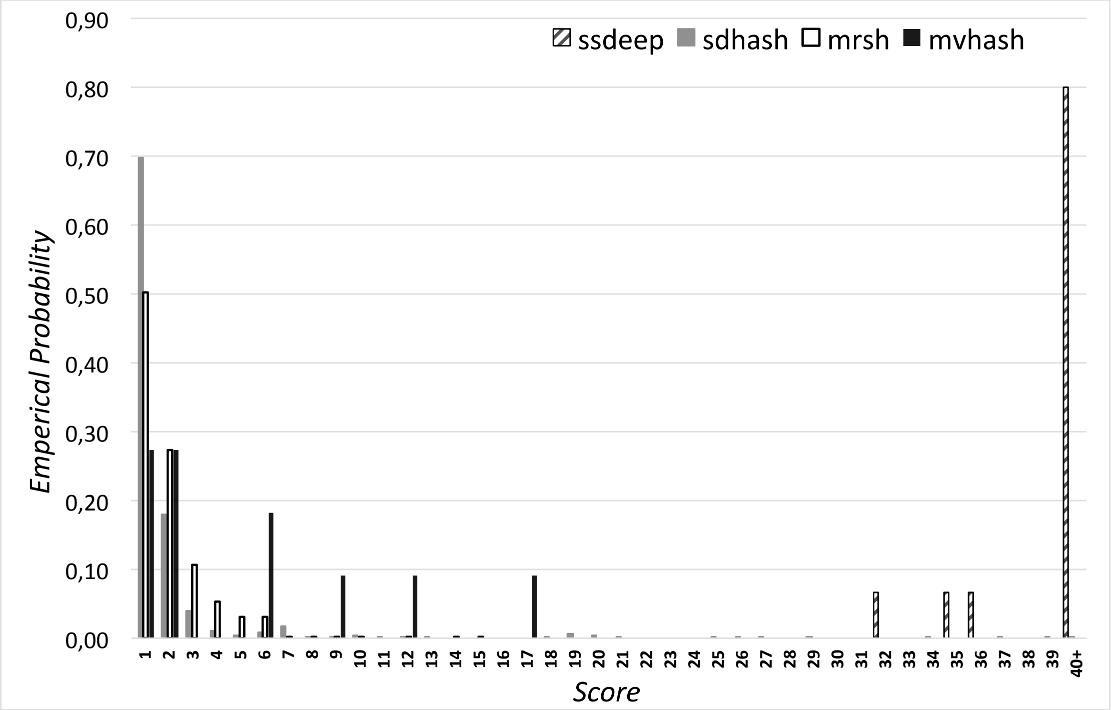


Figure 6.4.: Empirical probability distribution of S_h scores for approximate matching false positives.

in 95.4% of the time. Whenever it is not (4.6%), Figure 6.5 tells us that $\sim$ 98% of the time the L_r score would not exceed 15. Put together, the two observations tell us that $S_{sdhash} = 0$ implies 99.91%[5] certainty that the L_r score does not exceed 15.

Since the NPV for all four tools are similar, we can conclude that negative results from any of the tools are significant in that they allow us to bound the level of commonality that we may be missing with a very high level of certainty.

Analysis of true positives. The next question we would like to explore is: what is the correlation between true positive results (S_h) and the ground truth results (L_r)? To understand this behavior, we build the empirical probability distribution of the *difference* between the true score (as defined by L_r) and the similarity score; that is, $L_r - S_h$, for $h \in \{ssdeep, sdhash, mrsh, mvhash\}$ (see Fig. **??**).

We can see that `mrsh-v2`'s comes closest to having a classical Gaussion distribution that is symmetrical and fairly tight around the mean of zero; this implies that `mrsh-v2`'s positive results have about equal chance of being smaller, or larger than ground truth. Next, `sdhash`'s distribution also has a bell-like shape but has a 'bulge' and slight bias to the right of zero, implying that `sdhash` scores are somewhat more likely to be smaller than aLCS's score. `ssdeep`'s distribution is massively skewed to the left of zero (89% of

---

[5] Out of the 4.6%, there are 98% under 15. Thus, $95.4\% + 4.6\% \cdot 0.98 = 99.91\%$.

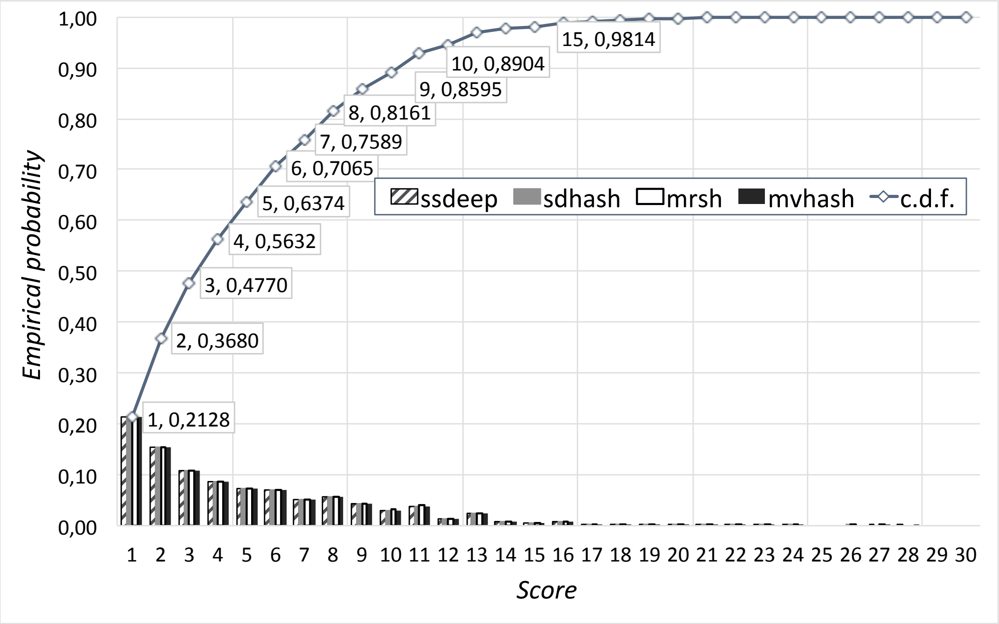


Figure 6.5.: Empirical probability distribution of L_r scores for approximate matching false negatives.

the mass) and shows no particular characteristic shape; still, the graph it tells us that we can view `ssdeep`'s score as an upper bound on the aLCS result. `mvhash`'s results are distributed over the whole x-axis and also does not show a characteristic shape. Thus, there is no correlation between the score and the aLCS.

Containment vs. resemblance. Having characterized the overall performance of the tools, we consider their behavior under the two basic usage scenarios – *resemblance* and *containment*. Following Broder's ideas [28], we try to approximate the informal notions of 'roughly the same' (resemblance) and 'roughly contained inside' (containment). For example, comparing two executable files similar in size is likely a resemblance query, whereas comparing a file against a RAM snapshot is clearly a containment query. However, we have no precise guidance as to where to draw the line between the two scenarios.

For this work, we chose a criterion based on the ratio of the file sizes. Namely, if the size of the bigger file is at least *two times* the size of the smaller one, we define this as a *containment* query; otherwise, it is a *resemblance* one. In other words, if more than one non-overlapping copy of one file can fit in the other, we assume the main interest to be *containment*. Evidently, if a similarity tool behaves the same way in both cases, we expect the performance metrics to remain stable across the two scenarios.

To establish a baseline, we use our ground truth (*gt*) results; Table 6.18 provides a summary. The first row (*gt-con*) provides the statistics for all containment cases (pairs of files); the second row covers all resemblance cases; the last row combines the results. The first column (TP) provides the number of true positives, followed by TP_{ratio} which

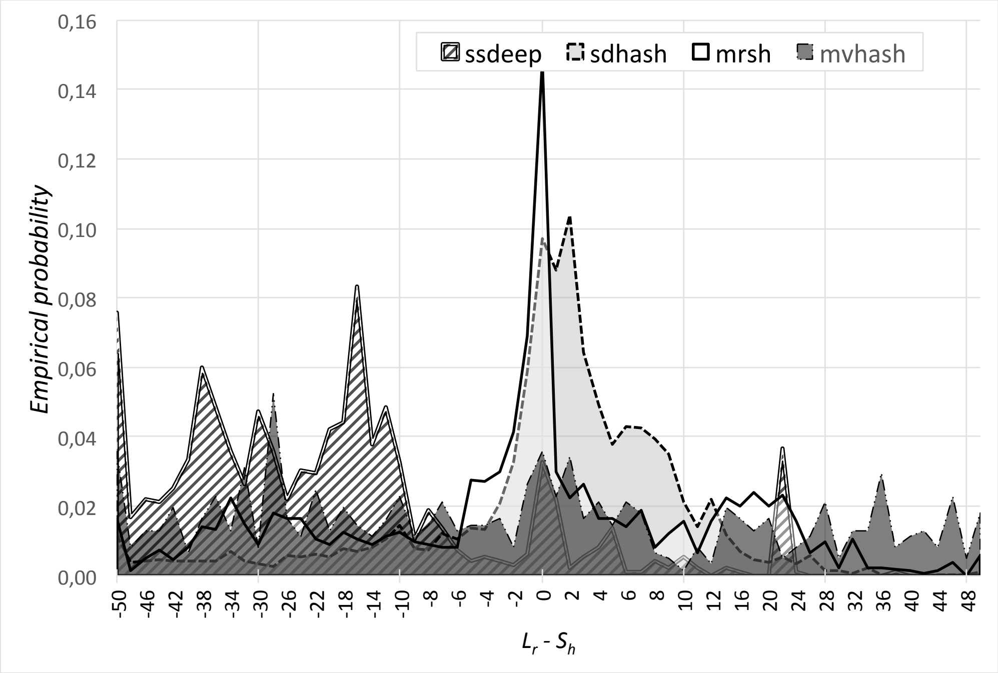


Figure 6.6.: Empirical probability distributions of $L_r - S_h$, for $h \in \{ssdeep, sdhash, mrsh, mvhash\}$.

gives the fraction of true positives for the particular case relative to the total number of true positives. The *TN* and TN_{ratio} provide analogous numbers with respect to true negatives; the last two columns provide totals. In simple terms, we see that about 78% of the pairs fall into the containment and 22% into the resemblance cases.

Table 6.18.: Ground truth statistics for containment/resemblance cases

	TP	TP_{ratio}	TN	TN_{ratio}	Total	$Total_{ratio}$
gt-con	354,914	0.775	7,382,141	0.779	7,737,055	0.779
gt-res	103,220	0.225	2,089,921	0.221	2,193,141	0.221
gt	458,134	1.000	9,472,062	1.000	9,930,196	1.000

Tables 6.19 and 6.20 present the statistics for the evaluated similarity tools. From the former, we can see that `ssdeep` and `mvhash` have a notably different behavior from both the baseline and the other two tools. Namely, about 92% of their matches come from the resemblance case; this is a logical result of their design, which makes the resolution of the similarity digest a function of file size. As file sizes draw apart, both tools simply lose the ability to compare them. `sdhash` follows a containment/resemblance ratio that is much closer to the baseline but is still tilted in favor of resemblance as the fraction of resemblance TP results is some 36.6% higher than in the ground truth case. `mrsh-v2` is very balanced but also favors the resemblance case.

Considering the information retrieval metrics in Table 6.20, one important observation is that all four tools yield better results for resemblance over containment. For `mrsh-v2` and `sdhash` the improvement is by 50-100%, while for `ssdeep` and `mrsh-v2` it is up to 40 times. Among the tools, the relative performance ratios remain comparable to the ones presented earlier in Table 6.17 with `sdhash` outperforming across the board.

Table 6.19.: Basic containment/resemblance statistics by approximate matching tool.

	TP	TP_{ratio}	FP	FP_{ratio}	TN	FN
ssdeep-con	74	0.078	5	0.333	7,382,136	354,840
ssdeep-res	877	0.922	10	0.667	2,089,911	102,343
ssdeep	951	1.000	15	1.000	9,472,047	457,183
sdhash-con	3,472	0.634	497	0.629	7,381,644	351,442
sdhash-res	2,002	0.366	293	0.371	2,089,628	101,218
sdhash	5,474	1.000	790	1.000	9,471,272	452,660
mrsh-con	576	0.431	3,940	0.437	7,378,201	354,338
mrsh-res	759	0.569	5,071	0.563	2,084,850	102,461
mrsh	1,335	1.000	9,011	1.000	9,463,051	456,799
mvhash-con	54	0.088	2	0.182	7,382,139	354,860
mvhash-res	557	0.912	9	0.818	2,089,912	102,663
mvhash	611	1.000	11	1.000	9,472,051	457,523

Baseline results absolute alcs. This paragraph presents the baseline case where $t = 0$ and $L_a \geq 2048$ bytes. Thus, again, we define any approximate matching score greater than zero as a positive result, and any zero score as a negative result. The observed statistics from the experiments are shown in Table 6.21 and lead us to the following initial observations:

- *Precision.* `mrsh-v2` yields the largest number of true positives, while the amount of true positives for the remaining algorithms decrease a little bit in contrast to the relative case. `mvhash` has the lowest number of false positives followed by `ssdeep` and `mrsh-v2`. This results in reasonably high precision for `mvhash` and `mrsh-v2` and a lower precision for `ssdeep` and `sdhash`.
- *Recall.* Due to the high number of false negatives across the board, the recall rates are quite low. In relative terms, `mrsh-v2` hold a considerable advantage over `ssdeep`, `sdhash` and `mvhash`.
- *TNR & Accuracy.* Due to the very high ratio of negative to positive results, these measures do not provide any meaningful differentiation among the tools.

While the results for relative and absolute alcs scores are almost equal for `ssdeep`, `sdhash` and `mvhash`, the rates are very different for `mrsh-v2`. To sum it up, the first

Table 6.20.: Performance measures by scenario and approximate matching tool.

	Precision	Recall	F_1	F_2	$F_{0.5}$	MCC
ssdeep-con	0.93671	0.00001	0.00002	0.00001	0.00005	0.01361
ssdeep-res	0.98873	0.00042	0.00084	0.00052	0.00209	0.08944
ssdeep	0.98447	0.00010	0.00020	0.00013	0.00050	0.04412
sdhash-con	0.87478	0.00047	0.00094	0.00059	0.00235	0.08976
sdhash-res	0.87233	0.00096	0.00191	0.00120	0.00476	0.12612
sdhash	0.87388	0.00058	0.00115	0.00072	0.00288	0.09913
mrsh-con	0.12755	0.00008	0.00016	0.00010	0.00039	0.00943
mrsh-res	0.13019	0.00036	0.00073	0.00045	0.00180	0.02026
mrsh	0.12904	0.00014	0.00028	0.00018	0.00070	0.01276
mvhash-con	0.96429	0.00001	0.00001	0.00001	0.00004	0.01181
mvhash-res	0.98410	0.00027	0.00053	0.00033	0.00133	0.07109
mvhash	0.98232	0.00006	0.00013	0.00008	0.00032	0.03532

Table 6.21.: Baseline approximate matching results for $L_a \geq 2048$ bytes and $t = 0$.

	`ssdeep`	`sdhash`	`mrsh-v2`	`mvhash`
TP	787	4,989	9.564	594
FP	179	1275	782	28
TN	9,500,344	9,499,248	9,499.741	9,500,495
FN	428,883	424,681	420,106	429,079
Precision	0.81470	0.79646	0.92442	0.95498
Recall	0.00008	0.00052	0.00101	0.00006
TNR	0.99998	0.99987	0.99992	1.00000
Accuracy	0.95679	0.95710	0.95762	0.95679

three algorithms work reliable for relative similarities and the latter one works better for absolute similarities. Thus, the perfect algorithm depends on the needs of an investigator and the working field.

6.4. Implementation details of the tests

The test results from the previous section were obtained by multiple scripts and implementations. More precisely, we can split the testing into three parts: efficiency, synthetic data and real world data.

Note, the following is a brief description of `FRASH` v1.0 which is a collection of scripts. We are currently working on a C++-framework that will include all tests and ease its usage. Once `FRASH` v2.0 is published, it will be easier to use the different test.

Efficiency. Obtaining the overall benchmark data is the simplest test, as we can use basic Linux commands. Thus, we measured the runtime efficiency by using the `time`-command. For instance,

```
$ time mrsh t5/* > mrsh_digests
real 0m13.300s
user 0m12.980s
sys  0m0.310s .
```

To define the compression, we analyzed the digest files where we considered the digest representation (e.g., Base64 vs binary) and removed unnecessary information like file names or paths.

Sensitivity & robustness and precision & recall on synthetic data. Since these two test categories are very similar, we combined them. Both tests use the same kind of modifications (i.e.,, fragment, alignment, white-noise and single-common-block). The main difference is that the sensitivity & robustness tests use the t5-corpus while the precision & recall on synthetic data creates random files. To perform these tests, we implemented a C++ tool which will be part of `FRASH` v2.0.

Before running the tests, the current version needs some configuration which can be set in class `TestConfig`. First, one has to choose the paths for the original and modified files. Next, one has to set the modification (e.g., fragment test) including all desired options). If no files are available, this means the tool first has to generate the input and therefore needs the ‘amount of random files’ and the ‘file size’.

Once the configuration is made, the actual testing consists of the following five steps:

1. *Prepare files* checks and generates files if necessary. For instance, if a set of original files is available, we only have to generate the modifications of the set. If no files are available, we have to create both. In the best case, we only have read in the file names and do not have to create any files.

2. *Generate file pairs* returns a list of necessary comparisons. The sensitivity & robustness test, for example, only compares the original file against its modified file while the precision & recall test compares all-against-all.

3. *Run comparison* is the actual processing of the files. Thus, all file pairs are processed (i.e., hashed and compared) using all desired algorithms and the final scores are generated. In order to reduce the run time, we parallelized this phase.

4. *Analyze results* is done by the `TestResultProcessor` which processes/administrates the comparison results including the match scores. For instance, we generate the histograms which are then used to create further statistics like true positive rate, false positive rate and so on.

5. *Print results* print the results to the screen and write them into a file.

In the following we briefly describe the most important classes. To handle the different parameters and outputs of the algorithms, each algorithm has its own class which is inherited from the base class `HashAlgorithm`. Furthermore, all modifications have their own classes to handle the different test cases and options.

The actual processing is solved by `BaseTest` which handles the comparison, folder structure or storing results. Depending on the test scenario, `SensitivityRobustnessTest` generates the genuine file pairs while `PrecisionRecallTest` generates all file pairs. In order to distinguish between genuine and impostor, `FilePair` stores the filenames (two genuies matches will have the same filename). `Histogram` has a list of all results, analyzes the scores and write the histograms into files. Once the histograms are ready, `statistics` computes additional values like matching rates.

Real world data. To study the behavior of approximate matching algorithms for real world data, we implemented several bash scripts; one per algorithm and case. Thus, there are three directories per algorithm, e.g., `mvhash`, `mvhash`-con and `mvhash`-res. The first is the general case, followed by the containment and resemblance cases.

Next, we describe the procedure for `mvhash`. There is a shell-script (i.e., mvhash.sh) in each directory, which requires the ground truth (i.e., the aLCS files from the `alcs` directory) and the results from the evaluated algorithm, the `gen` comparison. As all approximate matching algorithms produce a different output, we normalize it to a three column format and sort it. Note, in order to test a new algorithm, the command for 'reduce to three-column format' must be adjusted in the `mvhash.sh` line 12.

Then, the starting point is the ordered `mvhash.join` file, which looks like this:

```
000001.doc-000006.doc 006
000001.doc-001855.doc 009
000001.doc-001867.doc 003
...
```

In a second step, the aLCS file and the `mvhash.join` file are combined to a file called `alcs-mvhash.join`, which looks like this (no action is required, the `mvhash.sh` script does this itself):

```
000002.doc-000003.doc 044 38
000004.doc-000005.doc 037 2
000004.doc-000696.doc 042 4
...
```

where we have the file pairs in the beginning followed by the aLCS-score and the `mvhash` match score, respectively. If both numbers are > 0, this is a true positive. If the aLCS-score (column 2) is 0 and the match score is > 0, we have a false positive.

To analyze the negative matches, we studied the pairs that are unique in the ground truth. All pairs in the ground truth with a score > 0 but not listed in the `mvhash.join` file are false negatives. File pairs having an aLCS score of 0 and are not listed in `mvhash.join` are true negatives. The final output of the script is therefore the pairs of TPs, FPs, TNs and FNs including histograms.

Generating the containment and resemblance works similarly but instead of using the general ground truth, there are two .join files that split the set into the 'con' and 'res' cases (`alcs/alcs-con.join` and `alcs/alcs-res.join`). Before running the case script (e.g., `mvhash-con.sh`), one has to copy `mvhash.join` file from the original case, to the `mvhash-con` folder and rename it to `mvhash-all.join` To split a specific file, like `alcs-mvhash.join`, we did

```
join alcs-mvhash.join ../alcs/alcs-con.join
join alcs-mvhash.join ../alcs/alcs-res.join
```

Obviously, the `/alcs/` output can be copied from the other cases. With respect to the scripts, the variables in the beginning have to be adjusted.

6.5. Summary

In this chapter we took the challenge of *automating* the process of characterizing approximate matching algorithms' behavior with respect to standard information retrieval metrics, such as precision & recall, using synthetic and real world data. The results are multiple scripts that form the open source testing framework `FRASH` v1.0.

Our framework currently includes three test classes called *efficiency*, *sensitivity & robustness* and *precision & recall*. The efficiency test comprises generation efficiency, comparison efficiency and space efficiency. Sensitivity & robustness is composed of four sub-tests named fragment detection, single-common-block correlation, alignment robustness, and white-noise-resistance and define the performance envelope of the algorithms. Furthermore, we designed a set of precision & recall tests that are algorithm-neutral.

The main difficulty for precision & recall is establishing the ground truth by some algorithmic means. Therefore, we decided to have two ground truths: synthetic data and real world data. In the later case, we propose the use of *longest common substring* (LCS) as a useful measure of commonality between two files. The problem with using LCS, is that its computation is relatively expensive and cannot be easily scaled to the degree necessary – the digital forensics community needs a testing and evaluation framework that can be routinely deployed by practitioners. Thus, we designed an efficient approximation called approximate longest common substring (aLCS) that places a lower bound on the size of the lowest common substring. The observed performance shows that aLCS is, indeed, a practical approach estimating the size of LCS for real-world files.

We should note that further work is required to relate common substrings to human-observable (and forensically relevant) artifacts. For example, we have not controlled for long strings of sparse data (e.g., all zeros) that, more likely than not, are not of forensic interest.

Our evaluation showed that each of the algorithms has a distinct operational range and analysts must understand the relationships between input parameters and result significance in order to operate the tools correctly. Therefore, having a rigorous testing framework, such as `FRASH`, is critical to evaluating and calibrating various approximate matching algorithms.

As a next step, we would like to integrate this different scripts into a single framework `FRASH` v2.0.

7

Towards signature based similarity detection in forensic investigations

One of the biggest challenges in computer crime is coping with the huge amounts of data. To handle all of this, the forensic community developed investigation models to assist law enforcement [70] which mainly describe where agents should start their investigation. For instance, in 2006 Rogers presented the computer forensics field triage process model (CFFTPM) which is promoted to be "an on-site or field approach for providing the identification, analysis and interpretation of digital evidence in a short time frame" [78]. While this model precisely describes how to approach a computer crime, the author states that steps could be very time consuming due to the amount of data. Hence, it is important to reduce the amount of data to be inspected manually, by automatically distinguishing between relevant and non-relevant files. Thus, this chapter shows how hashing and approximate matching can support investigations.

The first section of this chapter presents a framework to reduce the time for hashing operations during forensic investigations using *prefetching* which was published in [20]. Instead of having several threads / cores hashing independently, we use one thread for reading and all remaining threads for hashing. The framework could be easily applied for all hashing algorithms or other approaches.

In Sec. 7.2 we compare the detection rates of the cryptographic hash function SHA-1 and the approximate matching algorithm `ssdeep` to demonstrate the benefits for investigations which was published in [26]. As expected, detection rates for `ssdeep` are significantly higher.

The third part gives an overview of semantic approximate matching followed by a brief comparison of algorithms (published in [19]). Besides a comparison of the same group (bytewise and semantic), we also present a sketchy comparison across groups to clearly demonstrate benefits and drawbacks.

Finally, we show a sample use case of how to integrate hashing and approximate matching into existing investigation models in Sec. 7.5. The summary at the end concludes this section.

7.1. Reducing time cost in hashing operations

The bottleneck of hashing operations during digital investigations is caused by the file handling. In order to improve this proceeding, we designed and implemented a mechanism called prefetching. The result is a *parallel framework for hashing* (PFH) which speeds up the proceeding as it uses multiple threads for different tasks: one for reading (I/O throughput) and the rest for processing. Compared to existing frameworks, we obtained a much better throughput. The framework can be downloaded on our web-page[1].

7.1.1. Framework overview or overview of multi-threading with prefetching.

The parallel framework for hashing (PFH) is written in C++ and utilizes OpenMP 3.1 for multi-threading (see next subsection). In contrast to traditional approaches where hash functions request and process a file, we implement a prefetching mechanism. A sketch of the overall approach is shown in Fig. 7.1.

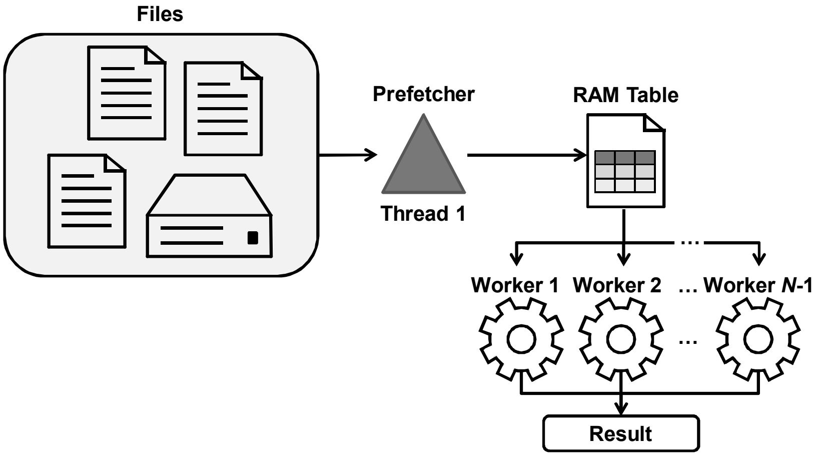


Figure 7.1.: Overview of the parallel hashing framework.

The main component is the *prefetcher* which handles the file reading and is responsible for the communication between hard disk and RAM. The general idea is that the critical resource, the hard disk, should 'work' all time. Thus, the prefetcher produces an ongoing file request.

All files are placed within the RAM which limits the amount of storage. All remaining threads are *workers* and proceed the files from the RAM using a pre-defined hashing algorithms. After hashing the file, there are several opportunities which are denoted by 'result'.

---

[1] `https://www.dasec.h-da.de/staff/breitinger-frank/#downloads` (last accessed 2014-03-17).

Depending on the computational efficiency of the hashing algorithm, there are two scenarios:

1. If the hashing algorithm is fast, the worker threads are faster than the prefetching process and thus the workers have to idle. However, the hard disk is at its limit and cannot process faster.

2. If the hashing algorithm is slow[2], the RAM table becomes full and cannot store any further files. Thus, the prefetcher starts to idle. In this case the prefetcher thread could turn into a normal worker and help to process the files in RAM table[3].

One might say that distributed systems may further increase the performance. As will be demonstrated later, the limiting resource is the underlying device and not the computational power of the modern system. Therefore, there will not be any improvement using distributed systems.

Our implementation is divided into two branches – simple multi-threading (SMT) and multi-threading with prefetching (MTP) – where our results focus on the latter approach. SMT was added to show the benefits of prefetching when compared to MTP and thus is only a side product.

Open MP. “The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer”[4].

Note, the framework also supports OpenMP 2.0, but with a decreased runtime efficiency due to ‘capture clause’ which is only available in OpenMP 3.1. The capture clause allows to copy a global variable into a local one and increment the global variable, in an atomic operation. In OpenMP 2.0 capture clauses can be replaced with critical sections, but this reduces runtime efficiency.

Command line parameters and operation modes. Before describing the details of our framework, we briefly introduce the command line options which allow a rough configuration. Let N denote the number of all processor cores of an architecture and let P be the number of desired prefetchers where $P < N$; default $P = 1$.

`c` - mode of framework operation [optional] (explained in the subsequent paragraph).

`d` - directory to be hashed or file with digests [is required].

---

[2] Of course, all hashing algorithms are supposed to be fast. However, some approximate matching algorithms like `sdhash` are multiple times slower than SHA-1 and thus we use the term slow.

[3] This functionality is future work. Currently the prefetcher never changes its role.

[4] `http://openmp.org/wp/about-openmp/` (last accessed 2014-03-01).

`r` - recursive mode for directory traversing [optional].

`p` - number of prefetching threads P [default is 1].

`t` - number of all threads [default is N].

`h` - hashing algorithm [default is `mrsh-v2`].

`m` - size of used memory in megabytes [default is 20 MB].

The default memory size of 20 MB is based on the max file size in the t5-corpus where the largest file is 16 MB. Currently there is one drawback: all files larger than the RAM table will be skipped. This issue will be fixed in an upcoming version.

In general the framework can operate in four different modes (`c`-option):

`HASH`: All files are hashed using the specified algorithm and the results are printed on the standard output [default].

`FULL`: The framework does an all-against-all comparison of all files in `directory`.

`<DIGEST>`: All files within `directory` are hashed and compared against `DIGEST` which is a single fingerprint.

If value of parameter `-d` is a fingerprint file, the framework will compare `DIGEST` against all fingerprints with the file - skipping the hashing stage.

`<FILENAME> <FILENAME>`: needs to be replaced by a path to a file containing a list of valid hash values. The framework hashes all files in `directory` and compares them against the list. If the signature is found within the list, it is a valid result[5]. This functionality is part of the framework, the implementation of the hash algorithm need not have an option for doing it.

Sample execution of the framework. The following command will execute the framework in the default mode, with a RAM table of size 256 MB. The t5-directory will be traversed recursively and all hashes are sent to the standard output. Since we did not specify `-t` and `-p`, the program has $P = 1$ prefetching thread and $N - P$ hashing threads. Recall, N is the number of available processor cores in the system.

```
$ pfh -c hash -m 256 -d t5 -r
```

Proceeding. On starting, there is an initializing part where the framework creates its 4 building blocks: options, hashing interface, RAM table and mode of operation. Next, the input parameters are parsed from the options class. In the following, the hashing interface is pointed to the chosen hashing algorithm and variables are set. The RAM table is created last, since it needs information from the hashing algorithm, to initialize its file filters.

---

[5] This is equal to the `-m` option of `ssdeep`.

The directory holding files is traversed and each file that passes the filter is added to the files-to-be-hashed-list. Currently the filter system concerns itself with the file size and access rights. For instance, files larger than the RAM table cannot be handled. Furthermore, an algorithm may need a minimum file size.

Next, we transfer the list structure to an array for easier thread processing. Knowing the amount of files that will be hashed, we initialize some of its internals to optimize its performance. The last part of the framework initialization sets the mode interface. Here, we point the comparison/result functions to the specified mode and set internal variables, if any.

The actual framework processing can be broken into three stages 1) reading/hashing files, 2) comparing hash values and 3) presenting results/scores, whereby only the first and second stage are executed with multiple threads while the third stage is sequential. Threads are created before the first stage and finalized at the end of the second stage. This way no time is lost for thread management (fork/join) during framework operation.

1. **Reading/hashing files.**
   - **SMT branch.** Each of the N threads put its file into the RAM table and hashes it. All threads continue until there are no more files in the queue.

     After being assigned to a role (reading or hashing), threads enter a 'work loop' for execution. Based on the return value, threads can change their role, e.g., if the RAM table is empty.
   - **MTP branch.** First, there is a thread assignment where every thread receives its role, i.e., we set P prefetchers. All $N - P$ threads are hashing threads. Currently all threads preserve their role over the whole runtime.
2. **Comparing hash values.** This phase is also executed in parallel using the OpenMP 'parallel for' clause, in which threads work on chunks of the global compare iterations. Scores from comparison are held in an array, because if threads print to screen, they have to synchronize and the speedup of parallelism is lost.
3. **Presenting results/scores.** At the end the file-path, hash value and score (if compare mode is used) are given to the standard output.

7.1.2. Implementation details

Besides the two branches SMT/MTP and the operation modes, the framework mainly consists of two objects named RAM table and hashing interface as shown in Fig. 7.2. All objects are explained in the following.

SMT and MTP. This paragraph describes how to switch between the two branches: simple multi-threading (SMT) and multi-threading with prefetching (MTP). Although we could not find any case where SMT outperforms MTP, we describe how to use it for the sake of completeness.

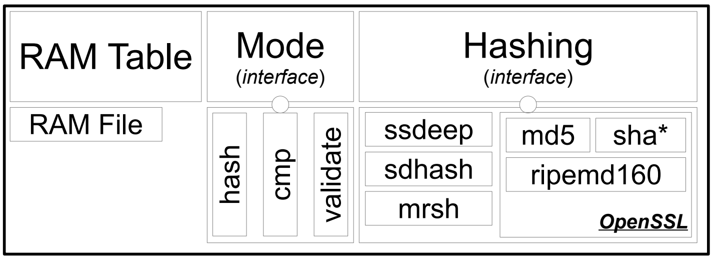


Figure 7.2.: Objects of the framework.

In order to change the branch, there is a configuration file called `configure.ac`. This is a template which is used by the configuration script when automake is executed. There are three options:

`-without-prefetching` disables prefetching of files and thus sets the branch to SMT (default: no and thus MTP mode).

`-with-timing` enables timing (default: no). Supported times are total, compare, hashing, accumulated time for waiting for RAM and file, reading from disk. It also provides throughput for hashing (MB/s) and comparing (items/s).

`-with-stats` enables statistics (default: no). Currently only two state variables are added, waiting for a file and waiting for space in ram table.

RAM Table. `RAM_table` is the class responsible for holding files and synchronizing threads. Files are presented with the `ram_file` class, which provides functionality for reading files from the hard disk and processing them using the hash algorithm interface. `ram_table` uses two semaphores[6] for realizing the producer/consumer model. One semaphore is used for waiting for free space in RAM table and the other for waiting for available/prefetched files in RAM table. POSIX and Windows semaphores are supported through macros expanded during compilation.

The processing of the files in the table is based on two indicies called fi and pi. fi is the amount of files within the table and set by the prefetcher, i.e., after every insertion fi increases by one. pi is the index of the worker threads. Thus, every time a worker thread fetches a new file from the table pi increases. As a consequence, if $pi = fi$, threads have to wait for data.

To avoid race hazards, we use the OpenMP 3.1 capture clause. Thus, a thread can take the current index and increase the global index in a single atomic operation. This way threads work with RAM files without the need for locking or critical sections.

Interfaces. The framework accesses all algorithms and modes through self-made interfaces and therefore every developer can add own hashing algorithm with a few lines of

---

[6] `http://en.wikipedia.org/wiki/Semaphore_(programming)` (last accessed 2014-03-12).

code. Realizations of interfaces are written in an own `*.hpp` file and included in the interface implementation file.

The hashing interface `hash_alg.cpp` also provides two extensions for handling output: one for algorithms with character output and the other for byte output. The difference between these two are functions for printing and saving a digest buffer. For instance, MD5 results in a buffer holding a byte array which needs to be converted to string.

Member variables of the class are:

Type of output - could either be hex or string.

Length of hash digest - is used to print the hash value.

Minimum file size - is necessary as some hashing algorithms have a minimum file size requirement. For instance `ssdeep` requires 4096 bytes.

Listing 7.1 and Listing 7.2 show the necessary changes for adding the `ssdeep` algorithm.

1. All hashing algorithms are implemented in their own file with the name `hash_alg_NAME.hpp`.

```
class hash_alg_ssdeep: public hash_alg_char_output{
public:
  int  hash(uchar *in, uint inlen, uchar **out){
    *out = get_out();
    return (NULL == fuzzy_hash_buf_r((const uchar*)in, inlen, *out))
           ? -1: FUZZY_MAX_RESULT;
  };

  int  cmp(uchar *a, uchar *b, uint len){
    return fuzzy_compare_r(a,b);
  };

  hash_alg_ssdeep(): hash_alg_char_output(){
    type             = HA_SSDEEP;
    max_result_size =
    hash_digest_size = FUZZY_MAX_RESULT;
    min_file_size   = SSDEEP_MIN_FILE_SIZE;
  };
};
```

Listing 7.1: Framework extension for `ssdeep`.

2. Add new case in interface initialization function for `ssdeep`.

```
if( 0 == htype.compare(0, 6, "ssdeep")){
    h = new hash_alg_ssdeep();
}
```

Listing 7.2: Initializing hashing interface for `ssdeep`.

Currently the framework includes several cryptographic hash functions and three approximate matching algorithms. We included MD5, SHA1, SHA2, SHA3 and RIPEMD160 from the OpenSSL library and added `ssdeep`, `sdhash` and `mrsh-v2` with source code.

The `mode.h` interface allows the framework to operate in different ways, after it's compiled. The interface itself consist of 3 virtual functions, that represent the 3 steps of the framework: hashing of files, comparing digests and printing results/digests.

Coding optimizations. The following optimizations reduce the number of buffer allocations during the execution and come with two advantages:

1. Pre-allocation of all digest buffers reduces execution time as there are less calls of `new[]`.
2. The Framework memory footprint is also reduced because all digest buffers are grouped into one linear buffer. For instance, the GNU C library uses a header (2 words - 8b/32bits and 16b/64bit systems) for each memory block. If we allocate a digest for MD5 (16b), we will have another 16b (on 64 bit systems) of OS administrative data (header)[7].

Both optimizations are only available for hashing algorithms with a static hash value length. In the case of `mrsh-v2` and `sdhash`, which have a variable hash value length, we can not allocate the linear digest buffer before hashing is done.

Future work. An improvement to the implementation will be the addition of a balancing function. By balancing we mean changing the order of files, in which they are processed, to reduce waiting for free table space and fragmentation (empty space in table). A simple example is shown below.

```
A(8), B(4), C(3), D(2), E(4), F(5) #Files order
TBL(0/10) #Table of size 10 with 0 space used
------------------------------------
T1:PREF(A) -> TBL(8/10)
T1:PREF(B) -> TBL(8/10) -> WAIT(4) #Wait because only space of 2 is available
T2:HASH(A) -> TBL(0/10)
------------------------------------
A(8), D(2), B(4), F(5), C(3), E(4) #Files order after balancing
```

Listing 7.3: RAM table balancing example.

7.1.3. Experimental results

The assessment of our framework is based on the t5-corpus and `ssdeep`, `sdhash` and `mrsh-v2`. All binaries were compiled using the same compiler and configuration options: `-g0` to disable debugging, `-O2` to enable second level of optimization and `-march=native` to allow usage of CPU specific instructions.

---

[7] Example is taken from `http://lwn.net/Articles/257209/` at the end of Sec. 7.3; last accessed 2013-05-09.

The test environment was a server having the following benchmark data:

```
CPU: 2xIntel(R) Xeon(R) E5430 2.66GHz x 4 cores
HDD: Seagate ES Series 250GB(SATA2) 8MB Cache 7200RPM
RAM: 8x2GB DDR2 FB-DIMM 667 MHz
KERNEL: Linux 2.6.32-279.11.1.el6.x86_64
GCC: gcc-4.4.6-4.el6.x86_64
```

Since our framework improves the whole processing, we measured the real-time using the Linux `time`-command.

Overall runtime efficiency. This paragraph demonstrates the general improvement of using the framework where we compared the original implementation against MTP. Both tests set `-t 2` which indicates one prefetching thread and one working thread.

Test one (T1) analyzes `ssdeep` in detail. The results are listed in Table 7.1. Using SMT improves the basic algorithms by approximately 17.5% which is in contrast to our expectations that it halves the time. The MTP proceeding shows an improvement of almost 40%. The lower speedup of SMT is due to the lack of data in RAM. Having two threads means there will be two times more requests for file data, but still having the same disk throughput. This way threads are being underfed and they have to idle. In the case of MTP we have a linear system – one reader and one hasher – which reduces the idle times of each thread.

Table 7.1.: T1: Runtime efficiency with ssdeep using standard output.

	Time	Difference	Terminal command
original	83.67 s	100.00%	$ ssdeep -r t5
SMT	69.05 s	82.52%	$ pfh -d t5 -t 2 -h ssdeep
MTP	52.19 s	62.37%	$ pfh -d t5 -t 2 -h ssdeep

Test two shows the improvement for different algorithms with the same configuration. All output was sent to `/dev/null` to eliminate any execution time deviation caused by printing. Table 7.2 shows the results.

The main result of T2 is that prefetching is very useful for approximate matching algorithms which are computationally more expensive than cryptographic hash functions. Using MTP we achieved similar runtimes for all algorithms expect `sdhash`. This shows that the limiting factor in this case is the underlying hardware.

Table 7.2.: T2: Runtime efficiency of different hashing algorithms.

	`MD5`	`SHA-1`	`mrsh-v2`	`ssdeep`	`sdhash`
original	51.65 s	52.35 s	75.61 s	83.67 s	145.38 s
MTP	51.74 s	51.64 s	51.79 s	52.19 s	89.09 s

Table 7.3 presents the results for T3 which runs in `FULL` mode. Thus, besides the hash value generation there is also an all-against-all comparison.

In case of `ssdeep` which is described in rows 1 and 2 we obtained an improvement of almost 45%. For `sdhash` (rows 3 and 4) the results are even better where we stopped the all-against-all comparison after 69 min (we assume that this was a bug of the used version 2.3) and MTP mode only needed 186 s.

Table 7.3.: T3: Runtime efficiency with of `FULL` mode [default `t=2`].

	Time	Difference	Terminal command[8]
`ssdeep`	119.21 s	100.00%	$ ssdeep -d -r t5
MTP	68.34 s	57.33%	$ pfh -c full -d t5 -t 8 -m 128
`sdhash`	> 69 min	-	$ sdhash -r -g -p 8 t5
MTP	186,56 s	-	$ pfh -c full -d t5 -t 8 -m 128

Impact of multiple cores. In the following we discuss the influence of multiple cores. Thus, we invoke the framework by

```
$ pfh -h ALG -c hash -d t5 -m 256 -t XX > /dev/null
```

where `XX` is the amount of cores/threads and `ALG` the used algorithm.

Our test includes two runs denoted by R1 and R2 which are shown in Table 7.4 and Table 7.5, respectively. The peculiarity is that we performed both runs in immediate succession and thus the files where still cached in R2.

R1 demonstrates that multiple cores are especially important for slower algorithms like `sdhash`. For fast algorithms, e.g., `mrsh-v2`, it does not scale well, as the underlying hardware is too slow. R2 simulates fast hardware as all files are cached. Due to the fact that all files are cached, the prefetcher thread is dispensable and thus SMT is faster.

Table 7.4.: R1: Runtime efficiency having different amount of threads.

		t=2	t=4	t=8
`mrsh-v2`	SMT	64.03 s	66.39 s	67.14 s
	MTP	51.79 s	51.81 s	52.02 s
`sdhash`	SMT	89.33 s	72.03 s	68.14 s
	MTP	89.09 s	51.90 s	52.08 s

As a conclusion we can say that in the first case the underlying hardware is too slow. More precisely, the hard disk is too slow, the prefetcher thread cannot fill the RAM table and thus the worker threads have to idle. Having a SSD device or RAID system it should scale better because of higher throughputs.

Impact of different memory sizes. This section shows the impact of different RAM table sizes wherefore we invoke the framework by

---

[8] We removed the `-h` option from both `pfh` commands for a better readability.

Table 7.5.: R2: Runtime efficiency having amount of threads and cached data.

		t=2	t=4	t=8
`mrsh-v2`	SMT	10.15 s	5.30 s	2.92 s
	MTP	17.68 s	6.23 s	2.90 s
`sdhash`	SMT	48.42 s	24.98 s	11.83 s
	MTP	88.15 s	31.12 s	15.07 s

```
$ pfh -h mrsh -c hash -d t5 -m XX -t 4 > /dev/null
```

where `XX` is the amount of memory in megabytes for the RAM table. Table 7.6 shows that the size of the RAM table does not influence the runtime efficiency which was also discussed on page 137.

Table 7.6.: Runtime efficiency having different memory sizes.

	m=128	m=256	m=512
SMT	66.36 s	66.39 s	66.20 s
MTP	51.62 s	51.81 s	51.72 s

Impact of multiple prefetchers. Although the number of prefetcher threads is adjustable, tests showed that the default setting of 1 is the best choice. Table 7.7 verifies that having two prefetchers worsen the runtime by 15% due to more overhead.

Table 7.7.: Impact of two prefetching threads

Time	Difference	Terminal command
52.05 s	100.00%	$ pfh -t 8 -c hash -d t5 -h md5
60.09 s	115.44%	$ pfh -t 8 -c hash -d t5 -h md5 -p 2

Impact to a forensic investigation. In the following we analyze the improvement with respect to real world scenarios. Therefore, we took two real life devices and used the results from this section to do a projection. Based on our previous findings, we can estimate the runtime written in Table 7.8.

7.1.4. Distinction to existing parallelization tools

There are a couple of tools which execute commands, scripts or programs in parallel. Thus, we compared our framework against *parallel*[9] and *Parallel Processing Shell Script*[10]

---

[9] `http://www.gnu.org/software/parallel/` (last accessed 2014-03-03).

[10] `http://code.google.com/p/ppss/` (last accessed 2014-03-03).

Table 7.8.: Summary of applying PFH on comprehensive two disks.

	Files	Size	Avg. size	stand-alone	SMT	MTP
MacOS	322,531	100.92 GB	328.08 KB	99 min 51 sec	73 min 43 sec	56 min 43 sec
Win7	139,303	36.55 GB	275.13 KB	36 min 10 sec	26 min 42 sec	20 min 32 sec

(abbreviated *ppss*) as they were ranked highly by Google. Both of the tools work on local cores (not distributed) with multi-threading and distribute the workload automatically to different threads.

The main conclusion from the results in Table 7.9 is that MTP outperforms existing tools/scripts. Both analyzed tools do a simple multi-threading and therefore we expect results similar to SMT. This is true for *parallel*. In case of *ppss* the performance is worse due to a lot of I/O operations. *ppss* saves its state to hard drive in text files.

Table 7.9.: Comparison of with different parallelization tools using `ssdeep`.

	Time	Difference	Terminal command
original	83.67 s	100.00%	$ ssdeep -r t5 > /dev/null
ppss	337.11 s	402.90%	$ ppss -p 8 -d t5 -c 'ssdeep'
parallel	69.81 s	83.43%	$ parallel ssdeep – data/t5/*
SMT	67.55 s	80.73%	$ pfh -t 8 -c hash -d t5 -h ssdeep
MTP	52.04 s	62.20%	$ pfh -t 8 -c hash -d t5 -h ssdeep

We verified our assumption that the slowness of *ppss* is due to the I/O bound by hashing 16 large files each having 1.2 GB. In this case there are only a few writing operations. The results are given in Table 7.10 where the performance increased. However, it is still slower than the original implementation which is due to hard disk reading operations. Each thread reads the file whereas using `ssdeep` as stand-alone reads sequentially.

Table 7.10.: Comparisson of ppss and `ssdeep` with 16×t5.gz

	Time	Difference	Terminal command
ssdeep	264 s	100%	$ ssdeep -r t5.bz
ppss	342 s	129%	$ ppss -p 8 -d t5.gz -c 'ssdeep'

7.2. Reducing data for forensic investigations using approximate matching

This section studies the quantitative results on identification rates of approximate matching for complete disk images. While reference results are available for cryptographic hashing (e.g., [103, 49]), approximate matching has not been evaluated on such a large scale before. Furthermore, we conduct an analysis of error rates for approximate matching and establish practice-oriented thresholds for similarity scores based on this analysis.

7.2.1. Methodology and experimental Setup

All tests are performed on an Ubuntu 12.04 host using `ssdeep` which was installed from the universe repository. A benefit of using a Linux system is the possibility of mounting disk images directly in a read-only mode.

The images are based on Windows and Linux. It is indisputable that Windows is the most commonly used system[11] and hence our study analyzes Windows XP (Professional, 32 bit) and Windows 7 (Professional, 32 bit). Furthermore, we included the results for a Linux Ubuntu (version 12.4, 32 bit) as we had access to a used image.

Testing methodology. In case of whitelisting an investigator is more interested in exact duplicates because small changes might change a file from good to suspicious. Thus, whitelisting should be done by cryptographic hashing. On the other hand approximate matching is quite reasonable for blacklisting, i.e., identifying files that are similar to suspicious files is helpful. However, neither are blacklist databases freely available nor is there illicit data on our test devices. Moreover, we just want to quantify the differences of the detection rates between cryptographic hashing and approximate matching–independently from the purpose of the underlying database.

Hence, we focus on the question: *What are the differences of applying SHA-1 and `ssdeep` for known file identification and is it reasonable to use approximate matching for blacklisting?*. Thus, we study the detection rates in a more general manner and use self-made as well as freely available whitelist databases.

The proceeding is quite simple. We have different databases as well as disk images at hand and analyze the detection rates with respect to true positive and false positives for cryptographic hashing (SHA-1) as well as approximate matching (`ssdeep`). Our assessment focuses on positives only. Note, if no evidence is found, an investigator needs to analyze the device manually and will consider all negative matches.

In some cases we were not able to generate `ssdeep` hashes, e.g., for some (but not all) large files. Since we use the algorithm as a blackbox, we treat these as bugs and did no further investigation. For instance, we could not process the `hyberfil.sys` (Win XP) or device files and named pipes (Ubuntu).

Overall the number of files for which no hash value could be generated is very low. In the worst case, which is Ubuntu$_\mathrm{U}$, we had to drop 0.12% of the files. If no digest could be generated, the file is treated as if it did not exist.

System description and snapshots. For this study we analyzed two of our own live systems (XP, Ubuntu). In addition, we set up a Windows 7 system and created some traces of usage.

---

[11] `http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0` (last accessed 2014-03-04).

In case of the Windows XP system, we possess two snapshots of an extensively used system which were 14 and 27 months 'old' at the time of the snapshots (the age is the time passed between system installation and snapshot generation). The XP system has been used on a daily basis for office work, and it contains software installations, artifacts of system and software updates, user created files, and many kinds of traces of software usage. The Ubuntu system was about half a year old when we took the snapshot. This snapshot also includes system updates as well as developing and web browsing artifacts. Our Windows 7 system was only used for half a day. During that time we applied the latest updates, created some files and visited a few web pages.

In order to have a trustful reference we created default installations of all systems which remain untouched – without any updates, service packs or additional programs. Table 7.11 provides a brief overview of all snapshots. WinXP$_{\mathrm{U1}}$ uses more disk space than WinXP$_{\mathrm{U2}}$ because in the meantime, a 82 GB image file was deleted to free up disk space.

Table 7.11.: Snapshots of the operating systems.

Operating system	File count	Disk usage
WinXP$_{\mathrm{D}}$ (Windows XP, default installation)	8,946	1.9 GB
WinXP$_{\mathrm{U1}}$ (Windows XP SP3, 14 months old)	195,186	128.4 GB
WinXP$_{\mathrm{U2}}$ (Windows XP SP3, 27 months old)	466,266	109.5 GB
Win7$_{\mathrm{D}}$ (Windows 7, default installation)	45,470	8.1 GB
Win7$_{\mathrm{U}}$ (Windows 7 SP1, used installation)	66,312	9.4 GB
Ubuntu$_{\mathrm{D}}$ (v12.04, default installation)	185,468	3.3 GB
Ubuntu$_{\mathrm{U}}$ (v12.04, used installation)	411,209	25.2 GB

Reference database. To determine the identification rates for each image we used the following reference databases:

NIST database. This is our term for the `ssdeep` dataset published by the NIST[12]. The set is provided as a text file showing one SHA-1 hash and one `ssdeep` hash per line. The SHA-1 hash is used to link the entry to the regular RDS from the NIST. The NIST database corresponds to the RDS 2.27 from December 2009 and has over eight million entries. We used both the `ssdeep` hashes and the SHA-1 hashes from this database for our experiments.

Self-created reference databases. As the NIST database is only an outdated subset of the current RDS 2.42 (from September 2013) and no newer `ssdeep` dataset is available, we decided to build our own reference sets based on the default installation from each operating system. This is also the way NIST creates its database – hashing default installations and installation media. This should simulate newer

---

[12] `http://www.nsrl.nist.gov/ssdeep.htm` (last accessed 2014-03-03).

databases and, therefore, the reduction rates of having more up to date database. This was done for both, `ssdeep` and SHA-1 hashes.

Quality of ssdeep matches. The matches reported by `ssdeep` correspond to the raw amount of 'positives'. In order to classify them into true positives and false positives, we need possibilities to reduce them automatically – they are far too many for manual inspection. It is obvious that the critical amount of files is the amount of all `ssdeep` matches minus the SHA-1 matches since we rate SHA-1 matches as true positives without further consideration.

Based on existing literature: In order to distinguish between similar and non-similar files, a possibility is to set a threshold t. Thus, a comparison yielding a match score M with $M \geq t$ indicates similar files (positive match). However, identifying an appropriate t is a challenging as

- a low t increases the *false positive* rate, and
- a high t decrease the *true positives* rate.

Roussev studied the ratio between true and false positives [81, Sect. 4.2.1] based on the comparison of 4,457 files (= 9,930,196 pairs) and published Fig. 7.3. Accordingly, we set the thresholds to $t = 60$ and $t = 40$. While the former threshold has only a few false positives, we rate the false positive rate of the latter as acceptable.

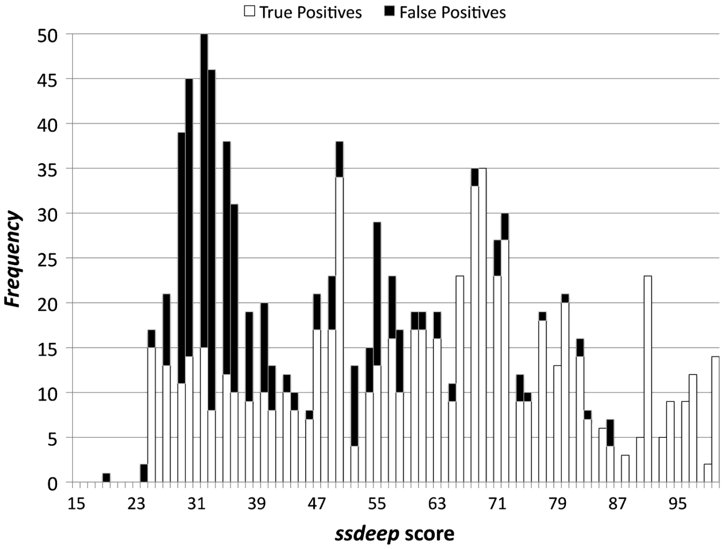


Figure 7.3.: Distribution of `ssdeep` scores according to Roussev [81].

Correlation by file name and path: The second classifier is based on the paths and names of files where we have the following two categories:

1. Identical path names (directory name and file name): Files with the same path within different images are considered as similar files (high indication that there is similarity).
2. Identical file names: An identical file name indicates similarity between two files (medium-high indication that there is similarity).

7.2.2. Assessment and Results

The upcoming subsections present the results of our case study. First, we discuss the difference between SHA-1 matches and ‘perfect’ `ssdeep` matches. Next, we show the identification rates when comparing the images to different databases. The third section presents the identification rates in correlation with file names and paths. The last section analyzes the relation between identification rates and different file types.

Seemingly identical files. As will be shown in the later sections, there are minor differences between a SHA-1 match and a `ssdeep` score of 100. It is a well known that a SHA-1 match implies two identical files with extremely high certainty. In fact, no SHA-1 collision has been found until today. In contrast, a `ssdeep` score of 100 does not necessarily presume two identical files. Additionally, identical `ssdeep` hashes may not get a score of 100.

The reason for the problem is the similarity digest comparison method of `ssdeep`: The algorithm for calculating the similarity requires that two digests have a common substring of length 7 – otherwise the score is directly set to 0[13]. Hence, identical files have a similarity score of 0 if their fingerprints are too short. In particular, comparing two empty files yields a score of 0 where SHA-1 matches them perfectly.

On the other hand, the `ssdeep` score can be 100 although the SHA-1 hashes are different which is due to fingerprint collisions and the comparison procedure. Note, each `ssdeep` digests consists of two fingerprints where it is sufficient to have one matching pair for a score of 100. Thus, there are two types of collisions: In some cases the `ssdeep` hashes match entirely, and in other cases only one pair of `ssdeep` fingerprints matches. We consider the occurrence of a score 100 for non-identical files as false positive (at least for the threshold of 100) because such a score pretends a perfect match.

Identification rates based on a given threshold. Based on the previous findings, we decided to analyze the detection rates based on $t = 40$ and $t = 60$. Thus, this section compares the detection rates using `ssdeep` and SHA-1 for automated file identification.

We used the following abbreviations in the upcoming sections.

- D denotes the analyzed system and $|D|$ denotes the amount of files on the device.
- I_A is the *identification rate* using algorithm $A \in \{\texttt{ssdeep}, \text{SHA-1}\}$. For instance, assuming $I_{\text{ssdeep}} = 10\%$ means that we found 10% of all files from device D within

---

[13] This has already been observed by Baier and Breitinger [4].

the database DB using `ssdeep`. Thus, the higher I is, the more files are identified automatically.

Test-case 1 compares the **default installations against the NIST database**. The results are shown in Table 7.12. The rates for the XP system are significantly higher compared to the other operation systems which is due to the underlying database: it is from December 2009 when XP was a prominent OS. Nevertheless, the trend is obvious, irrespective of the operating system the identification rate is much higher using `ssdeep`, e.g., nearly 10 times better for Win7 and setting $t = 40$.

Table 7.12.: TC1: Comparing default installations against the NIST database.

D	$\|D\|$	I_{ssdeep} $t = 40$	I_{ssdeep} $t = 60$	I_{ssdeep} $t = 100$	$I_{\text{SHA-1}}$
WinXP_{D}	8,946	68.69%	63.85%	35.84%	35.24%
Win7_{D}	45,470	16.59%	9.32%	1.73%	1.70%
Ubuntu_{D}	185,468	16.14%	9.54%	1.90%	1.95%

Test-case 2 compares the **used installations against the NIST database**. The results from Table 7.13 are comparable to the previous test. Again, the XP systems have the best identification rates because of the underlying database. However, the rate is smaller than in the previous case due to the large amount of files on the devices. For instance, the WinXP_{U2} has over 50 times more files as WinXP_{D}. Nevertheless, the identification rates are approximately 2 to 10 times higher using `ssdeep` with $t = 60$ compared to SHA-1.

Table 7.13.: TC2: Comparing used installations against the NIST database.

D	$\|D\|$	I_{ssdeep} $t = 40$	I_{ssdeep} $t = 60$	I_{ssdeep} $t = 100$	$I_{\text{SHA-1}}$
WinXP_{U1}	195,186	17.79%	14.70%	7.69%	8.03%
WinXP_{U2}	466,266	23.02%	17.39%	7.05%	7.30%
Win7_{U}	66,312	17.88%	10.21%	1.45%	1.44%
Ubuntu_{U}	411,209	23.76%	17.11%	1.79%	1.82%

Test-case 3 compares the **used installations against the default installations**. The main results are given in Table 7.14. The low rates for the Windows XP systems are a consequence of the 'small' default installation. Recall that the default installation only includes ≈ 9000 files. However, some files from the default installation are similar or identical to many files in the used installation. Examples are `desktop.ini` files, DLLs, and file shortcuts (`.lnk`). Typical locations for such files in the default installation are `WINDOWS/system32/config/systemprofile`, `WINDOWS/pchealth/helpctr/System`, and `WINDOWS/system32/dllcache`. Hence, it is possible to identify more files in the used system than present in the default system.

The high identification rates for the Windows 7 system reflect the fact that this system has not been used much. But we can clearly see that the installed updates introduced many files similar to files from the default system.

Table 7.14.: TC3: Comparing used against default installations.

D	$\lvert D\rvert$	I_{ssdeep} $t = 40$	I_{ssdeep} $t = 60$	I_{ssdeep} $t = 100$	$I_{\text{SHA-1}}$
WinXP_{U1}	195,186	5.12%	4.71%	4.13%	4.14%
WinXP_{U2}	466,266	2.27%	2.01%	1.72%	1.74%
Win7_{U}	66,312	93.85%	90.19%	67.41%	67.72%
Ubuntu_{U}	411,209	55.67%	53.38%	47.74%	47.83%

Test-case 4 compares **both used Windows XP snapshots against each other** where WinXP_{U1} emulates the database. The main results are given in Table 7.15. Despite a difference of 13 months we obtain identification rates between 24% for SHA-1 and 32% for `ssdeep` which is a difference of 8%. Although 8% sounds small, in the case of 466,266 files these are approximately 37,000.

Table 7.15.: TC4: Using WinXP_{U1} as database and comparing it against WinXP_{U2}.

D	$\lvert D\rvert$	I_{ssdeep} $t = 40$	I_{ssdeep} $t = 60$	I_{ssdeep} $t = 100$	$I_{\text{SHA-1}}$
WinXP_{U2}	466,266	31.98%	28.87%	24.04%	24.37%

Conclusion for TC1 to TC4. The tests confirmed (as expected) that the detection rates for `ssdeep` are higher than for SHA-1 in all cases which is especially relevant for blacklisting (similar files to suspicious ones might be evidence). The utilized thresholds showed in a comprehensive survey from Roussev that they are reasonable and have an acceptable false positive rate.

Identification rates in correlation with file name and path. This section analyzes the reliability of the positive matches for the used thresholds. Recall, the uncertain matches are the amount of total `ssdeep` matches minus exact matches due to SHA-1. An overview of the results is given in Table 7.16. The first column describes the comparison, e.g.,, WinXP U1 against U2. $|D|$ is the number of files on the device. *ssdeep hits* is the amount of matches with a score $\geq t$. The fourth column shows the benefit of `ssdeep`; matches that are not identified by SHA-1 (relative to $|D|$). This is the 'critical amount' of files. The last two columns are relative to *without SHA-1*.

Table 7.16.: Identification rates in correlation with file name and path for different images.

D	$\|D\|$	`ssdeep` hits	without SHA-1	Path matches	Name matches
WinXP$_{\text{(U1 vs. U2)}}$ t=40	466,266	31.98%	7.97%	10.57%	37.09%
WinXP$_{\text{(U1 vs. U2)}}$ t=60	466,266	28.87%	4.86%	15.58%	50.68%
Win7$_{\text{(B vs U)}}$; t=40	66,312	93.85%	26.44%	1.51%	5.79%
Win7$_{\text{(B vs. U)}}$; t=60	66,312	90.19%	22.78%	1.51%	5.91%
Ubuntu$_{\text{(B vs. U)}}$; t=40	411,209	55.67%	7.99%	16.99%	71.37%
Ubuntu$_{\text{(B vs. U)}}$; t=60	411,209	53.38%	5.71%	22.91%	85.89%

For instance, let us consider the last row. That is, we studied the default Ubuntu installation against the used one which contains 411,209 files. If $t = 60$, `ssdeep` detects 53.38% as similar. Reducing this by the amount of SHA-1 matches, there remain 5.71% (i.e., 23,480 files). 22.91% of these files have the same path and 85.89% have the same file name. Hence, we consider them to be true positives.

The remaining 100% − 85.89% = 14.11% are unclear. One the one hand, these might be false positives; on the other hand, they could be moved and renamed files. However, there are still 3313 files in total which needed to be analyzed manually for a final decision.

Identification rates for different file types. While approximate matching on the syntactic level can be applied to any file, it is not equally useful for each file type. It strongly depends on the file type and the supposed type of modifications: whether small modifications preserve most of the binary content of a file or lead to a completely different binary pattern. For example, plain text is favorable while compressed data causes problems. The reason for this is that small text modifications result in small changes in the binary data of a plain text file, but typical compression algorithms create very different byte sequences from similar inputs. Hence, approximate matching on the raw byte level cannot identify the similarity in the latter case.

Here, we try to identify file types for which we gained an extraordinary improvement compared to cryptographic hashing. To achieve this we consider `ssdeep` scores between 60 and 99 for the investigation of file types. Scores of 100 are ignored as they usually belong to identical files.

Table 7.17 shows the ten most frequent file types in the score range 60 to 99 for each analyzed operating system based on test-case 2. The numbers are percentages values compared to all identified within this range. For instance, assuming 10% means that 10% of all detected files with a score between 60 and 99 are of file type X. File types which can safely be identified as text files are marked with an asterisk in the table. Apparently, this is the majority of the listed files. However, there are also file types for binary code where we achieved high detection rates for similar items. The table shows

DLL, and in the other test cases not shown here we also found EXE, PYC, and SO among the top 10 file types. Note that files without name suffix are usually text files or binary executables.

Table 7.17.: File types with high identification rates for non-identical files on basis of test-case 2.

WinXP_{U2}		Win7_{U}		Ubuntu_{U}	
type	amount	type	amount	type	amount
	11.60%	`.mum`*	40.05%	`.html`*	60.62%
`.html`*	10.44%	`.inf`*	10.09%	`.h`*	19.40%
`.h`*	6.40%	`.dll`	8.18%		10.63%
`.yaml`*	6.09%	`.png`	6.06%	`.pm`*	1.42%
`.svn-base`*	4.52%	`.mui`	4.77%	`.gz`	0.73%
`.dll`	4.15%	`.gpd`*	4.42%	`.png`	0.63%
`.png`	3.82%	`.fon`	3.37%	`.py`*	0.63%
`.py`*	2.53%	`.nls`	3.30%	`.al`	0.55%
`.mf`*	1.86%	`.ttf`	2.48%	`.ent`	0.44%
`.htm`*	1.69%	`.ini`*	1.32%	`.ps`	0.36%

7.3. Overview semantic approximate matching

Before discussing advantages and disadvantages of different approximate matching levels, we first give a brief overview of semantic approaches. Recall, semantic approximate matching is bound to a media type and hence we limit this section to images (in fact, a main application of semantic approximate matching in digital forensics is the detection of child pornographic images).

Semantic approximate image matching originates from *content-based image retrieval* (CBIR). This term dates back to 1992 [55], while research in the field has an even longer history. CBIR systems evaluate the similarity of images based on descriptors for color, texture and shape [39]. A standardized set of image features for CBIR applications has been defined in the MPEG-7 standard [66]. However, the calculation of multiple image features is quite time consuming. The `INACT` software, which is based on MPEG-7 descriptors, and which has been developed for supporting forensic investigations, already requires 10 s for processing a medium resolution image (640 × 320 pixels) [47]. This is far too slow for usage in real-life conditions situations.

The analysis of maybe hundreds of thousands images in an investigation target and having up to millions of images in the reference database requires very fast methods for digest calculation and comparison. Hence, we focus on image features with the potential for high efficiency:

Histograms. Color histograms, lightness histograms etc. are very basic image features with a long history [96]. They just count how many pixels correspond to each value of the observed attribute. Robustness and compactness of the information can be increased by extracting features from the histogram like its first three moments [95], Haar wavelet coefficients (MPEG-7), or range selections [106]. However, the extent of images considered as similar in histogram-based matching approaches is more than just different versions of the same image, as the histograms do not consider any spacial information. Thus, such approaches are not well suited for recognizing known images. They are more appropriate for finding images from similar scenes and for clustering images according to the depicted scene.

Low-frequency coefficients. While high-frequency parts of images get easily disturbed or lost due to rescaling or lossy compression, low-frequency parts are quite robust. Accordingly, low-frequency Fourier coefficients, DCT coefficients [40], wavelet coefficients [101], etc. can be used as robust image features. The same idea can be used for deriving a key-dependent robust digest, by replacing the low-frequency basis functions of the aforementioned transformations with "random smooth patterns generated from a secret key" [40]. Typically, images are scaled to a fixed, low resolution before coefficient calculation for reasons of efficiency.

Block bitmaps. Robust features can be obtained by dividing an image into a small, fixed number of blocks and calculating one feature bit per block. The most simple version of this approach scales the image down so that it has one pixel per block and sets the bit according to whether the lightness of the pixel is above or below the median level of light [108]. An improved variant called `rHash` considers the median of each quadrant of the image separately and incorporates a flipping mechanism for robustness against mirroring [93]. Another approach derives an edge map from the image. Such a map can be obtained for example by thresholding the gradient magnitude calculated with the Sobel operator [73, 107]. However, more sophisticated edge detection algorithms should be avoided to keep the computing time low.

Projection-based. This class of approaches has been inspired by the Radon transformation, which calculates angle-dependent, one-dimensional projections of the image by integrating the image along straight lines parallel to each projection direction. The hashing algorithm `RASH` calculates the integral along one radial line for each direction [59]. The proposed improvement `RADISH` replaces the integral by the variance of the luminance of the pixels on the line [92]. Furthermore, the low-frequency DCT coefficients of the previously calculated angle-dependent function can be used as compact, robust digest of an image [36].

Interest points are another kind of image features. Such points are corners and other prominent points in the image, and various kinds of perceptual hashing based on interest points have been proposed [5, 65]. Each interest point can be attached with descriptors in the locality of that point [60, 61]. However, the calculation of interest points is computationally expensive – similar to sophisticated edge detection. Lv and Wang report an average processing time of 3.91 s for an image with their default size of 256 × 342 pixels [61].

For the evaluation we selected four algorithms which are potentially suitable for investigating huge amounts of images: DCT based hash [40], Marr-Hildreth filter based hash [5], radial variance based hash [36] and block mean based hash `rHash` [94]. A similar evaluation of the first three mentioned algorithms and a proof-of-concept implementation of the block bitmap approach based on [108] has been done by Zauner et al. [109, 110]. In contrast to their evaluation on a relatively small number of high-resolution images, we will show results for larger collections of images and different resolutions.

7.4. Benefits and drawbacks of approximate matching and hashing for investigations

Within this assessment we focus on the cryptographic hash function SHA-1 and the bytewise approximate matching algorithms `ssdeep`, `sdhash` and `mrsh-v2` as they are the most promising. Regarding semantic approximate matching, we run DCT based hash (`dct`), Marr-Hildreth operator based hash (`mh`), `radial` variance based hash and block mean value based `rHash`. The pHash C library[14] offers implementations of the first three functions. The implementation of `rHash` is based on the improved block mean value based hash algorithm [94]. Our experiments consist of three test sets of images:

TS_{2000} is a set of 2197 low resolution images (400 × 266 pixels) having 53.3 MB.

TS_{1500} is a set of 1500 medium resolution images (1000 × 800 pixels) having 603 MB.

TS_{1000} is a set of 998 high resolution images (3000 × 2250 pixels) having 719 MB.

7.4.1. Efficiency

Space efficiency. The compression for all algorithms is shown in Table 7.18. In contrast to bytewise approximate matching, the semantic approaches output fixed size digests which is favored.

Table 7.18.: Digest length for different algorithms (compression).

`dct`	`mh`	`radial`	`rHash`	`ssdeep`	`sdhash`	`mrsh-v2`	SHA-1
64 bit	576 bit	320 bit	256 bit	∼ 600 bit	∼ 1.6–2.6%	∼ 1.0%	160 bit

Generation efficiency. The assessment of the runtime efficiency is based on all test sets. The *build*-columns show the times to hash the original test set. For checking, the images in each set are downscaled by 25% and compressed by JPEG 20%.

---

[14] `http://phash.org` (last accessed 2014-03-17).

Table 7.19.: Time for Processing Test Sets (in seconds).

	TS_{2000}		TS_{1500}		TS_{1000}	
Algorithm	build	check	build	check	build	check
`mh`	491.45	5.25	497.17	2.36	1003.59	1.05
`radial`	30.60	91.39	166.67	43.06	710.42	19.11
`dct`	53.50	0.07	287.11	0.06	1534.69	0.025
`rHash`	20.12	0.12	95.23	0.06	419.52	0.033
`ssdeep`	6.21	4.38	41.75	1.91	48.68	1.17
`sdhash`	9.18	2.88	52.28	26.58	52.93	90.18
`mrsh-v2`	3.18	3.01	13.81	28.31	18.66	93.53
SHA-1	0.84	–	2.35	–	2.81	–

As shown in Table 7.19, the cryptographic hash function is the fastest for all sets, followed by bytewise approximate matching. Overall, SHA-1 is roughly one order of magnitude faster than bytewise approximate matching. We neglect the exact check-time for SHA-1 fingerprints (more details see next paragraph).

Regarding bytewise approximate matching only, `mrsh-v2` shows the best generation efficiency. However, `ssdeep` overcomes it when checking because the fingerprints are shorter and faster comparable. In addition, the hash database generated by `ssdeep` features a file size comparison which significantly speeds up the hash checking process when comparing against files of different sizes. The reason why comparing fingerprints is so slow, is the proportional length of the digests.

Among the four semantic algorithms, `rHash` has the best runtime for images of any resolution. The DCT based hash is very fast for low resolution images but becomes the slowest one while hashing high resolution images, where it takes about 4 times longer than `rHash`. Comparing with other perceptual hashes, `mh` is not efficient for low resolution images, but its speed is comparable with radial when coming to large images. Regarding the checking process, `dct` is fastest as it has the shortest binary fingerprint. `radial` is far slower than others, which indicates that peak of cross correlation is not so efficient as hamming distance for hash comparison.

Comparison efficiency. Equal to bytewise approximate matching, semantic approaches digests cannot be order and therefore have a quadratic lookup complexity (it is not possible to sort or index the similarity digests). First experimental results using locally sensitive hashing and binary trees have shown that minor improvements are possible [44]. However, this is ongoing work and final results are unclear.

To conclude, besides cryptographic hash functions only `ssdeep` offers a possibility to reduce the comparison against large databases down to a practical time.

7.4.2. Robustness and discriminability for semantic approaches

Semantic approximate matching has two essential properties: robustness and discriminability. Robustness refers to the ability to resist content-preserving processing and distortions, while discriminability is the ability to differentiate contents, i.e., to avoid collisions [102].

Robustness. Content-preserving processing includes the manipulations that only modify the digital representation of the image content and that apply insignificant perceptual changes on the image content. To evaluate the robustness the following manipulations are applied:

- mirroring: flipped horizontally.
- resizing: 61% downscaling.
- cropping: remove outer 10-15%
- rotation: 90 degree clockwise.
- blurring: Gaussian filter with 20 px radius.
- color modification: red and blue plus 100.
- compression: JPEG with 5% quality.
- stretching: horizontally 20% downscaling.

To test the robustness of semantic approximate matching, ten images are randomly selected out of TS_{1000} and compose a new test set TS_{10}. For easier comparison, the matching scores of all algorithms are represented by a normalized score varying from 0 to 100.

The score of each algorithm after resizing and blurring is always above 95 except `mh` which produces scores between 90 and 95. The color modification yields similar scores but both `mh` and `dct` produce slightly lower scores (90-95) than the others (95-100).

The results of the remaining five manipulations vary enormously and are presented in Fig. 7.4. For rotation and mirroring, the scores of all algorithms are around 50 except that `rHash` performs very well for mirroring. Cropping is a challenging manipulation, where `radial` performs best, followed by `dct`; `mh` and `rHash` are not robust. Both `radial` and `rHash` are very robust to compression, where `dct` and `mh` are inferior. Regarding stretching, all algorithms deliver scores around 98 except radial which is only at 55.

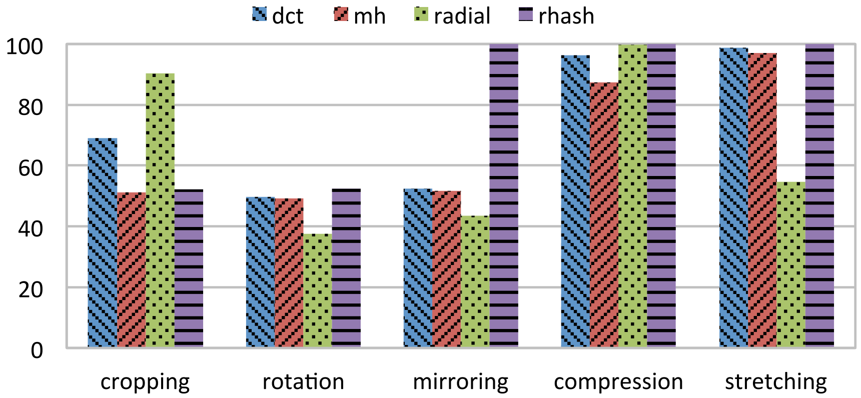
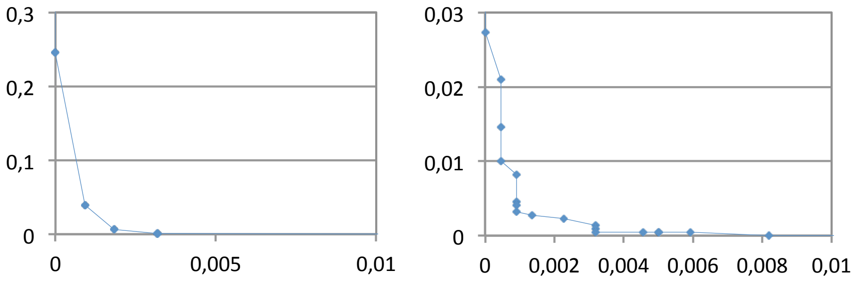


Figure 7.4.: Average scores for five most influencing perceptual changes on TS_{10}.

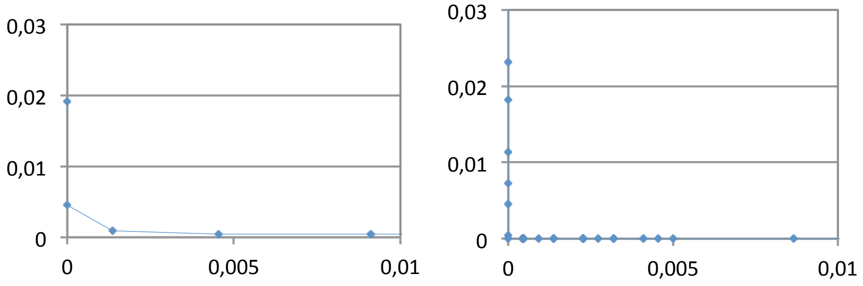
Discriminability. The false positive rate (FPR) and the false negative rate (FNR) are used to measure the discriminability. Therefore, TS_{2000} is the known image set and compared against a new set consisting of 2197 similar images, called TS_{2000}U (the unknown image set). The 4394 images in TS_{2000} and TS_{2000}U contain similar scenes of cheerleaders.

First, each algorithm builds a hash database out of TS_{2000}. Then, all images of the known image set TS_{2000} are downscaled by 25% followed by JPEG compression with a quality factor of 20. Finally, digest matching is performed on the modified TS_{2000} and TS_{2000}U, respectively.

The results are plotted in Fig. 7.5. The x-axis denotes FPR and the y-axis FNR. All algorithms obtain fairly good results except `dct`. Among the four algorithms, only `rHash` achieves zero FPR together with zero FNR. Under the requirement of zero FPR, `dct` has the worst result, whose FNR reaches as high as 0.245, while `mh` and `radial` obtain a FNR of 0.027 and 0.0045. For `dct` algorithm, the best trade-off is to achieve a FPR of 0.0009 with a FNR of 0.0396.



(a) dct (b) mh



(c) radial (d) rhash

Figure 7.5.: FNR/FPR of perceptual hashes on TS2000.

To conclude, all algorithms show good robustness in the case of format conversion, blurring, color modification, resizing, compression and stretching (except `radial`), but are not robust against rotation, cropping (except radial) and mirroring (except `rHash`). Furthermore, under combined manipulation of downscaling and compression, all algorithms (except `dct`) achieve good discriminability between known images and unknown images.

Byte level changes for semantic approximate matching. This paragraph analyzes the behavior of semantic approximate matching for byte level changes which corrupt the files. Therefore, we applied the following byte level modifications:

- broken data: randomly manipulates 10 bytes all in the file body.
- broken/missing header: deletes the first 128 bytes of the image file.
- missing content: deletes 128 bytes in the middle of the image file.
- missing end: deletes the last 128 bytes of the image file.
- inserting data: inserts 128 bytes from a random image file in the middle.

Real life scenarios where these manipulations could happen are: transmitting errors, defect hard disk sectors, RAM analysis or deleted / fragmented files analysis.

'Missing end' does not influence the score and all algorithms output a score of 100. Considering 'missing content' and 'broken data' the scores were still high at round about 90. The lowest scores were returned by 'inserting data' lying between 72 and 82. In all cases the algorithms warn about corrupt JPG data. Regarding 'broken/header' all algorithms failed to produce meaningful results, either by aborting, crashing or delivering out of range errors.

7.4.3. Wrap-up of experimental results

All approximate matching algorithms have a good to very good compression. Most of them produce a fixed length output or have a upper limit expect `sdhash` and `mrsh-v2` which have a proportional length.

As shown in Table 7.19 on page 157 the crypto hash SHA-1 is the fastest algorithm followed by the bytewise approximate matching algorithm. Regarding semantic approaches, there are huge differences in the processing time where `rHash` is by far the fastest.

Considering the comparison efficiency, it is obvious that the lookup complexity of $O(1)$ per hash for crypto hash functions is best followed by `ssdeep`. In Sec. 5.4 on page 69 we presented a lookup concept for bytewise approximate matching algorithm that use Bloom filters, however, this is early stage. The semantic approaches do not have any efficient lookup strategy and hence they are the worst for now.

The robustness is hard to decide. On the one hand we have the semantic algorithms which are very robust against domain specific attacks. However, they fail in fragment detection (e.g., the header is missing) or embedded object detection (e.g., JPG in a Word document) which are the benefits of bytewise approximate matching. In addition, semantic approximate matching is file domain bound and thus each domain needs its own algorithm, e.g., images, movies or music.

7.5. Sample use case: analyzing a USB hard disk drive

In this section we present a reasonable utilization of the three different hash function families on base of the use case *allegation of production and ownership of child abuse pictures*. During a house search the police and IT forensic special agents find besides different computers and DVDs, a USB hard disk drive of size 40 GB (presumably an old backup device).

Following Marcus Roger's process model CFFTPM "time is of the essence" and the search for evidence should start at the crime scene [78]. His key argument is that accused persons tend to be more cooperative within the first hours of an investigation.

During the planning phase of the CFFTPM the forensic investigator chooses hardware (e.g., a forensic workstation equipped with a hardware write blocker for USB devices) and software (implementation of at least one hash function of each family together with respective databases of incriminated pictures) to examine a device onsite for pictures of child abuse. The identification software is configured to run silently in the background and notify the investigator, if potential evidence is found or if the software terminates. An overview of our sample process model for the use case at hand is given in Fig. 7.6.

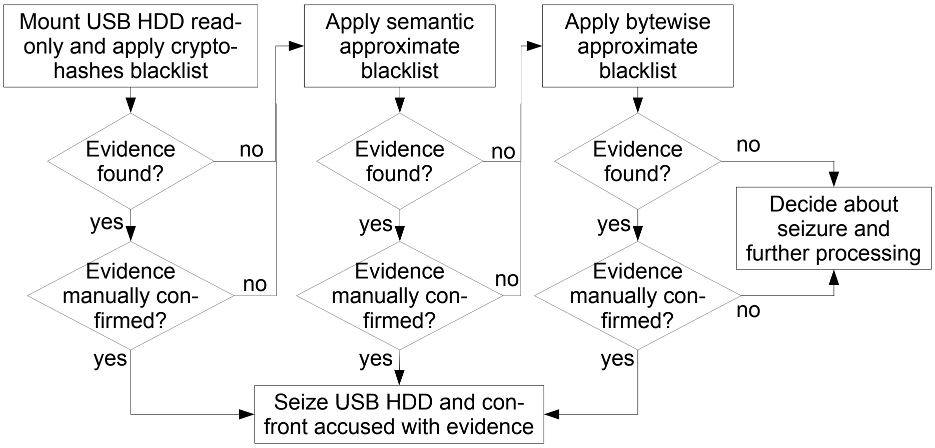

Mount USB HDD read-only and apply crypto-hashes blacklist
Apply semantic approximate blacklist
Apply bytewise approximate blacklist
Evidence found?
Evidence manually confirmed?
yes
no
Decide about seizure and further processing
Seize USB HDD and confront accused with evidence

Figure 7.6.: Process model in case of onsite search for pictures of child abuse.

In our sample use case the investigator decides that an analysis of a 40 GB volume is feasible at scene. He mounts the USB HDD read-only into the file system of his forensic workstation and starts the automatic identification of evidence.

Due to their superior efficiency with regard to generation efficiency, compression and fingerprint comparison, the identification software first applies a blacklist of *crypto hashes* (e.g., PERKEO[15]) to all files on the USB HDD. If there is a match the identification software notifies the investigator, who manually inspects the potential evidence. If it turns out to be a picture of child abuse, he seizes the USB HDD and informs the police to confront the accused person with the evidence.

If the blacklist crypto hashes do not yield a trace or if the alert turns out to be a false positive, the identification software turns to *semantic approximate matching*. We favor semantic approximate matching, because we expect a higher recall in this specific use case. However, this claim has yet to be confirmed. The investigator and the software operate analogously in the case of a true positive and false positive, respectively.

---

[15] `http://perkeo.com` (last accessed 2014-03-01).

Finally, if after the second step no evidence is found, the software performs file carving on the USB HDD and applies *bytewise approximate matching* to all extracted files / fragments. Please note that in contrast to semantic approximate matching, the final bytewise approximate matching may find fragments or embedded pictures of non-image data files (e.g., PDF, DOC). If after all no evidence is found, the investigator decides about a seizure of the device and the further processing, e.g., in the lab.

7.6. Summary

In the previous sections we first explained a method to speed up the overall procedure of automated file identification. Currently hashing a whole file system is very time consuming and done single threaded. We improved this for forensic purposes and provided a reusable framework. The results show an improvement of over 40% for an all-against-all comparison compared to the standard `ssdeep` algorithm. The design of the framework allows to add any other generic algorithm. Additionally, we showed that in a real world scenario the investigation time could be improved a lot, e.g., from 1 h 39 min to 56 min without acquiring any extra hardware.

Secondly, we have presented the very first evaluation of approximate matching on large test cases, i.e., on complete disk images and large reference databases. The results show that of course approximate matching provides a substantial benefit compared to cryptographic hashing. Approximate matching significantly increases the amount of files that can be identified as known files. In particular, all kinds of plain text files as well as files containing binary code can be filtered effectively with approximate matching on the syntactic level. However, we also saw that the reference database must be up to date to achieve the best identification rates.

Thirdly, we provided a rough overview of semantic approximate matching and studied the impact of different hashing technologies for forensic investigations. We discussed all three families of crypto hashes, semantic approximate matching algorithms and bytewise approximate matching functions and highlighted their strengths and weaknesses.

Semantic approximate matching has proven to be most powerful in the area of content identification. Compared to cryptographic hashing or approximate matching, they offer significantly higher detection quality in the areas of image (or other media) copyright violations or illegal material such as child pornography. However, they are bound to their file domain and it is therefore necessary to run perceptual hashing for multiple domains, e.g., images and movies – additional processing time. In addition, these approaches are by default slower than there bytewise opponents.

The key strength of bytewise approximate matching is the ability to detect embedded objects, e.g., detect a JPG within a Word document. In addition, it allows fragment detection which is especially important when dealing with network traffic or defect file systems, e.g., one may analyze the hard disk on the sector or block level.

Cryptographic hash functions are superior to their competitors with respect to efficiency. They are the most recognized in court (so far) and NIST provides a comprehensive database containing approximately 115 million hash values. In addition, they do not err, i.e., their security properties allow to identify equal files with nearly 100% probability, which is very important for whitelisting.

Fourthly, we presented a sample order of applying the hash function families within a sample use case of investigating a USB HDD at crime scene. However, an actual process model to optimize the operation of hash functions and its validation is still missing. A next step could be to identify typical use cases and propose a reasonable order of application of hash functions.

Finally we think that it is also necessary to consider the defendants where we see two possibilities. On the one hand the defendant is the ‘regular user’ and not very familiar with personal computers. Thus, the files reside somewhere unencrypted on the device. Maybe they are processed with a tool to all have the same size. On the other hand the defendant is an ‘expert’ and files might be encrypted. Hence, investigators can try to find fragments in the RAM[16] or in unallocated HDD sectors.

---

[16] Live response.

8

Excursus - Bloom filter in iris recognition

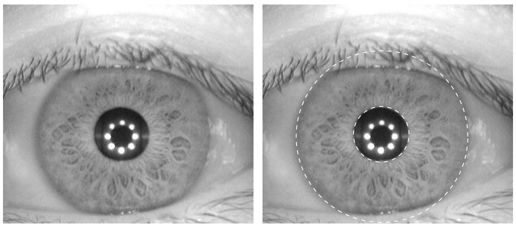
One goal of this dissertation was to demonstrate the feasibility of using techniques from approximate matching in the research area of biometrics which is published in [76, 77]. The main contribution was to identify and develop a technique for using approximate matching for biometric template protection, biometric data compression and efficient biometric identification.

8.1. Motivation

Iris biometric recognition is field-proven as a robust and reliable biometric technology and is used in diverse application scenarios, such as border control, forensic investigations, as well as cryptosystems. Daugman's algorithm [35] forms the basis of the vast majority of today's iris recognition systems, which report (true positive) identification rates above 99% and equal error rates less than 1%: (1) at enrollment an image of a subject's eye is acquired; (2) in the pre-processing step the boundaries of the pupil and the outer iris are detected and the iris (in the approximated form of a ring) is 'un-rolled' to obtain a normalized rectangular iris texture; (3) feature extraction is applied in order to generate a highly discriminative binary feature vector, i.e., iris-code; a typical processing chain of an iris biometric recognition system is depicted in Fig. 8.1. (4) at the time of authentication pairs of iris-codes are efficiently compared by calculating the Hamming distance between them, where template alignment is performed within a single dimension, applying a circular shift of iris-codes, to compensate for head tilts of a certain degree.

Technologies of iris recognition are already deployed on national-sized databases, e.g., the Unique Identification Authority of India, which aims at registering all 1.2 billion Indian citizens, is enrolling 1 million subjects on a daily basis. Resistance to false matches and comparison speed are vital for any large-scale biometric deployments.

From a privacy perspective most concerns about the common use of biometrics arise from the storage and misuse of biometric data as well as the permanent tracking and observation of activities. In accordance with the ISO/IEC IS 24745 [52] on biometric information protection, technologies of biometric template protection which are catego-



(a) Acquisition (b) Detection

(c) Iris texture

(d) Pre-processed iris texture

(e) Iris-code 1-D Log-Gabor filter

(f) Iris-code Ma *et al.*

Figure 8.1.: Preprocessing and applied feature extraction algorithms[1].

rized as biometric cryptosystems and cancelable biometrics, meet the two major requirements: irreversibility and unlinkability (avoid cross-matching). The idea of cancelable biometrics is to apply intentional, repeatable distortions of biometric signals based on transformations that provide a comparison of biometric templates in the transformed domain. However, the majority of approaches to cancelable biometrics report a significant decrease in biometric performance, which is caused by the fact that local neighborhoods of feature elements are often obscured and the transformed enrollment templates are not 'seen' at the time of authentication, i.e., alignment cannot be performed properly.

8.2. Contribution

The contribution was the proposal of a generic approach to obtain a cancelable rotation-invariant representation of iris-codes based on adaptive Bloom filters. Recall, a Bloom filter [6] is a space-efficient probabilistic data structure representing a set in order to support membership queries. In addition, they provide efficient storage and rapid processing of queries. We show how to do Bloom filter-based transformations to tackle all of the following issues regarding (iris) biometrics:

1. *Template protection*: the successive mapping of parts of a binary biometric template

---

[1] Image taken from CASIAv3-Interval iris database: `http://www.idealtest.org`.

to Bloom filters represents an irreversible transformation achieving alignment-free protected biometric templates.

2. *Biometric data compression*: the proposed Bloom filter-based transformation can be parameterized to obtain a desired template size, operating a trade-off between compression and biometric performance.
3. *Efficient identification*: a compact alignment-free representation of iris-codes enables a computaionally efficient biometric identification reducing the overall response time of the system.

According to these benefits, the proposed approach represents a secure template protection scheme which can be efficiently applied within an iris identification system.

8.3. Combining Bloom filters and iris recognition

As described in Sec. 2.5, a Bloom filter bf is a simple bit array of length m, where initially all bits are set to 0. In order to represent a set $S = \{s_0, s_1, ..., s_n\}$ a Bloom filter traditionally utilizes k independent hash functions $h_0, h_1, ..., h_{k-1}$ with range $[0, m-1]$. For each element $s \in S$, bits $h_i(s)$ of Bloom filter bf are set to 1, for $0 \leq i \leq k-1$.

The original concept is adapted in different ways. Given a Bloom filter bf of length m we restrict to inserting exactly l elements, where $l \leq m$. In case of uniformly distributed data the probability that a certain bit is set to 1 during the insertion of an element is $1/m$, i.e., the probability that a bit is still 0 is $1 - 1/m$. For inserting a total of l elements $1 - (1 - 1/m)^l$ bits are expected to be set to 1. For $m = l \cdot c$ and $c \in \mathbb{N}$, i.e., m represents a multiple of l, $\lim_{m\to\infty}(1 - 1/m)^l = 1/\mathrm{e}^{l/m}$. In addition, a trivial transformation Y is applied to each element $x \in S$ instead of multiple hash functions. Since feature elements are expected to be small the application of any hash function would not be resistant to brute force attacks.

In the following subsections the alignment-free adaptive Bloom filter-based system, which is illustrated in Fig. 8.2, and its properties with respect to template protection, biometric data compression and computationally efficient identification are described in detail.

8.3.1. Adaptive Bloom filter-based transformation

Iris-codes typically represent two-dimensional binary feature vectors of width W and height H (see Fig. 8.1 (e)-(f)). In the proposed scheme $W \times H$ iris-codes are divided into K blocks of equal size, where each column consists of $w \leq H$ bits. In case $w < H$ (e.g., for the purpose of compression), columns consist of the w upper most bits, i.e., features originating from outer iris bands, which are expected to contain less discriminitave information, are ignored. Subsequently, the entire sequence of columns of each block is successively transformed to according locations within adaptive Bloom filters, that is, a total number of K separate adaptive Bloom filters of length $n = 2^w$ form the template

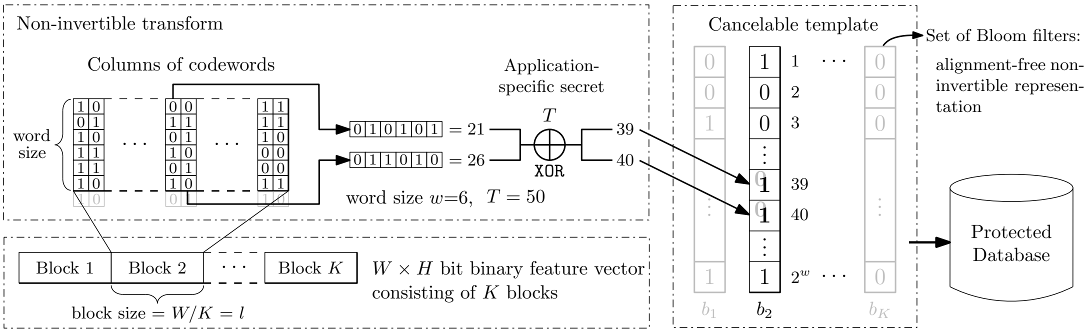


Figure 8.2.: Operation mode of the proposed rotation-invariant biometric templates applying Bloom filter-based transformations to feature vector columns. The highlighted codewords change in Bloom filter b_2 the element at index 39 (decimal representation of 100111) and also index 40 (decimal representation of 101000) to 1.

of size $K \cdot 2^w$. The transformation is implemented by mapping each column within the 2 D iris-code to the index of its decimal value, which is shown for two different codewords (=columns) as part of Fig. 8.2, for each column $x \in \{0,1\}^w$, the mapping is defined as,

$$bf[Y(x)] = 1, \text{ with } Y(x) = \sum_{j=0}^{w-1} x_j \cdot 2^j. \tag{8.1}$$

The very essence of the proposed transformation is that it is alignment-free, i.e., generated templates (= sets of Bloom filters) do not need to be aligned at the time of comparison. Equal columns within certain blocks (= codewords) are mapped to identical indexes within adaptive Bloom filters, i.e., self-propagating errors caused by an inappropriate alignment of iris-codes are eliminated (radial neighborhoods persist). The rotation-compensating property of the proposed system comes at the cost of location information of iris-code columns. At block boundaries miss-alignment of iris-codes will distribute a certain amount of potentially matching codewords among different blocks, which would be mapped to neighbored Bloom filters.

8.3.2. Comparison in transformed domain

The comparison in the transformed domain is basically based on the Jaccard index [53] which is a metric for calculating the similarity/dissimilarity of sample sets. Let $|bf|$ denote the amount of bits within a Bloom filter bf, which are set to 1. Then the dissimilarity DS between two Bloom filters bf_i and bf_j is defined as,

$$DS(bf_i, bf_j) = \frac{HD(bf_i, bf_j)}{|bf_i| + |bf_j|}, \text{ where } |bf_i| \neq 0, |bf_j| \neq 0. \tag{8.2}$$

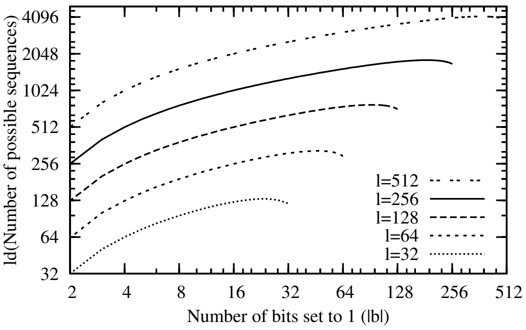


Figure 8.3.: Amount of possible sequences (per block) for different block sizes and proportions of re-mapped codewords.

Obviously, *DS* is computed as efficient as *HD* while *DS* does not have to be computed at numerous shifting positions. In order to incorporate masking bits obtained at the time of pre-processing, columns of iris-codes which are mostly affected by occlusions must not be mapped to adaptive Bloom filters, i.e., a seperate storage of bit masks is not required.

8.3.3. Template protection

The Bloom filter-based transformation conceals the original positions of codewords, i.e., given a Bloom filter bf it is not clear from which column a distinct 1-bit in the generated protected template originated. In addition, it is most likely that diverse columns are mapped to a single index and the occurrence of distinct codewords cannot be established from the stored template, i.e., the proposed transformation achieves irreversible alignment-free templates, implementing cancelable biometrics. In order to provide unlinkability between multiple cancelable templates of a single subject an application specific secret T in form of a bit vector of length w, $T \in \{0,1\}^w$, is incorporated. Each codeword is transformed applying this secret vector (of same length) by XORing prior to mapping it to a Bloom filter. It is important to note that this secret is application-specific (and potentially subject specific) and is only incorporated as parameter in order to suffice the property of unlinkability.

High correlation between codewords, especially neighboring ones, is expected. Consequently, a significant amount of codewords are mapped to identical positions in Bloom filters even for small values of l. Assume $|bf|$ bits are set to 1 within a Bloom filter after inserting l codewords, i.e., $|bf|$ different codewords occur in a block of length l. Hence, the amount of re-mapped bits is $1 - |bf|/l$. For a potential attacker the reconstruction of the original iris-code block involves an arranging of $|bf|$ codewords to l positions (K-times for the entire iris-code). For $|bf| \leq l$ the theoretical amount of possible sequences is recursively defined by the function $f(|bf|, l)$ where each of the $|bf|$ codewords have to appear at least once within l columns,

$$f(|b|,l) = \begin{cases} 1, & \text{if } |b| = 1, \\ |bf|^l - \sum\limits_{i=1}^{|bf|-1} \binom{|bf|}{i} \cdot f(i,l) & \text{otherwise.} \end{cases} \tag{8.3}$$

In other words, all sequences where less than $|bf|$ codewords appear are subtracted from the number of all possible sequences, $|bf|^l$. Fig. 8.3 illustrates the rapid increase of possible sequences even for small values of $|bf|$ (note the logarithmic scales on both axes). Peaks are located around $3l/4$, in case of $l = |bf|$ we get $f(l,l) = l!$ and $f(1,l) = 1$. For instance, for $l = 4$ and $|bf| = 2$ we get $f(2,4) = 2^4 - \binom{2}{1} \cdot f(1,4) = 16 - 2 \cdot 1 = 14$ possible sequences, for $l = 4$ and $|bf| = 3$ we get $f(3,4) = 3^4 - \binom{3}{1} \cdot f(1,4) - \binom{3}{2} \cdot f(2,4) = 81 - 3 \cdot 1 - 3 \cdot 14 = 36$ possible sequences and for $l = 4$ and $|bf| = 4$ we get $f(4,4) = 4! = 24$ possible sequences and so forth. In our paper [77] we demonstrated that for randomly generated bit vectors it is unfeasible for potential attackers to cross-match pairs of protected templates extracted from a single subject and that biometric performance is maintained at a high security level.

8.3.4. Biometric data compression

The original template size is $W \times H$ bits. In the proposed scheme the template is divided into ${}^{W}\!/_{l} = K$ blocks of length l resulting in a template size of $2^w \cdot K = 2^w \cdot {}^{W}\!/_{l}$ where $w \leq H$. If we set $l = 2^q$ a compression is achieved if,

$$W/l \cdot 2^w < W \cdot H \Leftrightarrow 2^{w-q}/H < 1 \tag{8.4}$$

applies, which is most likely the case as we will demonstrate in experiments. For instance, for an iris-code of size 2048 with $W = 256$ and $H = 8$, and the setting $l = 64$ and $w = 8$ we get ${}^{256}\!/_{64} \cdot 2^8 = 1024 < 2048$, i.e., a compression down to 50% of the original size is achieved (${}^{2^{8-6}}\!/_{8} = 0.5$). Sizes of transformed templates are operated by setting parameters l and w. Both, increasing l and decreasing w reduces the overall size of the resulting template, see Eq. 8.4. Our results show that biometric performance is maintained for a compression of down to 20% of original template size.

8.3.5. Adaptive Bloom filter-based identification

Despite indexing techniques, original iris-codes have been combined with compressed and rotation-invariant templates in serial combination scenarios in order to obtain a pre-selection of potential candidate templates. For both types of attempts, compressed templates and alignment-free feature extractors have been found to exhibit unpractical biometric performance, requiring the application of a more sophisticated algorithm within a second stage. In contrast, as it is shown in the paper, the proposed Bloom filter-based transformation generates rotation-invariant cancelable templates which maintain biometric performance.

If a biometric comparator is required to perform $\pm s$ bit shifts in each direction in order to compensate for head tilts the overall amount of bit comparisons increases to $W \cdot H \cdot (2s + 1)$. This means for the proposed approach the number of required bit comparisons is reduced to,

$$\frac{100 \cdot 2^{w-q}}{\big(H \cdot (2s+1)\big)}\% \ . \tag{8.5}$$

For example, if a comparator performs ±6 bit shifts and the proposed transformation retains the template size (no compression) a reduction of bit comparisons down to $1/(12+1) \simeq 7.7\%$ is obtained, while no second algorithm is required. Again, the proposed system takes major advantage of its rotation-compensating property.

In the paper we demonstrated that a reduction of bit-comparisons to less than 5% led to a substantial speed-up of the biometric system in identification mode.

8.4. Summary

The wide use of (iris) biometrics raises the need for privacy protection. Technologies of cancelable biometrics are designed to permanently secure biometric data, preventing from identity fraud and privacy violation. In addition, while a binary representation of biometric features enable a rapid comparison, computational limits are reached deploying national-sized biometric databases in identification mode and public deployments of iris recognition are still based on a brute force exhaustive search through a database. While the majority of approaches to biometric database indexing suffer from a significant decrease in biometric performance, indexing protected biometric templates represents an even greater challenge.

We presented an alignment-free cancelable iris biometric template based on adaptive Bloom filters. The generic adaptive Bloom filter-based transformation which is applied to binary feature vectors of different iris recognition algorithms enables (1) template protection, (2) a compression of biometric data, and (3) computational efficient biometric identification. Existing approaches to iris biometric template protection suffer from low biometric performance or utilize rather insecure alignment-preserving transformations. In contrast, the proposed rotation-invariant Bloom filter-based transformation provides a high level of security while recognition accuracy is maintained. In addition, the presented scheme can be parameterized in order to highly compress biometric templates (down to 10% of original size). Furthermore, since bit-shifting is obsolete at the time of biometric comparison (in transformed domain) a substantial speed-up of biometric identification is achieved. Finally, it is important to note that the proposed approach can be utilized in order to generate a fixed-length protected template based on a variable-length binary biometric feature vector which may be the case for other biometric characteristics, e.g., fingerprints. To the authors' knowledge the proposed approach represents the very first template protection scheme which enables compression and computationally efficient identification.

9

Conclusion & future work

The contribution of this dissertation is the development, assessment and establishment of (new) approaches and concepts of approximate matching. The main research question to answer was: *What is the utility of bytewise approximate matching and how can we establish it in the computer science community?*

To answer this question, we studied approximate matching from scratch and came up with a definition which comprises of common terminology, use cases and requirements. Next, we analyzed both existing approaches, presented improvements and released three new algorithms to extend the range. In order to ease the evaluation process of new algorithms in the future, we implemented the testing framework `FRASH` which is currently in a re-implementation phase. Furthermore, we discussed the application possibilities for approximate matching (byteweise and semantic) for digital forensic investigations. We demonstrated that besides investigations, approximate matching can also be applied to other working fields, such as network traffic analysis or iris template protection.

9.1. Answers to research questions

Based on all our findings, it is possible to answer the research questions given in the introduction:

Research question 1: What is a possible definition for approximate matching? The first step to establish approximate matching in the computer science community was to come up with a common definition and terminology. Basically approximate matching is a generic term describing any technique designed to identify similarities between two digital artifacts. In this context, an *artifact* (or an *object*) is defined as an arbitrary byte sequence, such as a file, which has some meaningful interpretation. Depending on the methods, it may operate at different levels of abstraction: bytewise–, syntactic– or semantic approximate matching.

Besides the actual properties that are required, we extended this description by use cases, e.g., object similarity detection, cross correlation, embedded object detection and fragment detection.

Research question 2: Do the existing implementations have any strengths/weaknesses (by design) and is it possible to improve them? Our review and evaluation of the existing implementations demonstrated that both algorithms have different strengths and weaknesses. For instance, we improved the runtime efficiency of `ssdeep` by 55%, however, an active adversary still can overcome this approach. With respect to `sdhash`, we identified that the algorithm is a lot more robust but had a few bugs that impair the original idea. Nevertheless, it is hard to exploit this approach.

With respect to multi-resolution similarity hashing, we improved the runtime efficiency and published a new implementation called `mrsh-v2`. As a highlight we demonstrated a new comparison function that has two modes: fragment detection and similar file detection.

Research question 3: Can we develop new approaches based on well established proceedings from computer science? In this dissertation we introduced three new algorithms called `bbHash`, `mvhash` and `saHash` which are based on well-known concepts from different areas of computer science.

`bbHash` utilizes ideas from data deduplication and eigenfaces. It tries to rebuild a file as accurately as possible, based on a given set of building blocks. It is the first approach that uses an external structure to create its digests. `mvhash` leans on majority voting in conjunction with run length encoding to compress the input data and uses Bloom filters to represent the similarity digest. As a result of these simple techniques, it is almost as fast as SHA-1. `saHash` uses four independent sub-hash functions which extract statistical peculiarities of a byte sequence. The digests allow to estimate the similarity on the basis of the well-known Levensthein distance, which is a unique feature of that approach.

Research question 4: How can we improve the quadratic lookup complexity of similarity digests? We presented and evaluated a procedure of how to speed up the lookup process for Bloom filter similarity digests. Depending on the use case, we present two different possibilities. The 'file-against-set' comparison allows us to answer the question of whether or not a (similar) file is present in the set by answering yes and no. In contrast, the 'file-against-file' comparison allows to identify all files that are similar to a given similarity digest. While originally both queries had a complexity of $O(z)$ (where z is the amount of digests in the database), we reduced them to $O(1))$ and $O(\log_2(z))$, respectively.

Research question 5: What is a reasonable methodology to test and evaluate bytewise approximate matching? To test and evaluate approximate matching, we presented a framework called FRASH that combines four different test categories. The efficiency-test focuses on the elemental properties of hashing algorithms like ease of computation, compression and fingerprint comparison. Sensitivity & robustness analyzes the behavior of implementations with respect to different challenges like fragment detection or single-common-block correlation. In addition, we included precision & recall on synthetic data as well as real world data. In the latter case, we presented a new procedure called 'approximate longest common subsequence' to create the ground truth. The output is an extensive survey how the algorithms managed different tests.

Research question 6: How can approximate matching improve and support the current forensic investigation process? The experiments demonstrated the benefits of approximate matching compared to cryptographic hash functions. Due to the current processing power (CPU power), there are no remarkable disadvantages when processing huge amounts of data with approximate matching (the bottleneck is the speed of the hard drive). Studying the detection rates exposed that approximate matching does not work perfectly but adds substantial benefits depending on the case. Combining different approaches (i.e., crypto hashes, semantic– and bytewise approximate matching) may provide enormous amounts of support for investigators.

Research question 7: Are there further applications/concepts for approximate matching besides automated file identification in death system analysis? Besides the traditional approach for hashing and approximate matching, we demonstrated the feasibility of being able to apply these algorithms on network traffic. Once the database of 'protected files' is built, the procedure is quite simple: hash the network packet and compare it against the database. Although a false alert rate around 10^{-5} is too high for practical use, there are ways to improve this which will be future work.

Research question 8: Is it possible to adapt concepts from byte approximate matching to improve existing biometric template protection schemes? We demonstrated the feasibility of using techniques from approximate matching in the research area of biometrics by presenting an alignment-free cancelable iris biometric template based on adaptive Bloom filters. The procedure is applied to binary feature vectors of different iris recognition algorithms and enables (1) template protection, (2) a compression of biometric data, and (3) computational efficient biometric identification. In contrast to existing approaches, the proposed rotation-invariant Bloom filter-based transformation provides a high level of security while recognition accuracy is maintained.

9.2. Future work

One limitation are the detection rates of the algorithms as evaluated in Sec. 6.3.4 – all algorithms have a high false negative rate which needs to be improved. In addition, we have to consider if the longest common subsequence is a reasonable measure for the ground truth. Therefore, we have to work on the algorithm to increase their precision. Besides that, we would like to study the possibilities of machine learning for approximate matching and clustering files.

Our test framework FRASH currently consists of multiple scripts implemented in Ruby, C++ and shell scripts. Hence, we are currently working on a C++-implementation that includes all tests, has a comprehensive manual and will be published on github.

We demonstrated that approximate matching can be used for network traffic analysis, where we identified sequences that overcome our current filtering methods which are the basis for good detection rates. Hence, we should aim at improving the filtering techniques. One idea is to provide a list of sequences which are very common and should be ignored. In order to create such a list, one may study each file format intensively or use counting Bloom filters which is the focus at the moment. Another possibility would be to extend `mrsh-net` by learning phase based on a file set so that common sequences are ignored. A second idea is the consideration of more than one packet. This requires a connection table containing the information that user A has a connection to user B. Then, we easily can count the amount of database hits and react, e.g., the connection is interrupted if three hits appear.

Besides the usage of approximate matching on network traffic, we would like to apply this technology to further working fields where especially malware and RAM analysis would be interesting starting points. According to an antivirus vendor, the reason of ignoring approximate matching for their needs was the slow database lookup. Now this challenge is solved and approximate matching algorithms operate with practical speed and may become relevant for them.

List of Appendices

A. Evaluation of `sdhash` implementation

This part is a detailed analysis of the `sdhash` version 1.2 code. In a first step this section compares the specification with the implementation – both have to coincide. As we treat the specification as ‘correct’, we show some discrepancies. The impact of a deviating implementation is an unexpected behavior of the `sdhash` program. Furthermore we discuss three bugs which are mainly shortcomings by design during the comparison of Bloom filters.

A.1. Popularity Rank Computation Bug

While inspecting the code from `sdhash`, we discovered two important bugs when computing the popularity rank R_{pop}:

1. A bug concerning the window size used to compute R_{pop} avoids the correct identification of the minimal R_{prec} value in the current window. We call this bug the *window size bug* in what follows.

2. An inconsistency between the description of the algorithm in [80] and its implementation yields unexpected results for R_{pop}. The inconsistency is related to the property of the algorithm in [80] to consider the leftmost minimal value of R_{prec} in a window. We therefore denote this bug as *leftmost bug*.

These two bugs lead to false results of R_{pop} and thus to an unexpected behaviour of the whole algorithm. The discussion in this section is with respect to the code of the file `sdbf_score.c` of `sdhash` version 1.2 shown in Listing A.1. However, the same bugs are found in the current version 1.3, where the file is now called `sdbf_core.c` (we are not aware of any reason of the renaming).

Before explaining the bugs in `sdbf_score.c` we first give an example of unexpected values for R_{pop}. In Sec. 3.3 we explained the computation of R_{pop} as presented in [80]. We point to a key property of the popularity rank: Let $R_{pop}(i)$ denote the popularity rank at position i. If $R_{pop}(i) = k$ $(1 \leq k \leq 64)$, then $R_{pop}(i + n) \leq n$ for all $1 \leq n < k$.

However, the following listing shows some unexpected values of R_{pop}. The listing is similar to Fig. 3.3 and it is an abridgment of the R_{prec} and its corresponding R_{pop} for file `000110.jpg`.

```
Pos:      76  77  78  79  80  81  82  83
R_prec: 432 353 333 333 325 396 432 472
R_pop:    0   0   1  63   2   1   1   1
[removed]
Pos:     603 604 605 606 607 608 609 610
R_prec: 372 364 335 335 391 416 443 445
R_pop:    1   1  25  40   1   1   1   1
```

The first line shows the position i of R_{prec} in `000110.jpg`, i.e., the offset of the first byte of the underlying feature of R_{prec}.

To recall [80, p.212] says that after a window Win has slided, the R_{pop} of *leftmost lowest* R_{prec} in Win is increased, which is in contrast to the above listing. For instance, for $i = 78$ the 1-neighborhood (positions 77 and 79) should have at most a R_{pop} of 1. In fact we would expect an output like:

```
Pos:      76  77  78  79  80  81  82  83
R_prec: 432 353 333 333 325 396 432 472
R_pop:     0   0   2   0  63   1   1   1
[removed]
Pos:     603 604 605 606 607 608 609 610
R_prec: 372 364 335 335 391 416 443 445
R_pop:    1   1  64  1   1   1   1   1
```

As an explanation we imagine a window Win of size 64 ending at position 78. In order for $R_{pop} = 1$ to hold at position 78, Win must contain 63 values $R_{prec} > 333$ to the left of position 78. We then slide Win by one to position 79. The new incoming R_{prec} is also 333 and thus the R_{pop} at position 78 should increase again as it is still the leftmost lowest R_{prec} in Win. Sliding one step further there is even a lower $R_{prec} = 325$, wherefore R_{pop} at position 80 should be increased.

We now explain the location of the window size bug using Listing A.1 and show that the false value $R_{pop} = 63$ at position 79 is due to this bug. The window size bug means that two different window sizes W are used in the code to compute R_{pop}. In line 92 of the listing the current window Win starts at position i and uses a window size W. However, the positions i until $i + W$ comprise $W + 1$ positions and therefore a -1 needs to be added. On the other side, at line 101 of Listing A.1 the correct window size W is used to find the minimal value in Win[1].

---

[1] Actually also W is used, but the condition uses a $<$ instead of $\leq$.

```
86  UINT min_pos = 0;
87  UINT min_rank = ranks[min_pos];
88  for( i=0; i<sdbf_sys.file_size-sdbf_sys.pop_win_size; i++) {
89      // try sliding on the cheap
90      if( i>0 && min_rank) {
91          UINT ix=0;
92          while( ranks[i+sdbf_sys.pop_win_size] >= min_rank && i<min_pos && i<sdbf_sys.
               file_size-sdbf_sys.pop_win_size+1) {
93              if( ranks[i+sdbf_sys.pop_win_size] == min_rank)
94                  min_pos = i+sdbf_sys.pop_win_size;
95              pop_scores[min_pos]++;
96              i++;
97          }
98      }
99      min_pos = i;
100     min_rank = ranks[min_pos];
101     for( j=i+1; j<i+sdbf_sys.pop_win_size; j++) {
102         if( ranks[j] < min_rank && ranks[j]) {
103             min_rank = ranks[j];
104             min_pos = j;
105         } else if( min_pos == j-1 && ranks[j] == min_rank) {
106             min_pos = j;
107         }
108     }
109     if( ranks[min_pos] > 0) {
110         pop_scores[min_pos]++;
111     }
112 }
113 free( ranks);
114 fclose( in);
115 return pop_scores;
116 }
```

Listing A.1: Abridgment of `sdbf_score.c` from `sdhash 1.2`.

We shortly discuss the implications of this bug. Due to the window size $W+1$ in line 92 the `while`-loop is always one R_{prec} ahead. Thus when the `while`-loop in line 92 processes position 80, it reads in $R_{pop} = 325$ where the first condition does not hold and we jump to line 99. However, the implementation now checks in line 101 for a window of size W which ends at position 79 and therefore computes a minimal value $R_{pop} = 333$.

Additionally the leftmost bug in lines 93 and 94 chooses for the rightmost position if the righmost R_{pop} is equal to the previous minimal value (a similar bug is implemented in lines 105 and 106).

To resolve these two bugs we propose the following changes to the source code:

1. Replace the first condition in line 92 by

   ```
   ranks[i+sdbf_sys.pop_win_size-1]
                            >= min_rank
   ```

   to resize the window to its correct length of 64.

2. Remove lines 93, 94, 105 – 107 to correct the leftmost bug.

A.2. Design errors within the comparison class

The comparison function of a similarity digest algorithm is crucial for detecting related files. However, when reviewing the code of the matching part of `sdhash`, we discovered some shortcomings by design:

- In case that one of the files has at most 63 features, the comparison functions yields two different results depending on the order of the two files. If the file containing at most 63 features is invoked as the second argument it results in the similarity score -1, a not comparable. We denote this bug by *not-comparable bug.*
- A self-comparison of a file may result in a similarity score significantly smaller than 100 (a lower bound is 50). We denote this bug by *self-comparison bug.*
- It is possible that two different files yield a 100 match by design (and not at random). We denote this bug by *collision bug.*

Responsible for the first two bugs is the misplaced if-condition `if(s2 < 64)` in the file `sdbf_score.c` in line 198 where `s2` denotes the amount of features within a Bloom filter. Roughly speaking this if-condition skips the comparison of partially filled Bloom filters, if the threshold of 63 features is not exceeded.

Not-comparable bug: This bug can arise while comparing two files where at least one has at most 63 features and thus results in only one Bloom filter. Listing A.2 shows the comparison of a 48-feature-file against a 84-feature-file and vice-versa.

As a result we obtain a match score of 100 or a -1 (not-comparable) depending on the ordering. Both results are not reasonable. The supposed perfect match is due to the collision bug as explained below. The not-comparable result is due to the aforementioned if-condition: The comparison is skipped if the second input file does not have enough (≥ 64) features.

```
$ sdhash -g 48.ftrs 84.ftrs
48.ftrs 84.ftrs 100

$ sdhash -g 84.ftrs 48.ftrs
84.ftrs 48.ftrs -01
```

Listing A.2: Wrong order comparison yields a not comparable.

A possible solution is given at the end of ‘self-comparison bug’.

Self-comparison bug: Let $SD = \{bf_1, bf_2\}$ be the similarity digest of the input for the self-comparison, where $\overline{bf_1} = 128$ and $\overline{bf_2} < 64$ (i.e., the first Bloom filter contains the maximum amount of 128 features and the second one is below the self-comparison threshold of 64 features as defined by the `if`-condition from above).

To receive the similarity score there is an all-against-all comparison of Bloom filters as defined by Eq. 3.10 of the two similarity digests. In our sample case, SD is compared against itself. Thus $SF_{score}(bf_1, bf_1)$ results in a 100 match and is therefore the best match. But due to the `if`-condition it is not allowed to compute $SF_{score}(bf_2, bf_2)$ as the if-condition requires at least 64 features for the second Bloom filter bf_2. Therefore $SF_{score}(bf_2, bf_1)$ is used. However, we have $0 \leq SF_{score}(bf_2, bf_1) \leq 100$, which yields a lower bound of 50 for the self-comparison similarity score.

```
1  $ sdhash t5/000256.doc
2  t5/000256.doc sdbf:sha1:256:5:7ff:128:2:18: IiYTAaVQMQRUUlEFbLOBDKUQAIkEOJEDMEEAgPIoiAi
       gDQNYAdsADCEdEGBAC9wBYiEA4AFARGYJdMASZEJABS [removed]
       AAAAAQECAAAAAAAAAAEEAgAQAACIAAAAEBAAA=
3
4  $ 1.2/sdhash -g t5/000256.doc t5/000256.doc
5  t5/000256.doc t5/000256.doc 055
```

Listing A.3: A file compared to itself with a match score of only 55.

Let us look at an example from the t5-corpus given in Listing A.3. First, we print the similarity digest for `00256.doc`. Row 2 shows some basics about the digest itself, e.g., `sdhash` uses SHA-1, has Bloom filters of 256 bytes each with 128 features, overall there are 2 Bloom filters and the last one contains 18 features. Due to the second Bloom filter with only 18 features the self-comparison in rows 4 and 5 yield an overall similarity score of 55. Eq. 3.10 yields the similarity score $SF_{score}(bf_2, bf_1)$:

$$\frac{100 + SF_{score}(bf_2, bf_1)}{2} = 55 \ ,$$

hence $SF_{score}(bf_2, bf_1) = 10$.

To generalize, if the similarity score of a file is built up of r Bloom filters and the last one contains less than 64 features, it is not trustful. If such a file is compared to itself a lower bound of its similarity score is therefore $\frac{(r-1)\cdot 100+0}{r} = 100 - \frac{100}{r}$.

To avoid the bugs, all thresholds need to be adjusted where we recommend an upper bound of 6. In order not to reject the last Bloom filter if it contains less than 6 features, there should be an extension where we allow more than 128 elements within the last filter. Thus in case the last Bloom filter would be skipped due to too less features we merge the last two filters. As a consequence the last Bloom filter could have at most 133 features.

B. All results of the precision & recall test for synthetic data

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mrsh	Alignment	1k	25	236	99000	0	764	0,24	1,00	0,00	0,76	1,00
mrsh	Alignment	1k	50	237	99000	0	763	0,24	1,00	0,00	0,76	1,00
mrsh	Alignment	1k	100	237	99000	0	763	0,24	1,00	0,00	0,76	1,00
mrsh	Alignment	1k	200	237	99000	0	763	0,24	1,00	0,00	0,76	1,00
mrsh	FragmentEnd	1k	50	4	99000	0	996	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	1k	60	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	1k	70	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	1k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	50	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	60	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	70	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	Noise	1k	0,5	199	99000	0	801	0,20	1,00	0,00	0,80	1,00
mrsh	Noise	1k	1	117	99000	0	883	0,12	1,00	0,00	0,88	1,00
mrsh	Noise	1k	1,5	52	99000	0	948	0,05	1,00	0,00	0,95	1,00
mrsh	Noise	1k	2	19	99000	0	981	0,02	1,00	0,00	0,98	1,00
mrsh	Noise	1k	2,5	6	99000	0	994	0,01	1,00	0,00	0,99	1,00
mrsh	SingleCommor	1k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	1k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	1k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	1k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	1k	20	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	1k	30	11	99000	0	989	0,01	1,00	0,00	0,99	1,00
mrsh	SingleCommor	1k	40	39	99000	0	961	0,04	1,00	0,00	0,96	1,00
mrsh	SingleCommor	1k	50	86	99000	0	914	0,09	1,00	0,00	0,91	1,00
mvHash	Alignment	1k	25	996	99000	0	4	1,00	1,00	0,00	0,00	1,00
mvHash	Alignment	1k	50	946	99000	0	54	0,95	1,00	0,00	0,05	1,00
mvHash	Alignment	1k	100	547	99000	0	453	0,55	1,00	0,00	0,45	1,00
mvHash	Alignment	1k	200	14	99000	0	986	0,01	1,00	0,00	0,99	1,00
mvHash	FragmentEnd	1k	50	374	99000	0	626	0,37	1,00	0,00	0,63	1,00
mvHash	FragmentEnd	1k	60	103	99000	0	897	0,10	1,00	0,00	0,90	1,00
mvHash	FragmentEnd	1k	70	16	99000	0	984	0,02	1,00	0,00	0,98	1,00
mvHash	FragmentEnd	1k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mvHash	FragmentRand	1k	50	340	99000	0	660	0,34	1,00	0,00	0,66	1,00
mvHash	FragmentRand	1k	60	84	99000	0	916	0,08	1,00	0,00	0,92	1,00
mvHash	FragmentRand	1k	70	10	99000	0	990	0,01	1,00	0,00	0,99	1,00
mvHash	FragmentRand	1k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	1k	0,5	99	99000	0	901	0,10	1,00	0,00	0,90	1,00
mvHash	Noise	1k	1	9	99000	0	991	0,01	1,00	0,00	0,99	1,00
mvHash	Noise	1k	1,5	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	1k	2	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	1k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	20	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	30	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	40	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	1k	50	24	99000	0	976	0,02	1,00	0,00	0,98	1,00
sdhash	Alignment	1k	25	860	99000	0	140	0,86	1,00	0,00	0,14	1,00
sdhash	Alignment	1k	50	865	99000	0	135	0,87	1,00	0,00	0,14	1,00
sdhash	Alignment	1k	100	890	99000	0	110	0,89	1,00	0,00	0,11	1,00
sdhash	Alignment	1k	200	897	99000	0	103	0,90	1,00	0,00	0,10	1,00
sdhash	FragmentEnd	1k	50	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	1k	60	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	1k	70	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	1k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	50	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	60	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	70	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	Noise	1k	0,5	639	99000	0	361	0,64	1,00	0,00	0,36	1,00
sdhash	Noise	1k	1	600	99000	0	400	0,60	1,00	0,00	0,40	1,00
sdhash	Noise	1k	1,5	396	99000	0	604	0,40	1,00	0,00	0,60	1,00
sdhash	Noise	1k	2	172	99000	0	828	0,17	1,00	0,00	0,83	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
sdhash	Noise	1k	2,5	52	99000	0	948	0,05	1,00	0,00	0,95	1,00
sdhash	SingleCommor	1k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	1k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	1k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	1k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	1k	20	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	1k	30	64	99000	0	936	0,06	1,00	0,00	0,94	1,00
sdhash	SingleCommor	1k	40	425	99000	0	575	0,43	1,00	0,00	0,58	1,00
sdhash	SingleCommor	1k	50	609	99000	0	391	0,61	1,00	0,00	0,39	1,00
ssdeep	Alignment	1k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	1k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	1k	100	964	99000	0	36	0,96	1,00	0,00	0,04	1,00
ssdeep	Alignment	1k	200	922	99000	0	78	0,92	1,00	0,00	0,08	1,00
ssdeep	FragmentEnd	1k	50	971	99000	0	29	0,97	1,00	0,00	0,03	1,00
ssdeep	FragmentEnd	1k	60	678	99000	0	322	0,68	1,00	0,00	0,32	1,00
ssdeep	FragmentEnd	1k	70	30	99000	0	970	0,03	1,00	0,00	0,97	1,00
ssdeep	FragmentEnd	1k	80	17	99000	0	983	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentEnd	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	1k	50	975	99000	0	25	0,98	1,00	0,00	0,03	1,00
ssdeep	FragmentRand	1k	60	679	99000	0	321	0,68	1,00	0,00	0,32	1,00
ssdeep	FragmentRand	1k	70	35	99000	0	965	0,04	1,00	0,00	0,97	1,00
ssdeep	FragmentRand	1k	80	18	99000	0	982	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentRand	1k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	1k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	1k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	1k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	1k	0,5	998	99000	0	2	1,00	1,00	0,00	0,00	1,00
ssdeep	Noise	1k	1	944	99000	0	56	0,94	1,00	0,00	0,06	1,00
ssdeep	Noise	1k	1,5	767	99000	0	233	0,77	1,00	0,00	0,23	1,00
ssdeep	Noise	1k	2	564	99000	0	436	0,56	1,00	0,00	0,44	1,00
ssdeep	Noise	1k	2,5	388	99000	0	612	0,39	1,00	0,00	0,61	1,00
ssdeep	SingleCommor	1k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	1k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	1k	5	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	1k	10	55	99000	0	945	0,06	1,00	0,00	0,95	1,00
ssdeep	SingleCommor	1k	20	587	99000	0	413	0,59	1,00	0,00	0,41	1,00
ssdeep	SingleCommor	1k	30	951	99000	0	49	0,95	1,00	0,00	0,05	1,00
ssdeep	SingleCommor	1k	40	995	99000	0	5	1,00	1,00	0,00	0,01	1,00
ssdeep	SingleCommor	1k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	4k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	4k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	4k	100	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mrsh	Alignment	4k	200	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	4k	50	926	99000	0	74	0,93	1,00	0,00	0,07	1,00
mrsh	FragmentEnd	4k	60	763	99000	0	237	0,76	1,00	0,00	0,24	1,00
mrsh	FragmentEnd	4k	70	415	99000	0	585	0,42	1,00	0,00	0,59	1,00
mrsh	FragmentEnd	4k	80	71	99000	0	929	0,07	1,00	0,00	0,93	1,00
mrsh	FragmentEnd	4k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	4k	50	942	99000	0	58	0,94	1,00	0,00	0,06	1,00
mrsh	FragmentRand	4k	60	763	99000	0	237	0,76	1,00	0,00	0,24	1,00
mrsh	FragmentRand	4k	70	437	99000	0	563	0,44	1,00	0,00	0,56	1,00
mrsh	FragmentRand	4k	80	89	99000	0	911	0,09	1,00	0,00	0,91	1,00
mrsh	FragmentRand	4k	90	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	Noise	4k	0,5	921	99000	0	79	0,92	1,00	0,00	0,08	1,00
mrsh	Noise	4k	1	269	99000	0	731	0,27	1,00	0,00	0,73	1,00
mrsh	Noise	4k	1,5	33	99000	0	967	0,03	1,00	0,00	0,97	1,00
mrsh	Noise	4k	2	7	99000	0	993	0,01	1,00	0,00	0,99	1,00
mrsh	Noise	4k	2,5	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	4k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	4k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	4k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	4k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	4k	20	22	99000	0	978	0,02	1,00	0,00	0,98	1,00
mrsh	SingleCommor	4k	30	283	99000	0	717	0,28	1,00	0,00	0,72	1,00
mrsh	SingleCommor	4k	40	719	99000	0	281	0,72	1,00	0,00	0,28	1,00
mrsh	SingleCommor	4k	50	954	99000	0	46	0,95	1,00	0,00	0,05	1,00
mvHash	Alignment	4k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	Alignment	4k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	Alignment	4k	100	958	99000	0	42	0,96	1,00	0,00	0,04	1,00
mvHash	Alignment	4k	200	11	99000	0	989	0,01	1,00	0,00	0,99	1,00
mvHash	FragmentEnd	4k	50	810	99000	0	190	0,81	1,00	0,00	0,19	1,00
mvHash	FragmentEnd	4k	60	210	99000	0	790	0,21	1,00	0,00	0,79	1,00
mvHash	FragmentEnd	4k	70	4	99000	0	996	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	4k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	4k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	4k	50	790	99000	0	210	0,79	1,00	0,00	0,21	1,00
mvHash	FragmentRand	4k	60	172	99000	0	828	0,17	1,00	0,00	0,83	1,00
mvHash	FragmentRand	4k	70	2	99000	0	998	0,00	1,00	0,00	1,00	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mvHash	FragmentRand	4k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	4k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	4k	0,5	64	99000	0	936	0,06	1,00	0,00	0,94	1,00
mvHash	Noise	4k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	4k	1,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	4k	2	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	4k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	20	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	30	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	40	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	4k	50	36	99000	0	964	0,04	1,00	0,00	0,96	1,00
sdhash	Alignment	4k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	4k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	4k	100	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	4k	200	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	4k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	4k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	4k	70	990	99000	0	10	0,99	1,00	0,00	0,01	1,00
sdhash	FragmentEnd	4k	80	41	99000	0	959	0,04	1,00	0,00	0,96	1,00
sdhash	FragmentEnd	4k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	4k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	4k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	4k	70	994	99000	0	6	0,99	1,00	0,00	0,01	1,00
sdhash	FragmentRand	4k	80	40	99000	0	960	0,04	1,00	0,00	0,96	1,00
sdhash	FragmentRand	4k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	Noise	4k	0,5	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	4k	1	998	99000	0	2	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	4k	1,5	720	99000	0	280	0,72	1,00	0,00	0,28	1,00
sdhash	Noise	4k	2	94	99000	0	906	0,09	1,00	0,00	0,91	1,00
sdhash	Noise	4k	2,5	4	99000	0	996	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	4k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	4k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
sdhash	SingleCommor	4k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	4k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	4k	20	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	4k	30	134	99000	0	866	0,13	1,00	0,00	0,87	1,00
sdhash	SingleCommor	4k	40	982	99000	0	18	0,98	1,00	0,00	0,02	1,00
sdhash	SingleCommor	4k	50	996	99000	0	4	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	4k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	4k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	4k	100	965	99000	0	35	0,97	1,00	0,00	0,04	1,00
ssdeep	Alignment	4k	200	928	99000	0	72	0,93	1,00	0,00	0,07	1,00
ssdeep	FragmentEnd	4k	50	962	99000	0	38	0,96	1,00	0,00	0,04	1,00
ssdeep	FragmentEnd	4k	60	700	99000	0	300	0,70	1,00	0,00	0,30	1,00
ssdeep	FragmentEnd	4k	70	38	99000	0	962	0,04	1,00	0,00	0,96	1,00
ssdeep	FragmentEnd	4k	80	20	99000	0	980	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentEnd	4k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	4k	50	978	99000	0	22	0,98	1,00	0,00	0,02	1,00
ssdeep	FragmentRand	4k	60	691	99000	0	309	0,69	1,00	0,00	0,31	1,00
ssdeep	FragmentRand	4k	70	41	99000	0	959	0,04	1,00	0,00	0,96	1,00
ssdeep	FragmentRand	4k	80	19	99000	0	981	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentRand	4k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	4k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	4k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	4k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	4k	0,5	601	99000	0	399	0,60	1,00	0,00	0,40	1,00
ssdeep	Noise	4k	1	154	99000	0	846	0,15	1,00	0,00	0,85	1,00
ssdeep	Noise	4k	1,5	34	99000	0	966	0,03	1,00	0,00	0,97	1,00
ssdeep	Noise	4k	2	13	99000	0	987	0,01	1,00	0,00	0,99	1,00
ssdeep	Noise	4k	2,5	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	4k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	4k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	4k	5	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	4k	10	65	99000	0	935	0,07	1,00	0,00	0,94	1,00
ssdeep	SingleCommor	4k	20	632	99000	0	368	0,63	1,00	0,00	0,37	1,00
ssdeep	SingleCommor	4k	30	939	99000	0	61	0,94	1,00	0,00	0,06	1,00
ssdeep	SingleCommor	4k	40	996	99000	0	4	1,00	1,00	0,00	0,00	1,00
ssdeep	SingleCommor	4k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	16k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	16k	100	1000	98998	2	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	16k	200	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	16k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mrsh	FragmentEnd	16k	70	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	16k	80	1000	98997	3	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	16k	90	778	98993	7	222	0,78	1,00	0,00	0,22	0,99
mrsh	FragmentEnd	16k	95	85	98998	2	915	0,09	1,00	0,00	0,92	0,98
mrsh	FragmentEnd	16k	97	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentEnd	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	16k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	16k	70	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	16k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	16k	90	765	98989	11	235	0,77	1,00	0,00	0,24	0,99
mrsh	FragmentRand	16k	95	80	98997	3	920	0,08	1,00	0,00	0,92	0,96
mrsh	FragmentRand	16k	97	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mrsh	FragmentRand	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	Noise	16k	0,5	989	99000	0	11	0,99	1,00	0,00	0,01	1,00
mrsh	Noise	16k	1	37	99000	0	963	0,04	1,00	0,00	0,96	1,00
mrsh	Noise	16k	1,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	Noise	16k	2	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	Noise	16k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	16k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	16k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	16k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	16k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	16k	20	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mrsh	SingleCommor	16k	30	252	99000	0	748	0,25	1,00	0,00	0,75	1,00
mrsh	SingleCommor	16k	40	949	99000	0	51	0,95	1,00	0,00	0,05	1,00
mrsh	SingleCommor	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	Alignment	16k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	Alignment	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	Alignment	16k	100	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	Alignment	16k	200	725	99000	0	275	0,73	1,00	0,00	0,28	1,00
mvHash	FragmentEnd	16k	50	999	99000	0	1	1,00	1,00	0,00	0,00	1,00
mvHash	FragmentEnd	16k	60	585	99000	0	415	0,59	1,00	0,00	0,42	1,00
mvHash	FragmentEnd	16k	70	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	16k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	16k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	16k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	16k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	16k	50	999	99000	0	1	1,00	1,00	0,00	0,00	1,00
mvHash	FragmentRand	16k	60	555	99000	0	445	0,56	1,00	0,00	0,45	1,00
mvHash	FragmentRand	16k	70	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	16k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	16k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	16k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mvHash	FragmentRand	16k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	16k	0,5	248	99000	0	752	0,25	1,00	0,00	0,75	1,00
mvHash	Noise	16k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	16k	1,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	16k	2	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	16k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	16k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	16k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	16k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	16k	10	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	16k	20	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	16k	30	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	SingleCommor	16k	40	10	99000	0	990	0,01	1,00	0,00	0,99	1,00
mvHash	SingleCommor	16k	50	604	99000	0	396	0,60	1,00	0,00	0,40	1,00
sdhash	Alignment	16k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	16k	100	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	16k	200	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	16k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	16k	70	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	16k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	16k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	16k	95	50	99000	0	950	0,05	1,00	0,00	0,95	1,00
sdhash	FragmentEnd	16k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentEnd	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	16k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	16k	70	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	16k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	16k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	16k	95	48	99000	0	952	0,05	1,00	0,00	0,95	1,00
sdhash	FragmentRand	16k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	Noise	16k	0,5	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	16k	1	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	16k	1,5	814	99000	0	186	0,81	1,00	0,00	0,19	1,00
sdhash	Noise	16k	2	52	99000	0	948	0,05	1,00	0,00	0,95	1,00
sdhash	Noise	16k	2,5	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	16k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	16k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	16k	5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	16k	10	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	16k	20	534	99000	0	466	0,53	1,00	0,00	0,47	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
sdhash	SingleCommor	16k	30	866	99000	0	134	0,87	1,00	0,00	0,13	1,00
sdhash	SingleCommor	16k	40	976	99000	0	24	0,98	1,00	0,00	0,02	1,00
sdhash	SingleCommor	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	16k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	16k	100	975	99000	0	25	0,98	1,00	0,00	0,03	1,00
ssdeep	Alignment	16k	200	933	99000	0	67	0,93	1,00	0,00	0,07	1,00
ssdeep	FragmentEnd	16k	50	962	99000	0	38	0,96	1,00	0,00	0,04	1,00
ssdeep	FragmentEnd	16k	60	692	99000	0	308	0,69	1,00	0,00	0,31	1,00
ssdeep	FragmentEnd	16k	70	48	99000	0	952	0,05	1,00	0,00	0,95	1,00
ssdeep	FragmentEnd	16k	80	21	99000	0	979	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentEnd	16k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	16k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	16k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	16k	50	976	99000	0	24	0,98	1,00	0,00	0,02	1,00
ssdeep	FragmentRand	16k	60	709	99000	0	291	0,71	1,00	0,00	0,29	1,00
ssdeep	FragmentRand	16k	70	32	99000	0	968	0,03	1,00	0,00	0,97	1,00
ssdeep	FragmentRand	16k	80	12	99000	0	988	0,01	1,00	0,00	0,99	1,00
ssdeep	FragmentRand	16k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	16k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	16k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	16k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	16k	0,5	18	99000	0	982	0,02	1,00	0,00	0,98	1,00
ssdeep	Noise	16k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	16k	1,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	16k	2	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	16k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	16k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	16k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	16k	5	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	16k	10	46	99000	0	954	0,05	1,00	0,00	0,95	1,00
ssdeep	SingleCommor	16k	20	624	99000	0	376	0,62	1,00	0,00	0,38	1,00
ssdeep	SingleCommor	16k	30	939	99000	0	61	0,94	1,00	0,00	0,06	1,00
ssdeep	SingleCommor	16k	40	992	99000	0	8	0,99	1,00	0,00	0,01	1,00
ssdeep	SingleCommor	16k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	64k	25	1000	98933	67	0	1,00	1,00	0,00	0,00	0,94
mrsh	Alignment	64k	50	1000	98853	147	0	1,00	1,00	0,00	0,00	0,87
mrsh	Alignment	64k	100	1000	98814	186	0	1,00	1,00	0,00	0,00	0,84
mrsh	Alignment	64k	200	1000	98754	246	0	1,00	1,00	0,00	0,00	0,80
mrsh	FragmentEnd	64k	50	1000	98908	92	0	1,00	1,00	0,00	0,00	0,92
mrsh	FragmentEnd	64k	60	1000	98987	13	0	1,00	1,00	0,00	0,00	0,99
mrsh	FragmentEnd	64k	70	1000	98997	3	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	64k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	64k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00

			Opt	**TP**	**TN**	**FP**	**FN**	**TPR**	**TNR**	**FPR**	**FNR**	**Precision**
mrsh	FragmentEnd	64k	95	1000	98977	23	0	1,00	1,00	0,00	0,00	0,98
mrsh	FragmentEnd	64k	97	925	98723	277	75	0,93	1,00	0,00	0,08	0,77
mrsh	FragmentEnd	64k	99	19	98980	20	981	0,02	1,00	0,00	0,98	0,49
mrsh	FragmentRand	64k	50	1000	98884	116	0	1,00	1,00	0,00	0,00	0,90
mrsh	FragmentRand	64k	60	1000	98983	17	0	1,00	1,00	0,00	0,00	0,98
mrsh	FragmentRand	64k	70	1000	98998	2	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	64k	80	1000	98999	1	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	64k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	64k	95	1000	98966	34	0	1,00	1,00	0,00	0,00	0,97
mrsh	FragmentRand	64k	97	911	98720	280	89	0,91	1,00	0,00	0,09	0,76
mrsh	FragmentRand	64k	99	17	98982	18	983	0,02	1,00	0,00	0,98	0,49
mrsh	Noise	64k	0,5	1000	98876	124	0	1,00	1,00	0,00	0,00	0,89
mrsh	Noise	64k	1	144	98885	115	856	0,14	1,00	0,00	0,86	0,56
mrsh	Noise	64k	1,5	26	98887	113	974	0,03	1,00	0,00	0,97	0,19
mrsh	Noise	64k	2	12	98892	108	988	0,01	1,00	0,00	0,99	0,10
mrsh	Noise	64k	2,5	7	98864	136	993	0,01	1,00	0,00	0,99	0,05
mrsh	SingleCommor	64k	1	3	98896	104	997	0,00	1,00	0,00	1,00	0,03
mrsh	SingleCommor	64k	3	36	98877	123	964	0,04	1,00	0,00	0,96	0,23
mrsh	SingleCommor	64k	5	38	98870	130	962	0,04	1,00	0,00	0,96	0,23
mrsh	SingleCommor	64k	10	47	98876	124	953	0,05	1,00	0,00	0,95	0,27
mrsh	SingleCommor	64k	20	611	98870	130	389	0,61	1,00	0,00	0,39	0,82
mrsh	SingleCommor	64k	30	837	98839	161	163	0,84	1,00	0,00	0,16	0,84
mrsh	SingleCommor	64k	40	959	98861	139	41	0,96	1,00	0,00	0,04	0,87
mrsh	SingleCommor	64k	50	996	98859	141	4	1,00	1,00	0,00	0,00	0,88
mvHash	Alignment	64k	25	1000	368	98632	0	1,00	0,00	1,00	0,00	0,01
mvHash	Alignment	64k	50	1000	385	98615	0	1,00	0,00	1,00	0,00	0,01
mvHash	Alignment	64k	100	1000	21	98979	0	1,00	0,00	1,00	0,00	0,01
mvHash	Alignment	64k	200	1000	1	98999	0	1,00	0,00	1,00	0,00	0,01
mvHash	FragmentEnd	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	FragmentEnd	64k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	FragmentEnd	64k	70	641	99000	0	359	0,64	1,00	0,00	0,36	1,00
mvHash	FragmentEnd	64k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	64k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	64k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	64k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	64k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	FragmentRand	64k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mvHash	FragmentRand	64k	70	624	99000	0	376	0,62	1,00	0,00	0,38	1,00
mvHash	FragmentRand	64k	80	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	64k	90	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	64k	95	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	64k	97	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	64k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	64k	0,5	1000	685	98315	0	1,00	0,01	0,99	0,00	0,01

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mvHash	Noise	64k	1	1000	721	98279	0	1,00	0,01	0,99	0,00	0,01
mvHash	Noise	64k	1,5	1000	733	98267	0	1,00	0,01	0,99	0,00	0,01
mvHash	Noise	64k	2	999	683	98317	1	1,00	0,01	0,99	0,00	0,01
mvHash	Noise	64k	2,5	1000	684	98316	0	1,00	0,01	0,99	0,00	0,01
mvHash	SingleCommor	64k	1	995	661	98339	5	1,00	0,01	0,99	0,01	0,01
mvHash	SingleCommor	64k	3	1000	574	98426	0	1,00	0,01	0,99	0,00	0,01
mvHash	SingleCommor	64k	5	999	624	98376	1	1,00	0,01	0,99	0,00	0,01
mvHash	SingleCommor	64k	10	1000	668	98332	0	1,00	0,01	0,99	0,00	0,01
mvHash	SingleCommor	64k	20	1000	700	98300	0	1,00	0,01	0,99	0,00	0,01
mvHash	SingleCommor	64k	30	1000	563	98437	0	1,00	0,01	0,99	0,00	0,01
mvHash	SingleCommor	64k	40	1000	601	98399	0	1,00	0,01	0,99	0,00	0,01
mvHash	SingleCommor	64k	50	1000	566	98434	0	1,00	0,01	0,99	0,00	0,01
sdhash	Alignment	64k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	64k	100	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	64k	200	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	70	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	95	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	97	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	64k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	FragmentRand	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	64k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	64k	70	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	64k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	64k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	64k	95	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	64k	97	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	64k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	Noise	64k	0,5	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	64k	1	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	64k	1,5	958	99000	0	42	0,96	1,00	0,00	0,04	1,00
sdhash	Noise	64k	2	25	99000	0	975	0,03	1,00	0,00	0,98	1,00
sdhash	Noise	64k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	64k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	64k	3	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	64k	5	114	99000	0	886	0,11	1,00	0,00	0,89	1,00
sdhash	SingleCommor	64k	10	826	99000	0	174	0,83	1,00	0,00	0,17	1,00
sdhash	SingleCommor	64k	20	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	64k	30	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	64k	40	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
ssdeep	Alignment	64k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	64k	100	968	99000	0	32	0,97	1,00	0,00	0,03	1,00
ssdeep	Alignment	64k	200	933	99000	0	67	0,93	1,00	0,00	0,07	1,00
ssdeep	FragmentEnd	64k	50	950	99000	0	50	0,95	1,00	0,00	0,05	1,00
ssdeep	FragmentEnd	64k	60	680	99000	0	320	0,68	1,00	0,00	0,32	1,00
ssdeep	FragmentEnd	64k	70	42	99000	0	958	0,04	1,00	0,00	0,96	1,00
ssdeep	FragmentEnd	64k	80	17	99000	0	983	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentEnd	64k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	64k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	64k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	64k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	64k	50	975	99000	0	25	0,98	1,00	0,00	0,03	1,00
ssdeep	FragmentRand	64k	60	694	99000	0	306	0,69	1,00	0,00	0,31	1,00
ssdeep	FragmentRand	64k	70	34	99000	0	966	0,03	1,00	0,00	0,97	1,00
ssdeep	FragmentRand	64k	80	17	99000	0	983	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentRand	64k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	64k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	64k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	64k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	64k	0,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	64k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	64k	1,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	64k	2	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	64k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	64k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	64k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	64k	5	2	99000	0	998	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	64k	10	70	99000	0	930	0,07	1,00	0,00	0,93	1,00
ssdeep	SingleCommor	64k	20	625	99000	0	375	0,63	1,00	0,00	0,38	1,00
ssdeep	SingleCommor	64k	30	946	99000	0	54	0,95	1,00	0,00	0,05	1,00
ssdeep	SingleCommor	64k	40	997	99000	0	3	1,00	1,00	0,00	0,00	1,00
ssdeep	SingleCommor	64k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	Alignment	256k	25	1000	98951	49	0	1,00	1,00	0,00	0,00	0,95
mrsh	Alignment	256k	50	1000	98935	65	0	1,00	1,00	0,00	0,00	0,94
mrsh	Alignment	256k	100	1000	98935	65	0	1,00	1,00	0,00	0,00	0,94
mrsh	Alignment	256k	200	1000	98918	82	0	1,00	1,00	0,00	0,00	0,92
mrsh	FragmentEnd	256k	50	1000	98814	186	0	1,00	1,00	0,00	0,00	0,84
mrsh	FragmentEnd	256k	60	1000	98933	67	0	1,00	1,00	0,00	0,00	0,94
mrsh	FragmentEnd	256k	70	1000	98964	36	0	1,00	1,00	0,00	0,00	0,97
mrsh	FragmentEnd	256k	80	1000	98983	17	0	1,00	1,00	0,00	0,00	0,98
mrsh	FragmentEnd	256k	90	1000	98989	11	0	1,00	1,00	0,00	0,00	0,99
mrsh	FragmentEnd	256k	95	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	256k	97	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentEnd	256k	99	996	98539	461	4	1,00	1,00	0,00	0,00	0,68

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mrsh	FragmentRand	256k	50	1000	98856	144	0	1,00	1,00	0,00	0,00	0,87
mrsh	FragmentRand	256k	60	1000	98957	43	0	1,00	1,00	0,00	0,00	0,96
mrsh	FragmentRand	256k	70	1000	98984	16	0	1,00	1,00	0,00	0,00	0,98
mrsh	FragmentRand	256k	80	1000	98988	12	0	1,00	1,00	0,00	0,00	0,99
mrsh	FragmentRand	256k	90	1000	98990	10	0	1,00	1,00	0,00	0,00	0,99
mrsh	FragmentRand	256k	95	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	256k	97	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
mrsh	FragmentRand	256k	99	995	98529	471	5	1,00	1,00	0,00	0,01	0,68
mrsh	Noise	256k	0,5	1000	98940	60	0	1,00	1,00	0,00	0,00	0,94
mrsh	Noise	256k	1	62	98938	62	938	0,06	1,00	0,00	0,94	0,50
mrsh	Noise	256k	1,5	10	98944	56	990	0,01	1,00	0,00	0,99	0,15
mrsh	Noise	256k	2	3	98931	69	997	0,00	1,00	0,00	1,00	0,04
mrsh	Noise	256k	2,5	1	98944	56	999	0,00	1,00	0,00	1,00	0,02
mrsh	SingleCommor	256k	1	5	98939	61	995	0,01	1,00	0,00	1,00	0,08
mrsh	SingleCommor	256k	3	30	98947	53	970	0,03	1,00	0,00	0,97	0,36
mrsh	SingleCommor	256k	5	236	98930	70	764	0,24	1,00	0,00	0,76	0,77
mrsh	SingleCommor	256k	10	855	98945	55	145	0,86	1,00	0,00	0,15	0,94
mrsh	SingleCommor	256k	20	1000	98925	75	0	1,00	1,00	0,00	0,00	0,93
mrsh	SingleCommor	256k	30	1000	98945	55	0	1,00	1,00	0,00	0,00	0,95
mrsh	SingleCommor	256k	40	1000	98949	51	0	1,00	1,00	0,00	0,00	0,95
mrsh	SingleCommor	256k	50	1000	98941	59	0	1,00	1,00	0,00	0,00	0,94
mvHash	Alignment	256k	25	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	Alignment	256k	50	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	Alignment	256k	100	909	19746	79254	91	0,91	0,20	0,80	0,09	0,01
mvHash	Alignment	256k	200	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	256k	50	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	FragmentEnd	256k	60	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	FragmentEnd	256k	70	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	FragmentEnd	256k	80	663	41589	57411	337	0,66	0,42	0,58	0,34	0,01
mvHash	FragmentEnd	256k	90	663	99000	0	337	0,66	1,00	0,00	0,34	1,00
mvHash	FragmentEnd	256k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	256k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentEnd	256k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	256k	50	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	FragmentRand	256k	60	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	FragmentRand	256k	70	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	FragmentRand	256k	80	654	41955	57045	346	0,65	0,42	0,58	0,35	0,01
mvHash	FragmentRand	256k	90	484	99000	0	516	0,48	1,00	0,00	0,52	1,00
mvHash	FragmentRand	256k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	256k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	FragmentRand	256k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
mvHash	Noise	256k	0,5	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	Noise	256k	1	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	Noise	256k	1,5	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	Noise	256k	2	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01

			Opt	TP	TN	FP	FN	TPR	TNR	FPR	FNR	Precision
mvHash	SingleCommor	256k	1	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	SingleCommor	256k	3	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	SingleCommor	256k	5	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	SingleCommor	256k	10	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	SingleCommor	256k	20	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	SingleCommor	256k	30	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	SingleCommor	256k	40	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
mvHash	SingleCommor	256k	50	1000	0	99000	0	1,00	0,00	1,00	0,00	0,01
sdhash	Alignment	256k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	256k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	256k	100	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Alignment	256k	200	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	70	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	95	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	97	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentEnd	256k	99	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	60	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	70	1000	98999	1	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	80	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	90	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	95	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	97	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	FragmentRand	256k	99	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	256k	0,5	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	256k	1	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	Noise	256k	1,5	981	99000	0	19	0,98	1,00	0,00	0,02	1,00
sdhash	Noise	256k	2	26	99000	0	974	0,03	1,00	0,00	0,97	1,00
sdhash	Noise	256k	2,5	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
sdhash	SingleCommor	256k	1	14	99000	0	986	0,01	1,00	0,00	0,99	1,00
sdhash	SingleCommor	256k	3	762	99000	0	238	0,76	1,00	0,00	0,24	1,00
sdhash	SingleCommor	256k	5	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	256k	10	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	256k	20	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	256k	30	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	256k	40	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
sdhash	SingleCommor	256k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	256k	25	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	256k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00
ssdeep	Alignment	256k	100	976	99000	0	24	0,98	1,00	0,00	0,02	1,00
ssdeep	Alignment	256k	200	937	99000	0	63	0,94	1,00	0,00	0,06	1,00

			Opt	**TP**	**TN**	**FP**	**FN**	**TPR**	**TNR**	**FPR**	**FNR**	**Precision**
ssdeep	FragmentEnd	256k	50	972	99000	0	28	0,97	1,00	0,00	0,03	1,00
ssdeep	FragmentEnd	256k	60	698	99000	0	302	0,70	1,00	0,00	0,30	1,00
ssdeep	FragmentEnd	256k	70	39	99000	0	961	0,04	1,00	0,00	0,96	1,00
ssdeep	FragmentEnd	256k	80	21	99000	0	979	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentEnd	256k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	256k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	256k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentEnd	256k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	256k	50	968	99000	0	32	0,97	1,00	0,00	0,03	1,00
ssdeep	FragmentRand	256k	60	682	99000	0	318	0,68	1,00	0,00	0,32	1,00
ssdeep	FragmentRand	256k	70	43	99000	0	957	0,04	1,00	0,00	0,96	1,00
ssdeep	FragmentRand	256k	80	24	99000	0	976	0,02	1,00	0,00	0,98	1,00
ssdeep	FragmentRand	256k	90	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	256k	95	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	256k	97	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	FragmentRand	256k	99	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	256k	0,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	256k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	256k	1,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	256k	2	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	Noise	256k	2,5	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	256k	1	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	256k	3	0	99000	0	1000	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	256k	5	1	99000	0	999	0,00	1,00	0,00	1,00	1,00
ssdeep	SingleCommor	256k	10	72	99000	0	928	0,07	1,00	0,00	0,93	1,00
ssdeep	SingleCommor	256k	20	658	99000	0	342	0,66	1,00	0,00	0,34	1,00
ssdeep	SingleCommor	256k	30	954	99000	0	46	0,95	1,00	0,00	0,05	1,00
ssdeep	SingleCommor	256k	40	999	99000	0	1	1,00	1,00	0,00	0,00	1,00
ssdeep	SingleCommor	256k	50	1000	99000	0	0	1,00	1,00	0,00	0,00	1,00

Bibliography

[1] Cory Altheide and Harlan Carvey. *Digital Forensics with Open Source Tools: Using Open Source Platform Tools for Performing Computer Forensics on Target Systems: Windows, Mac, Linux, Unix, etc*, volume 1. Syngress Media, May 2011.

[2] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. In *11th annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 794–803, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[3] Knut Petter Åstebøl. mvhash - a new approach for fuzzy hashing. Master's thesis, Gjøvik University College, 2012.

[4] Harald Baier and Frank Breitinger. Security Aspects of Piecewise Hashing in Computer Forensics. *IT Security Incident Management & IT Forensics (IMF)*, pages 21–36, May 2011.

[5] Sushil Bhattacharjee and Martin Kutter. Compression tolerant image authentication. In *1998 International Conference on Image Processing*, volume 1, pages 435–439. IEEE Computer Society, 1998.

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

[7] Ranjan Bose. *Information Theory, Coding and Cryptography.* Tata McGraw-Hill Education, 2008.

[8] Tim Boyles. *CCNA Security Study Guide: Exam 640-553*, volume 1. John Wiley & Sons, March 2010.

[9] Frank Breitinger. Security Aspects of Fuzzy Hashing. Master's thesis, Hochschule Darmstadt, February 2011. last accessed on 2012-07-05.

[10] Frank Breitinger. Similar file detection on network traffic using approximate matching. *6th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, Sept 2014. in review.

[11] Frank Breitinger, Knut Petter Åstebøl, Harald Baier, and Christoph Busch. mvhash-b - a new approach for similarity preserving hashing. In *IT Security Incident Management and IT Forensics (IMF), 2013 Seventh International Conference on*, pages 33–44, March 2013.

[12] Frank Breitinger and Harald Baier. A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance. In *7th annual Conference on Digital Forensics, Security and Law (ADFSL)*, pages 89–100, May 2012.

[13] Frank Breitinger and Harald Baier. Performance issues about context-triggered piecewise hashing. In Pavel Gladyshev and Marcus K. Rogers, editors, *Digital Forensics and Cyber Crime*, volume 88 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 141–155. Springer Berlin Heidelberg, 2012.

[14] Frank Breitinger and Harald Baier. Properties of a similarity preserving hash function and their realization in sdhash. In *Information Security for South Africa (ISSA), 2012*, pages 1–8, 2012.

[15] Frank Breitinger and Harald Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In Marcus Rogers and Kathryn C. Seigfried-Spellar, editors, *Digital Forensics and Cyber Crime*, volume 114 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 167–182. Springer Berlin Heidelberg, 2013.

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

[17] Frank Breitinger, Harald Baier, and Douglas White. On the database lookup problem of approximate matching. *1st Digital Forensics Research Conference EU (DFRWS-EU'14)*, 2014.

[18] Frank Breitinger, Barbara Guttman, Michael McCarrin, and Vassil Roussev. Approximate matching: Definition and terminology. *NIST Special Publication 800-168*, DRAFT, 2014.

[19] Frank Breitinger, Huajian Liu, Christian Winter, Harald Baier, Alexey Rybalchenko, and Martin Steinebach. Towards a process model for hash functions in digital forensics. *5th International Conference on Digital Forensics & Cyber Crime*, Sept 2013.

[20] Frank Breitinger and Kaloyan Petrov. Reducing time cost in hashing operations. In *Ninth Annual IFIP WG 11.9 International Conference on Digital Forensics (IFIP WG11.9)*, January 2013.

[21] Frank Breitinger, Christian Rathgeb, and Harald Baier. An efficient similarity digests database lookup - a logarithmic divide & conquer approach. *6th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, Sept 2014. in review.

[22] Frank Breitinger and Vassil Roussev. Automated evaluation of approximate matching algorithms on real data. *1st Digital Forensics Research Conference EU (DFRWS-EU'14)*, 2014.

[23] Frank Breitinger, Georgios Stivaktakis, and Harald Baier. FRASH: A framework to test algorithms of similarity hashing. In *13th Digital Forensics Research Conference (DFRWS'13)*, Monterey, August 2013.

[24] Frank Breitinger, Georgios Stivaktakis, and Vassil Roussev. Evaluating detection error trade-offs for bytewise approximate matching algorithms. *5th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, Sept 2013. Abstract.

[25] Frank Breitinger, Georgios Stivaktakis, and Vassil Roussev. Evaluating detection error trade-offs for bytewise approximate matching algorithms. *Digital Investigation*, Mai 2014. perapring for printing.

[26] Frank Breitinger, Christian Winter, York Yannikos, Tobias Fink, and Michael Seefried. Reducing data for forensic investigations using approximate matching. In *Tenth Annual IFIP WG 11.9 International Conference on Digital Forensics (IFIP WG11.9)*, January 2014.

[27] Frank Breitinger, Georg Ziroff, Steffen Lange, and Harald Baier. sahash: Similarity hashing based on levenshtein distance. In *Tenth Annual IFIP WG 11.9 International Conference on Digital Forensics (IFIP WG11.9)*, January 2014.

[28] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.

[29] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise Independent Permutations. *Journal of Computer and System Sciences*, 60:327–336, 1998.

[30] Andrei Z. Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2005.

[31] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *SIGCOMM Comput. Commun. Rev.*, 35(2):71–82, April 2005.

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

[33] Digital Corpora. NPS Corpus. `http://digitalcorpora.org/corpora/files`, April 2013.

[34] Digital Corpora. NPS Corpus. `http://digitalcorpora.org/corpora/disk-images`, May 2013.

[35] John Daugman. How iris recognition works. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1):21–30, 2004.

[36] Cedric De Roover, Christophe De Vleeschouwer, Frédéric Lefèbvre, and Benoit Macq. Robust image hashing based on radial variance of pixels. In *ICIP 2005*, volume 3, pages 77–80. IEEE, 2005.

[37] Andreas Dewald and Felix Freiling. Is Computer Forensics a Forensic Science? In Max-Planck-Institut für ausländisches und internationales Strafrecht and Universität Freiburg, editors, *Proceedings of Current Issues in IT Security 2012*, pages 0–0. Duncker & Humblot, 2012.

[38] Sarang Dharmapurikar, Praveen Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, pages 44–51, 2003.

[39] John Eakins and Margaret Graham. Content-based image retrieval. JTAP report 39, University of Northumbria at Newcastle, October 1999.

[40] Jiri Fridrich. Robust bit extraction from images. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 536–540. IEEE Computer Society, 1999.

[41] Patrick Gallagher. Secure Hash Standard (SHS). Technical report, National Institute of Standards and Technologies, Federal Information Processing Standards Publication 180-1, 1995.

[42] Simson L. Garfinkel. Digital forensics research: The next 10 years. *Digitial Investigation*, 7:64–73, August 2010.

[43] Alexander Geschonneck. Ceic 2008: Block based hash analyse mit encase. `http://computer-forensik.org/2008/04/30/ceic-2008-block-based-hash-analyse-mit-encase/`, April 2008.

[44] Oscar Andrés Giraldo Triana. Fast similarity search for robust image hashes. Bachelor thesis, Technische Universität Darmstadt, 2012.

[45] AUTEM GmbH. Perkeo++. `http://www.perkeo.com`, 2012.

[46] Seed Forensics GmbH. Artemis Triage Child Pornography Tool. `http://www.artemis-scan.de`, July 2012.

[47] Michael Grega, Damian Bryk, and Maciej Napora. INACT—INDECT advanced image cataloguing tool. *Multimedia Tools and Applications*, July 2012.

[48] Vikas Gupta. File fragment detection on network traffic using similarity hashing. Master's thesis, Denmark Technical University, 2013.

[49] Harald Baier and Christian Dichtelmüller. Hatenreduktion mittels kryptographischer Hashfunktionen in der IT-Forensik: Nur ein Mythos? In *DACH Security 2012*, pages 278–287, September 2012.

[50] Ewa Huebner and Stefano Zanero. *Open Source Software for Digital Forensics*, volume 1. Springer, February 2010.

[51] Dustin Hurlbut. Fuzzy hashing for digital forensic investigators. Technical report, AccessData, January 2009.

[52] ISO/IEC JTC1 SC27 Security Techniques. *ISO/IEC 24745:2011. Information Technology - Security Techniques - Biometric Information Protection.* International Organization for Standardization, 2011.

[53] Paul Jaccard. Distribution de la flore alpine dans le bassin des drouces et dans quelques regions voisines. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:241–272, 1901.

[54] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11:37–50, February 1912.

[55] Toshikazu Kato. Database architecture for content-based image retrieval. In *Image Storage and Retrieval Systems*, volume 1662 of *Proc. SPIE*, pages 112–123. IS&T/SPIE Electronic Imaging, February 9–14, 1992, San Jose, California, April 1992.

[56] Seema Kedar. *Database Management System.* Technical Publications Pune, 2009.

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

[58] George Lawton. New technology prevents data leakage. *Computer*, 41(9):14–17, 2008.

[59] Frédéric Lefèbvre, Benoit Macq, and Jean-Didier Legat. Rash: Radon soft hash algorithm. In *EUSIPCO'2002*, volume 1, pages 299–302. TéSA, 2002.

[60] David G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, number 2 in ICCV'99, pages 1150–1157. IEEE Computer Society, 1999.

[61] Xudong Lv and Z. Jane Wang. Perceptual image hashing based on shape contexts and local feature points. *IEEE T. Inf. Foren. Sec.*, 7(3):1081–1093, June 2012.

[62] Balakrishna S. Maddodi, Vidya Attigeri, and A.K. Karunakar. Data Deduplication Techniques and Analysis. In *Emerging Trends in Engineering and Technology (ICETET)*, pages 664–668, nov. 2010.

[63] Udi Manber. Finding similar files in a large file system. In *Technical Conference on USENIX Winter 1994*, pages 1–10, 1994.

[64] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, volume 5. CRC Press, August 2001.

[65] Vishal Monga and Brian L. Evans. Perceptual image hashing via feature points: Performance evaluation and tradeoffs. *IEEE T. Image Process.*, 15(11):3453–3466, November 2006.

[66] MPEG. Information technology – multimedia content description interface – part 3: Visual. Technical Report 15938-3, ISO/IEC, 2002.

[67] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558 –560, may 1990.

[68] NIST Information Technology Laboratory. National Software Reference Library, 2003–2014. `http://www.nsrl.nist.gov`.

[69] Landon Curt Noll. Fnv hash. `http://www.isthe.com/chongo/tech/comp/fnv/index.html`, 1994–2012.

[70] Mark M. Pollitt. An ad hoc review of digital forensic models. In *Systematic Approaches to Digital Forensic Engineering, 2007. SADFE 2007. Second International Workshop on*, pages 43–54, 2007.

[71] Jon Postel. Rfc 701: Internet protocol. 1981.

[72] Thomas Proter. The perils of deep packet inspection. *Symantec.com*, October 2010.

[73] Maria Paula Queluz. Towards robust, content based techniques for image authentication. In *Multimedia Signal Processing*, pages 297–302. IEEE, 1998.

[74] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR1581, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1981.

[75] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets.* Cambridge University Press, Cambridge, 2012.

[76] Christian Rathgeb, Frank Breitinger, and Christoph Busch. Alignment-free cancelable iris biometric templates based on adaptive bloom filters. In *Biometrics (ICB), 2013 International Conference on*, pages 1–8, June.

[77] Christian Rathgeb, Frank Breitinger, Christoph Busch, and Harald Baier. On the application of bloom filters to iris biometrics. *IET Biometrics*, 2013.

[78] Marcus K. Rogers, James Goldman, Rick Mislan, Timothy Wedge, and Steve Debrota. Computer forensics field triage process model. *Conference on Digital Forensics, Security and Law*, pages 27–40, 2006.

[79] Vassil Roussev. Building a better similarity trap with statistically improbable features. In *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*, pages 1–10, 2009.

[80] Vassil Roussev. Data fingerprinting with similarity digests. In Kam-Pui Chow and Sujeet Shenoi, editors, *Advances in Digital Forensics VI*, volume 337 of *IFIP Advances in Information and Communication Technology*, pages 207–226. Springer Berlin Heidelberg, 2010.

[81] Vassil Roussev. An evaluation of forensic similarity hashes. *Digital Investigation*, 8:34–41, August 2011.

[82] Vassil Roussev. t5-corpus. `http://roussev.net/t5/t5-corpus.zip`, February 2011.

[83] Vassil Roussev. Managing terabyte-scale investigations with similarity digests. In Gilbert Peterson and Sujeet Shenoi, editors, *Advances in Digital Forensics VIII*, volume 383 of *IFIP Advances in Information and Communication Technology*, pages 19–34. Springer Berlin Heidelberg, 2012.

[84] Vassil Roussev, Yixin Chen, Timothy Bourg, and Golden G. Richard III. md5bloom: Forensic filesystem hashing revisited. *Digital Investigation*, 3:82–90, 2006.

[85] Vassil Roussev, Golden G. Richard III, and Lodovico Marziale. Multi-resolution similarity hashing. *Digital Investigation*, 4:105–113, September 2007.

[86] Andrew Rukhin, Juan Soto, James Nechvatal, et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, National Institute of Standards and Technology, `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`, April 2010.

[87] Caitlin Sadowski and Greg Levin. Simhash: Hash-based similarity detection. http://simhash.googlecode.com/svn/ trunk/paper/SimHashWithBib.pdf, December 2007.

[88] SANS Institute. Understanding and selecting a data loss prevention solution, 2010.

[89] Claude E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, pages 3–55, January January 2001.

[90] Guidance Software. Network-enabled incident response and endpoint data control through cyberforensics. `http://www.realediscovery.com/WorkArea/DownloadAsset.aspx?id=1000000404`, 2010.

[91] Stackoverflow.com. What do 'real', 'user' and 'sys' mean in the output of time(1)?

[92] François-Xavier Stanaert, Frédéric Lefèbvre, Gaël Rouvroy, Benoit Macq, Jean-Jacques Quisquater, and Jean-Didier Legat. Practical evaluation of a radial soft hash algorithm. In *ITCC 2005*, volume 2, pages 89–94. IEEE Computer Society, 2005.

[93] Martin Steinebach. Robust hashing for efficient forensic analysis of image sets. In *Digital Forensics and Cyber Crime*, volume 88 of *LNICST*, pages 180–187. Springer, 2012.

[94] Marting Steinebach, Huajian Liu, and York Yannikos. Forbild: Efficient robust image hashing. *SPIE 8303, Media Watermarking, Security, and Forensics*, 2012.

[95] Markus Stricker and Markus Orengo. Similarity of color images. In *Storage and Retrieval for Image and Video Databases III*, volume 2420 of *Proc. SPIE*, pages 381–392. IS&T/SPIE Electronic Imaging, February 5–10, 1995, San Jose, California, March 1995.

[96] Michael J. Swain and Dana H. Ballard. Color indexing. *Int. J. Comput. Vision*, 7(1):11–32, November 1991.

[97] Andrew S. Tanenbaum. *Computer networks.* Prentice Hall PTR, 2003.

[98] Harold F. Tipton and Micki Krause. *Information Security Management Handbook.* Auerbach Publications, 5 edition, December 2003.

[99] Andrew Tridgell. spamsum. `http://www.samba.org/ftp/unpacked/junkcode/spamsum/`, 2002–2009. last accessed on 2013-04-10.

[100] Matthew Turk and Alex Pentland. Face recognition using eigenfaces. In *Computer Vision and Pattern Recognition, 1991. IEEE*, pages 586–591, jun 1991.

[101] Ramarathnam Venkatesan, S.-M. Koon, M. H. Jakubowski, and P. Moulin. Robust image hashing. In *2000 International Conference on Image Processing*, volume 3, pages 664–666. IEEE, 2000.

[102] Li Weng and Bart Preneel. From image hashing to video hashing. In *16th International Multimedia Modeling Conference*, volume 5916, pages 662–668, 2010.

[103] Douglas White. Hashing of File Blocks: When Exact Matches Are Not Useful. Presentation at American Academy of Forensic Sciences (AAFS), 2008.

[104] Christian Winter, Markus Schneider, and York Yannikos. F2S2: Fast forensic similarity search through indexing piecewise hashsignatures. *Digital Investigation*, 10(4):361–371, 2013.

[105] X-Ways. Winhex & x-ways forensics newsletter archive. `http://www.winhex.com/winhex/mailings/index.html`, July 2010.

[106] Shijun Xiang and Hyoung Joong Kim. *Transactions on Data Hiding and Multimedia Security VI*, volume 6730 of *LNCS*, chapter Histogram-Based Image Hashing for Searching Content-Preserving Copies, pages 83–108. Springer, 2011.

[107] Liehua Xie and Gonzalo R. Arce. A class of authentication digital watermarks for secure multimedia communication. *IEEE T. Image Process.*, 10(11):1754–1764, November 2001.

[108] Bian Yang, Fan Gu, and Xiamu Niu. Block mean value based image perceptual hashing. In *Intelligent Information Hiding and Multimedia Multimedia Signal Processing*. IEEE Computer Society, 2006.

[109] Christoph Zauner. Implementation and benchmarking of perceptual image hash functions. Master's thesis, University of Applied Sciences Upper Austria, July 2010.

[110] Christoph Zauner, Martin Steinebach, and Eckehard Hermann. Rihamark: perceptual image hash benchmarking. In *Media Watermarking, Security, and Forensics III*, volume 7880 of *Proc. SPIE*, pages 7880 0X–1–15. IS&T/SPIE Electronic Imaging, January 23–27, 2011, San Francisco, California, February 2011.

[111] Georg Ziroff. Approaches to similarity-preserving hashing, Bachelor's thesis, Hochschule Darmstadt, February 2012.