University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master of Science in Computer Science	Spring semester, 2016 Open
Writer: Ole Tobiesen	…… (Writer's signature)
Faculty supervisor: Reggie Davidrajuh	
Thesis title: Data Fingerprinting -- Identifying Files and Tables with Hashing Schemes	
Credits (ECTS): 30	
Key words: Data Fingerprinting Fuzzy Hashing Machine Learning Merkle Trees Finite Fields Mersenne Primes	Pages: 145, including table of contents and appendix Enclosure: 4 7z archives Stavanger, 15 June 2016

Front page for master thesis
Faculty of Science and Technology
Decision made by the Dean October 30th 2009

Abstract

INTRODUCTION: Although hash functions are nothing new, these are not limited to cryptographic purposes. One important field is *data fingerprinting*. Here, the purpose is to generate a digest which serves as a fingerprint (or a license plate) that uniquely identifies a file. More recently, fuzzy fingerprinting schemes — which will scrap the avalanche effect in favour of detecting local changes — has hit the spotlight. The main purpose of this project is to find ways to classify text tables, and discover where potential changes or inconsitencies have happened.
METHODS: Large parts of this report can be considered applied discrete mathematics — and finite fields and combinatorics have played an important part. Rabin's fingerprinting scheme was tested extensively and compared against existing cryptographic algorithms, CRC and FNV. Moreover, a self-designed fuzzy hashing algorithm with the preliminary name No-Frills Hash has been created and tested against Nilsimsa and Spamsum. NFHash is based on Mersenne primes, and uses a sliding window to create a fuzzy hash. Futhermore, the usefullness of lookup tables (with partial seeds) were also explored. The fuzzy hashing algorithm has also been combined with a k-NN classifier to get an overview over it's ability to classify files. In addition to NFHash, Bloom filters combined with Merkle Trees have been the most important part of this report. This combination will allow a user to see where a change was made, despite the fact that hash functions are one-way. Large parts of this project has dealt with the study of other open-source libraries and applications, such as Cassandra and SSDeep — as well as how bitcoins work. Optimizations have played a crucial role as well; different approaches to a problem might lead to the same solution, but resource consumption can be very different.
RESULTS: The results have shown that the Merkle Tree-based approach can track changes to a table very quickly and efficiently, due to it being conservative when it comes to CPU resources. Moreover, the self-designed algorithm NFHash also does well in terms of file classification when it is coupled with a k-NN classifyer.
CONCLUSION: Hash functions refers to a very diverse set of algorithms, and not just algorithms that serve a limited purpose. Fuzzy Fingerprinting Schemes can still be considered to be at their infant stage, but a lot has still happened the last ten years. This project has introduced two new ways to create and compare hashes that can be compared to similar, yet not necessarily identical files — or to detect if (and to what extent) a file was changed. Note that the algorithms presented here should be considered prototypes, and still might need some large scale testing to sort out potential flaws.
KEYWORDS: *Fingerprinting, k-NN, Supervised Learning, Sliding Window, Rolling Hash, Classification, fuzzy fingerprinting, locality-sensitive hashing, one-way function, Merkle Trees, collision resistance, hash functions, Galois fields, Mersenne primes, Rabin's Fingerprinting Scheme, CRC, Bloom Filters, Jaccard, Damerau-Levenshtein*

For my fiancee, Dongjing

List of Figures

List of Tables

Listings

Acknowledgements

This project would not have been possible without the help and support from several people. I would like to thank the supervisors in this project, Derek Göbel (Avito LOOPS), Paolo Predonzani (Avito LOOPS) and Reggie Davidrajuh (UiS) for helping me with this thesis, for providing me with feedback and tips, for helping me better test and debug the library — and for offering good advice. Furthermore, I would also like to thank Chunlei Li (UiS) for providing me with good knowledge of hashing algorithms and applied discrete mathematics. All pictures used in this report are public domain or created by me.

No code in this project is subject to any patents or copyrights.

Acronyms

Table 1: Important Acronmyms

Abbrevation	**Meaning**
BFS	Breadth-First Search
CRC	Cyclic Redundancy Check
CSV	Comma Separated Value
DFS	Depth-First Search
FNV	Fowler-Noll-Vo
$g(x)$	Discriminant function of x
$GF(2)$	Binary Galois (Finite) Field
$H(a)$	Hashed output from input a
kNN	k-Nearest Neighbor
LSH	Locality-Sensitive Hashing
MD	Message Digest
MLE	Most Likely Estimate
μOps	Micro-Opearations
$NFHash$	No-Frills Hash
ω_i	Class i in the field of machine learning
$P(\omega_i)$	Prior distribution of class i
SHA	Secure Hashing Algorithm

Contents

> A mathematical formula should never be "owned" by anybody! Mathematics belong to God.
>
> Donald Knuth

Chapter I
Introduction

This chapter will briefly introduce the project, it's content, and what it is intended for. Readers are encouraged to skim through this report quickly to get an overview before reading it in detail, as it will introduce many new terminologies and concepts. Furthermore, it is somewhat mathematically intensive. It's main target group is people with a master's degree in computer science. However, the majority of this thesis is based on principles and technologies that are not part of the mandatory curriculum in most colleges and universities. Therefore, details should be given special consideration.

1 Objectives

To many people, hash-functions are a mysterious and uncharted branch of algorithms. It is widely known that these are one-way, and that they are indeed useful. However, people tend to underestimate the versatilitiy of these hash-functions, mainly associating hash functions with hash tables and password authentication.

The primary motivation behind this thesis is to explore data fingerprinting algorithms for indentifying tables and updates done on these — in addition to classifying them using a machine learning tool. Data fingerprinting simply refers to high performance hash functions, used to identify files quickly. These can be considered more accurate checksums, with an especially high focus on collision resistance.

As described by Avito — the company responsible for this research — data fingerprinting can be summarized as follows:

> "Fingerprinting is a technique that analyses a document of an arbitrary size to produce a much shorter string - the "fingerprint" - possibly of fixed size, that almost uniquely identifies the original document. In a sense, a fingerprint is a proxy for a document's identity: if two documents have the same fingerprint, we can infer that they are in fact the same document; if a document "changes" its fingerprint, we can infer that the contents of the document have been modified, effectively making it a different document. Fingerprinting can be applied at different levels of granularity. In Excel files, a fingerprint can be

> computed for a whole workbook, for a worksheet, for a row or even for a single cell. In this way modifications in the file can be detected at different levels, which has practical applications, e.g., in content versioning and change tracking."

This report will also look at how well-known discrete mathematical properties can be used to implement a new hash function, as well as how versatile hash functions can be — in terms of chunkwise hashing, ordinary hashing, and of course fuzzy hashing. After all, there is more to hashing than just finding exact matches; some approaches can be taken to make them a well-suited tool for matching *similar* rather than just identical content as well.

Currently existing fuzzy fingerprinting (hash functions changing the output to the same degree as the input) algorithms tend to be cumbersome to use and slow — or they are somewhat inaccurate and "rigid", and although they serve their purpose very well in forensics and spam detection, they are not that well suited for identifying and classifying large plaintext files or tables. To try to find a balance between pros and cons of each fuzzy hashing scheme, a prototype called No-Frills Hash will be presented later on. The other implementation presented in this report serves as a way to work around the fact that a hash function is one-way; a two-way hash function is an oxymoron, so it does not reverse the digest directly. Rather, it keeps track of which hash digest belongs where in a document by storing each hashed element in a node. This can be combined with first implementation, so that both implementations will complement each other.

The solutions presented here can also be used on other forms of plaintext rather than just raw data tables. However, since this is the main reason behind the project and why Avito requested this research, this is also what the report will mainly focus on.

1.1 Research Questions

In the project descriptions, the objectives were described as follows:

> "There are two goals in this research topic. The first one is to investigate fingerprinting algorithms that are computationally efficient. The second one is to investigate fuzzy fingerprints, i.e. fingerprints that not only say if a document has changed or not but also to which extent it has changed (e.g. on a scale of 0 to 100%) while retaining some similarity of content."

The research questions serving as guidelines in this project have therefore been:

1. Where are fingerprinting algorithms better suited than cryptographic hash functions — in the context of raw data tables?
2. How does fuzzy hashing work?
3. What can fuzzy hashing be used for — in terms of plaintext and tables?

4. How does one continue the work of others to make a new fuzzy hashing scheme?
5. Is there any way to bypass the one-way properties of hash functions, so that one can see where the changes or inconsistencies happened?

Most importantly: **Is there any way to more efficiently, by the use of hashing, tell to which extent two files were changed? Can hashing be used to detect where the changes happened? Finally: Can fuzzy hashing be used in the field of machine learning?**. These three questions have different solutions, but they can be merged into one solution, as will be seen later on in this report.

2 Outline

The literature review chapter (Chapter II) will contain the research of other mathematicians and engineers — and will revolve around fundamental mathematical prerequisites, what hash-functions are, what hash-functions do — and what hash functions are used for. Of course, this will mainly be about fingerprinting uses and not that much about the cryptographical uses. With that being said, some cryptographical uses will also be presented; understanding data fingerprinting (both fuzzy and ordinary) without understanding the security-related or mathematical prerequirements, can be seen as analogous to watching the second *The Godfather* movie without watching the first *The Godfather* movie.

The "Method and Design" chapter will mainly focus on how the two main implementations are designed, how they work, what their virtues and vices are, and what datastructures and algorithms were necessary to create them. This chapter will also present all datastructures and algorithms used that are not part of the curriculum in a typical master's degree (i.e. it will explain Bloom Filters in detail, but not stacks and so on). Chapter III is closely linked with Chapter IV, which explains the code and the steps taken to make the library perform well.

The Testing and Verifications chapter will contain the testing part, including the results, how the different algorithms compare to each other, and what conclusions can be drawn from the tests. Lastly, everything will be summarized and discussed in the final chapter, where further works will also be proposed.

3 Used Software

All software and algorithms used in the programming part of this project are open-source and freely available. The enclosed library is written in Java, but the report will also contain some x86-64 assembly code and pseudocode for illustrative purposes. The software SSDeep is an existing example of fuzzy fingerprinting being employed for practical tasks, but this is specifically designed for forensics. Nevertheless, studying

the source code to better understand rolling hashes proved useful. Also, studying the noSQL tool Cassandra — and the technology behind Bitcoins was important. The project has nothing to do with either of the two technologies, but they still contained vital datastructures useful in this project.

Software Used for Writing the Report:

1. TeXworks
2. LaTeX

Programming Tools Used for Writing the Library and Testing it:

1. Java
2. IntellijIDE
3. NetBeans 8.1
4. JUnit

Open-Source Software Studied to get Inspiration:

1. Apache POI
2. Apache Cassandra
3. SSDeep
4. Bitcoins

Software Used for Drawings and Illustrations:

1. yED
2. 3DS Max Design 2013
3. GiMP
4. Macromedia Flash 8
5. MS Excel 2013

4 Methods Used

literature Review

The literature review part is where stuff done by others was studied extensively. This was a corner stone in the project, due to the fact that this made it possible to see what already existed, what could be improved on, and which problems already had satisfactory solutions. Furthermore, different mathematical theorems were studied, so that new solutions could be found.

Analysis and Mathematical Reasoning

After the literature study, certain parts needed to be proven by "pen and paper" and by mathematics before it could be used in the program code — to ensure that there were no red herrings that could potentially lead to time being wasted.

Unit and Low-Level Testing

This can be seen as an extension of the previous phase. There was also some overlap between this phase and the previous one. Here, which algorithms to use in the project and which ones to discard was decided. Also, implementations of finite fields and Mersenne primes were tested. Lastly, different data structures such as the Bloom Filter and the Merkle Tree was put into use, while others (eg. the Trie) were shown to be not that well-suited to this project.

Full-scale implementation

Everything that proved viable in the previous phase was implemented properly in this phase. Also, No-Frills Hash was designed, based on the results from the previous phase concerning (among others) CRC-64, lookup tables and Mersenne Primes.

GUI-testing — and Debugging

After most of the code was finished, a GUI was designed to test on a larger scale. Some bugs were found and were subsequently corrected. This was done because a unit testing environment is "artificial" and constructed. Moreover, with a GUI, it is easier to test the approaches on many tables at the same time.

Full-scale testing and evaluation

The last phase is described in detail in Chapter V. Here, the performance of NFHash was tested against the performance of Nilsimsa and Spamsum (two currently existing, but very different approaches to fuzzy hashing), and the ability of the Merkle Tree-based approach to detect changes was tested. Classification was tested with the k-NN classifier.

5 Terminiology: Fuzzy or Locality-Sensitive Hashing?

The terms fuzzy hashing or locality-sensitive hashing roughly refer to the same thing, and there is no universally accepted definition for the former (the term was first used by dr. Andrew Tridgell in 2002, referring to his algorithm Spamsum) or the latter. "Fuzzy hashing" is not a generic trademark similar to Frisbee, Thermos or Discman — and thus it is often used to refer to any hash function with similar capabilities as Spamsum. Typically, locality-sensitive hashing refers to hashing used to find nearest-neighbors in high-dimensional data, with hash functions that changes in accordance with the input. [1]

The document *Fuzzy Hashing for Digital Forensic Investigators*, by Dustin Hurlbut, defines fuzzy hashing as [2]:

> "The fuzzy hash utility makes use of traditional hashing, but in pieces. The document is hashed in segments depending on the size of the document. These segments will contain fragments of traditional hashes joined together for comparative purposes. Before this segmented hash can be done, a rolling hash is used to begin the process."

There is a significant overlap between the two definitions, and fuzzy fingerprinting was the term used in the project description published on It's Learning. Thus, fuzzy hashing (and fuzzy fingerprinting) will be used to describe all algorithms in this report who are capable of changing just parts of the digest based on differences in the input — as an unbrella term — to avoid confusion.

> I say with Didacus Stella, a dwarf standing on the shoulders of a giant may see farther than a giant himself.
>
> Robert Burton, 1621

Chapter II
Literature Review

This literature Review chapter will feature the material that was studied when starting this project, in terms of hash functions and discrete mathematics. To avoid spending time on finding a complex solution to something that had already been solved in an easy manner, this material proved very useful.

This chapter has three main sections: one featuring prerequirements in discrete mathematics (can be skipped if the reader has good familiarity with this), one featuring background material (concerning hash functions and their attributes) and one explaining the need for two new approaches in a system based on plaintext and file classification and identification — useful in more advanced enterprise tools, such as Avito Loops.

1 Introduction to Mathematical Prerequirements

To better understand this report and the enclosed Java code — as well as how the algorithms and approaches in this report work, it is important with some knowledge about discrete mathematics and Boolean algebra — in addition to how this can be applied practically in code. In terms of algorithms and optimizations in general, mathematical skills are just as important as programming skills. This section assumes familiarity with Boolean operators, as well as an understanding of rudimentary discrete mathematics and overall mathematical skills on par with the curriculum of a master's degree graduate in computer science.

This section is included due to the fact that discrete mathematics are not mandatory in most colleges or universities. *Readers who are familiar with finite fields, Mersenne primes, and how Boolean operands can significantly improve performance* ***may skip this section****, but are encouraged to look at the subsection on pitfalls.*

1.1 Performance Improvements From Boolean Logic

Some arithmetic operations (most notably divisions and modulus operations) are computationally expensive if done frequently. Dividing numbers is well-known to be a costly

operation, even on modern hardware — and most compilers will still not do optimizations at this level.

Various operations and their Boolean equivalents ($m \wedge (n-1)$ and $m \mod n$ does exactly the same thing if n is a power of two) have been tested in the JUnit framework. Here, a for loop with 10 billion iterations was used. The results can be seen in the table below:

Table 2: Arithmetic operations compared to their Boolean equivalents

Function	Time Used
$m \mod n$	3s 471ms
$m \wedge (n-1)$	711ms
$m \times n$	696ms
m/n	3s 456ms
$m >>$	708ms
$m <<$	694ms

1.1.1 Avoidng the Modulus Operands — If Possible

As seen in Table 2, division is a costly operation on even a modern CPU. Since the modulus operand in most compilers is not compiled into faster Boolean operations, this is therefore something that should be used by caution. [3] In the x86 architecture, the integer division will also give the remainder after the division in a different register — regardless of whether the remainder is needed or not.

Fortunately, a Boolean AND is computationally very cheap. If a number n is a power of two, one can find the remainder after a division by simply doing an AND operation with $n-1$. As an example:

$$99 \mod 8 = 3 \tag{1}$$

$$99 \wedge 7 = 3 \tag{2}$$

This principle is technically true for all cases where the divisor is a power of two. All classes in this thesis are implemented without the modulus operand and without division where this is possible. The reasons for this will be even more obvious in Chapter III, where the designs are shown.

1.1.2 Benefits of Using Bitwise Shifts

Federal Standard 1037C defines arithmetic shifts the following way:

> "A shift, applied to the representation of a number in a fixed radix numeration system and in a fixed-point representation system, and in which only the characters representing the fixed-point part of the number are moved. An arithmetic shift is usually equivalent to multiplying the number by a positive or a negative integral power of the radix, except for the effect of any rounding; compare the logical shift with the arithmetic shift, especially in the case of floating-point representation." [4]

While multiplications are not very expensive on the x86 architecture, divisions are. A bitwise shift of one to the left or right (denoted by << and >> in Java), means moving all the integers in a binary number to either left or right, followed by a leading or a trailing zero, depending on the direction. The number 111 shifted two times to the left, will for example become 11100, while 101 shifted to the right will become 010. Clearly, this indicates that a shift to the left of m numbers is the same as multiplying by 2^m, and doing a shift to the left of m numbers is the same as dividing it by 2^m, while discarding the remainder.

1.1.3 Mersenne Primes

By definition, a *Mersenne number* is a number on the form $2^n - 1$ [5]. This means that any integer that is a power of two, becomes a Mersenne number if it is decremented by one. More specifically, all Mersenne numbers can be represented as binary numbers with no zeroes, except for leading zeroes, meaning that they are repunits. Examples of Mersenne numbers are 7, 255, 8191, who can be represented in binary numbers as 111_2, 11111111_2 and 1111111111111_2 respectively. This also illustrates why hexadecimal numbers are useful — as these are very easy to convert to binary form.

A Mersenne prime is simply a Mersenne number that is also a prime. The following numbers are known Mersenne primes that will fit into a signed long integer or an unsigned 32 bit integer: $2^2 - 1$, $2^3 - 1$, $2^5 - 1$, $2^7 - 1$, $2^{13} - 1$, $2^{17} - 1$, $2^{19} - 1$ and $2^{31} - 1$ [6]. An interesting feature about Mersenne primes, is that the modulus of any number divided by a Mersenne prime can be found the following way:

$$m \wedge p + m >> s \tag{3}$$

Where m is any number larger than p — and s is the bitnumber for $p + 1$. Since this is a prime, there will always be a remainder as long as the prime is not a factor of the dividend, making it a useful feature in one-way functions. This theorem will serve as a basis for NFHash, which will be explained in detail in Chapter III.

1.2 Finite Field Arithmetics

A Galois Field (also called a Finite Field) refers to a polynomial with a finite set of elements. These are on the form $GF(p^k)$ (there are several ways to write this, such as

Z_{p^k}, F_{p^k}, etc.), where p is a prime and k is an integer > 0. If the field is binary ($GF(2)$), then the only coefficients are either zero or one. While this may not seem intuitive at first, the exponent m of each X in the polynomial will be treated by a computer as a 1 with m trailing numbers. This section will only focus on binary fields, but will deal with arbritary degrees. The degree of the largest element determines the degree of the polynomial. A polynomial $p(X) = X^4 + X + 1$, will for example have a degree of four; as will a polynomial $P(X) = X^4 + X^3 + X^2 + X$ — and a polynomial consisting of a single X^4.

Binary Galois fields have experienced a widespread use in RSA, CRCs, Reed-Solomon, Elliptic Curve Cryptography and **Rabin's Fingerprinting Scheme**, to mention a few.

Table 3: Example of how a binary Galois field with a degree of 6 works

Possible Values	X^6	X^5	X^4	X^3	X^2	X^1	X^0
Bit String Representation	1	0	0	1	1	0	1
On/Off	*On*	*Off*	*Off*	*On*	*On*	*Off*	*On*
P(X)	$X^6+X^3+X^2+1$						
Decimal	77						
Hexadecimal	4D						

Table 3 shows the basic intuition between so-called binary Galois fields; that is Galois fields on the form $GF(2)$. In Rabin's fingerprinting scheme, the polynomials should be irreducible over $GF(2)$ — i.e. the smallest finite field. This in turn means that it should be irriducible modulo 2.

1.2.1 Arithmetic Operations

One interesting feature about finite field arithmetics over $GF(2)$, is that addition, subtraction and exclusive OR operations are identical. The two former are done with a modulo of 2, making them similar to binary number arithmetics, except without a carryover. This means that $X + X = 0$ and that $-X = X$; likewise $X^2 + X^2 = 0$ and $X^4 - X = X^4 + X$. The polynomial $(X^5 + X^4 + 1) \pm (X^4 + X^3 + X)$ is therefore equal to $(X^5 + X^3 + X + 1)$ — not $(X^5 + 2X^4 + X^3 + X + 1)$ or $(X^5 - X^3 - X + 1)$ like a "typical" polynomial would be.

Table 4: Addition and subtraction in $GF(2)$ (top) can be considered identical to an XOR operation. Multiplication (bottom) can be considered identical to an AND operation

±	0	X
0	0	X
X	X	0

×	0	X
0	0	0
X	0	X

The intuition behind multiplication is quite simple, although it requires a little more work than addition or subtraction. One good way to do this, is to simply multiply the $GF(2)$ polynomials as if they were conventional polynomials, and then perform a reduction modulo 2 (essentially, delete any X that has a constant which is a power of two). A step-by-step guide can be seen below:

1. Begin with two or more $GF(2)$ polynomials: $(X^5 + X^2 + X)(X^4 + X^3 + X + 1)$
2. Multiply the two GF(2) polynomials as if they were ordinary polynomials: $(X^9 + X^8 + 2X^6 + 3X^5 + X^4 + X^3 + 2X^2 + X)$
3. Perform a reduction modulo 2, which will give the result: $(X^9 + X^8 + X^5 + X^4 + X^3 + X)$. Remember that the same rule as in addition also apply in this final step

Division in the scope of $GF(2)$ is for the most part irrelevant although it can be done in conventional long division or by using the *Newton-Raphson* method. Since binary numbers can be represented as decimal and hex numbers, using the builtin bitwise operations in Java makes more sense. Netwon-Raphson will not be covered here, as division of polynomials will not be used *directly* in this project.

1.2.2 Irreducibility and a Probabilistic Algorithm for Testing This

A polynomial is irreducible if it cannot be factored into nontrivial solutions over a given field. [7] For this project, this given field is $GF(2)$. Remember that the same multiplication rule also applies when factoring over $GF(2)$. If none of the polynomials in GF(2) of a given degree divides a given polynomial $f(x)$, it is irreducible over that field.

$x^2 + x + 1$ is irreducible, since it cannot be factored into any nontrivial solution. $x^2 + 1$ is not irreducible, as it can be factored into a nontrivial solution. An example of how to check if a polynomial is irreducible over $GF(2)$:

$$(x + 1)(x + 1) = x^2 + 2x + 1 \equiv x^2 + 1 \ (mod\ 2) \tag{4}$$

Table 5: Examples of irreducible polynomials of various degrees over GF(2)

Degree (n)	**Irreducible polynomials**
1	$1 + x$, x
2	$1 + x + x^2$
3	$1 + x + x^3, 1 + x^2 + x^3$
4	$1 + x + x^4$, $1 + x + x^2 + x^3 + x^4$, $1 + x^3 + x^4$

The explanation behind this is that over $GF(2)$, $x^2 + x + 1 = (x + 1)(x + 1)$. x^2 leaves a remainder when divided by two; as does 1. $2x$ will, on the other hand, not leave a remainder. If the remainder on the other hand is zero after factorization and division, the polynomial is not irreducible over $GF(2)$.

Rabin's test of irreducibility is very similar to *Distinct-degree factorization* (it tests every degree smaller than or equal to half the degree of the input polynomial). The fraction of irreducible polynomials over a finite field of degree n, can be roughly estimated as $\frac{1}{n}$. Rabin's test of irreducibility is based on the fact that a polynomial $C \in GF(p)$ of degree n is irreducible if and only if $x^{p^d} \equiv x \, mod \, C$ (d refers to the degrees being tested and not the degree of the polynomial).

The numbers are tested with an irreducibility test to verify that they cannot be reduced. Rabin's test of irreducibility can be explained as follows [8], where n is degree, p is the number of elements (in this case 2), q is power (in this case 1), $p_1, ..., p_k$ is every prime divisor of n:

Data: A random polynomial f of degree n
Result: Either a logical true if polynomial is irreducible—or a logical false if it is reducible

```
for j = 1 to k do
    for i = 1 to k do
        int h = x^(x^(n_i)) − x  mod  f; int g = gcd(f, h);
        if g ≠ 1 then
            return false;
        end
    end
    int g = x^(q^n) − x  mod  f
    if g=0 then
        return true;
    end
    else
        return false;
    end
end
```

Algorithm 1: Rabin's Test of Irreducibility

This algorithm cannot tell with complete certainty is a polynomial is irreducible, but it can still tell if it is *probably* irreducible. In other words, this is a *probabilistic algorithm.*

1.2.3 Implementing polynomials Over a Galois field

Since a computer processes information in a different manner than the human brain, *the best way to implement these is by simply implement them with a hexadecimal number*, as seen in table 3. The advantages of using a hex number are as follows:

1. There is no overhead due to using a datastructure
2. It saves space. While a two digit decimal number can represent anything from 0 to 99, a hexadecimal number can represent any number from 0 to 255 (denoted as FF or $0xFF$ in hex form) using no more space
3. The number 16 is the same as 2^4. This makes it computationally easier to convert it to a binary number, which is used on the lowest level by a CPU

Because of this, this project will simply represent them as hexadecimal numbers. This section is also meant to bridge the gap between mathematical prerequisites on paper and mathematical prerequisites in the form of C-like code.

Polynomials can also be represented by datastructures or binary strings, but since this adds a lot of overhead for arithmetic operations (not to mention that it will significantly lead to more memory being used), it will not be employed in this project.

1.3 Pitfalls

Mersenne Primes

Not all Mersenne numbers are Mersenne primes, and thus, not all Mersenne numbers will leave a remainder just like that. Furthermore, it is important to note that just because p is a prime does not necessarily mean that $2^p - 1$ is a prime. On the other hand, if $2^p - 1$ is a prime, it means that p is also a prime. A total of 8 Mersenne primes can be represented by a long integer.

Galois Fields

1. The bit string representation cannot be converted to a Galois field as if it were a binary number. Instead of converting 10010 to $X^{16} + X$, it has to be converted to $X^4 + X$; i.e. the power of X depends on the position from right, starting from 0
2. "Keep it simple, stupid". An integer or a long integer as a hexadecimal or decimal number is much easier on the computer hardware than an advanced datastructure – and will result in fewer lines of code
3. The fact that a polynomial cannot be factored into roots does not mean that it is irreducible.

Galois Fields have several properties in common with real numbers. These are:

- Addition has an identity element (which is 0) and an inverse for every element
- Multiplication also has an identity element (which is 1) and an inverse for every element except 0
- Multiplication takes precedence over addition
- Addition and multiplication are both commutative and associative

There are also two noticable dissimmilarities, that are important to acknowledge:

- $-x = x \iff x + x = 0$
- $x \times x = x$(the exponent here only determines this position in the binary representation)

2 Background

Hash functions are examples of applications of discrete mathematics and Boolean logic. As such, it makes sense to explain this after a brief explanation of mathematical prerequisites. Hash functions are essentially what this entire project boils down to — but

not the hash functions most readers will be familiar with (hash tables and cryptographical hash functions are typically what is part of the curriculum for a master's degree in computer science).

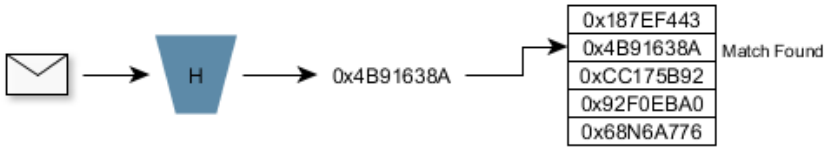
2.1 Brief Introduction to Hash Functions

A hash function (in the general sense) refers to a one-way function which compresses an input of arbritary length to an output of fixed size. By definition, these are one-way, in the sense that undoing the output is computationally infeasible, due to the fact that information becomes lost in the process. Easy in this context means polynomial as a function of input. [9] The entire process can be seen as analogous to what a paper shredder does.

Figure 1: Just like it is difficult and tedious to reassemble the shredded paper into a complete document, it is also computationally infeasible to try to decipher a hashed output. (This picture is public domain)

The output (refered to at the digest in this report — or sometimes the fingerprint when it is used to fingerprint content) from a hash function is not deciphered per se. Rather, an input is scrambled — and if something needs authentication, the hashed output is compared to other hashed outputs. This is because of the fact that a hash function is deterministic; i.e. if the input and the settings are the same, the output will also be the same. [10] The speed, efficiency and one-way properties of hash functions, mean that they have many uses. An example (one of the most well-known uses) is password authentication, where *cryptographic* hash functions such as SHA-2 can be used to see if the password entered matches any existing passwords on the computer. This is for instance used when loging into a computer, or when starting a car with an immobilizer. [10] Hash functions also have their place in terms of generating passwords from weak sources (eg. Bcrypt and Scrypt) – and when generating random (for example when hashing sampling noise) or pseudorandom numbers.

Figure 2: A hash functions scrambles an input and then checks for a match among other scrambled outpuds.



This project will focus on hash-functions used for fingerprinting files and fingerprinting plain text in the form of unformatted tables. Each file hashed, will have an output that is fixed for a given set of settings within a given algorithm. Just like you can identify any person and his wheareabouts based on fingerprint matches (without reversing the fingerprint and creating a new person), you can also do the same thing for any file and text table.

While data fingerprinting itself is nothing new, it has only become a popular subject during the past few years. The most well-known algorithm designed for this is **Rabin's Fingerprinting Scheme** (which will be cross-referenced a lot in this and subsequent sections). This is an open and widely-used standard for fingerprinting, first presented by the mathematician Michael Rabin in 1981. [11] Rabin's Fingerprinting Scheme can be considered a more accurate checksum. It's main virtue is that the probability of collision is very low (courtesy of irreducible polynomials over GF(2)) while at the same time offering very good performance.

Fuzzy fingerprinting (or fuzzy hashing) is a different form of hashing that can be used for similarity search and classifications, and is most commonly used in spam detection filters and forensics applications. After the release of the forensics tool *ssdeep* in 2006, **fuzzy hashing** has gotten more attention. This means that the so-called avalance effect, which causes a significant part of the digest to change if only just one bit from the input changes, is scrapped in favour of detecting local changes in a file. Just like twins will have similar, but not identical fingerprints, two related files might have similar, but not identical digests. Fuzzy hashing will be explored in later sections, but the difference between fuzzy fingerprinting and ordinary fingerprinting can be seen as analogous to the difference between the *mean of a function* and the *running mean of the same function*.

In some cases, it might be beneficial to know what was changed, rather than just how much it was changed. An example might be which row or which column was changed in a text table. Here, **Merkle Trees** will offer a lot of possibilities, and these will be explored extensively later on in the report.

2.1.1 Collision Resistance and Other Security Meassures

Since a hash function authenticates something by comparing outputs rather than by deciphering, avoiding collisions becomes important. Otherwise, the system might accidentally think that two different files or two different pieces from a table are the same [12], which can become troublesome when merging files in a database query, when scanning for viruses or in other file applications where hashing is used. Preimage resistance and pseudorandomness are of lesser importance in terms of fingerprinting — since fingerprinting algorithms should not be used for sensitive information, except internally in an already secure environment. A list of requirements for a *secure* hash function can be seen below [13]:

Table 6: Security requirements for a secure hashing scheme

Requirement	**Description**
Variable input size	*An input x can be of arbritary size*
Fixed output size	*An output $H(x)$ is compressed to a fixed size*
Efficiency	*$H(x)$ is computationally easy to compute*
Preimage resistance	*$H(x)$ cannot be reversed efficiently back to the input x*
Weak collision resistance	*Also known as second preimage resistance. For any given block x, it is computationally infeasible to find $H(y) = H(x)$ if $x \neq y$*
Strong collision resistance	*For any given pair of two inputs (x, y), it is computationally infeasible to find two hashed outputs $H(x) = H(y)$*
Pseudorandomness	*The ouput $H(x)$ meets the standard tests for pseudorandomness*

A crude example of a one-way function is finding the remainder after dividing a number by a prime and doing an exclusive or with a number after that. As seen below, two different inputs can indeed produce the same output, which is very unfortunate when authenticating something.

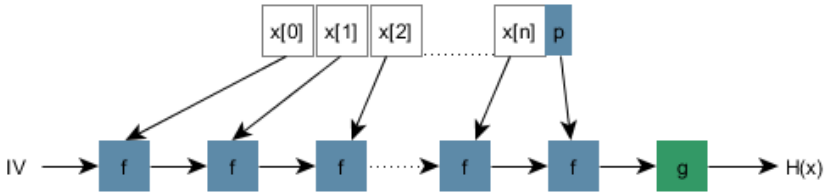
$$211 \mod 101 = 9 \tag{5}$$
$$9 \oplus 4 = 13$$
$$919 \mod 101 = 10 \tag{6}$$
$$10 \oplus 7 = 13$$

This example consists of two equations. In both equations, an arbritary number is divided by 101 — before the remainder is XORed with another arbritary number and then returned. In both cases, the number returned is 13, despite the fact that the

arbritary numbers used in both equations are completely different. Two less "artificial" and more realistic examples are MD5 and CRC32, which will be explained in detail later in this section.

An important aspect of hashing (even moreso in terms of fingerprinting than in terms of cryptographic hashing) is the **Pigeonhole Principle**. This states that if there are n n pigeons and m holes, one hole must necessarily contain more than one pigeon if $n > m$. [14]

Figure 3: How the Merkle-Damgård construction operates. Here $x[i]$ refers to blocks from the input message, f refers to the compression function, and g refers to the finalization function. IV refers to the initialization vector, while p is the padding



If this principle is extended, it can be proven that at least one hole will contain at least $\lceil \frac{n}{m} \rceil$ pigeons. In the same manner, if we have n possible preimages and a bit size of m, clearly, a collision may happen if $n > m$. Bit size in a hashing algorithm (whether cryptographical or simply a fingerprinting algorithm) is therefore crucial.

Hashing algorithms use a fixed size output — and a 5 MB (5242880 bits) input will in this case be mapped to a much smaller digest. There are $2^{5242880}$ ways to assemble a 5 MB file, but only 2^{512} possible outputs for a SHA-512 digest. Clearly, this indicates that there are a lot of possible collisions. However, *it does not by any means mean that a user is likely to find such a collision — whether it is deliberately or accidentaly.* Perfect collision resistance is impossible; rather, it is more important to make it difficult to find these collisions. Since there is no theoretical upper limit to file sizes (ext4 has a maximum size of 16 TiB), the number of possible collisions for a given digest must also necessarily be infinite.

Many of the strong security capabilities of a cryptographic hash function are achieved through the Merkle-Damgård construction. This splits an input file into smaller blocks, hashes each block individually, and then runs a finalizing hash on all the hashed blocks. [15] While this is what gives (or gave, in the case of the two latter) SHA-2, SHA-1, and MD5 their security, it is also what makes these algorithms slower than a typical fingerprinting scheme.

Rabin's fingerprinting algorithm (which will be presented in detail shortly) only guarantees a very low probability of collisions, but cannot guarantee that these will never

happen. The probability of two distinct strings colliding here is:

$$p \leq \frac{n \times m^2}{2^k} \tag{7}$$

where n is the number of unique strings, m is the string length and k is the bit length of the fingerprint. [16] Since collisions are statistically independent occurences, Michael O. Rabin suggests using two irreducible polynomials.

For fingerprinting algorithms that are being used internally (where security against malicious attacks are not important) a digest size of 64 bits is enough. An example where this will be used (as seen later on in the project) is in terms of hashing rows and columns. A 64bit number size will still fit into a long integer, and will still offer a collision probability of $\frac{1}{2^{64}}$ if the algorithm is perfectly balanced, since a bit can be eiether 0 or 1 (two possible states) and there are 64 bits in total.

The increased performance of a fingerprinting scheme compared to a cryptographic hash function also means that it is easier to deliberately find a collision (better performance also means that more candidates can be tested in a given amount of time) in these for malicious purposes, which is another reason why they should never be used for security purposes. Note that a user is still very unlikely to stumble upon a collision by accident.

2.1.2 The Odds Put Into Perspective

Assuming a perfectly balanced hashing scheme, the probability of generating two unique fingerprints is [17]:

$$p = \frac{N-1}{N} \tag{8}$$

Where N is the digest size (for a 64 bit hashing scheme, $N = 2^{64}$) and k is the number of digests. For multiple numbers, due to statistical independence, this formula can be modified into:

$$p = \prod_{i=0}^{k-1} \frac{N-i}{N} \tag{9}$$

This is tedious to calculate both for humans and computers, and instead, it can be approximated as:

$$p = e^{\frac{-k(k-1)}{2N}} \tag{10}$$

Reformulated, this can also be used to calculate the risk for one or more collisions, so that the formula becomes

$$p_{collision} = 1 - e^{\frac{-k(k-1)}{2N}} \tag{11}$$

The problem described here can be formalized as the **birthday paradox**, and deals with the probability of two out of m people have the same birthday. [18] Since there are only 365 days in a year, the odds of two people sharing the same birthday is thus very high. If inserted into the formula above $((1 - \frac{1}{365}) \times (1 - \frac{2}{365} \times ... \times (1 - \frac{m-1}{365})))$, it can be shown that if there are at least 23 people at a social gathering, in a school class or in a workplace, there is more than 50% chance of two people sharing their birthday. Table 7 shows the collision risk for different fingerprint sizes depending on the number of fingerprints.

Table 7: Collisions risks and fingerprint sizes

No. 32-bit values	No. 64-bit values	Odds of a Collision
77163	5.06 billion	0.5
30084	1.97 billion	0.1
9292	609 million	0.01
2932	192 million	0.001
927	60.7 million	0.0001
294	19.2 million	0.00001
93	6.07 million	0.000001
30	1.92 million	0.0000001
10	607401	0.00000001
	192077	0.000000001
	60740	0.0000000001
	19208	0.00000000001
	6074	0.000000000001
	1921	0.0000000000001
	608	0.00000000000001

As seen from the table, a 32 bit fingerprint is clearly insufficient when fingerprinting tables on a row-by-row or column-by-column basis (as the Merkle Trees in the next chapter will do).

2.2 Currently Existing Fingerprinting Schemes

Universal hashing algorithms, checksums, and the most commonly used cryptographic functions are all keyless hash functions, but that is where the similarities end.

In theory, any hash function can be used for fingerprinting files, large chunks of text and so on. Cryptographic hash functions are powerful, but might seem a bit "overkill" in terms of fingerprinting, especially given their performance. There are many lightweight hash functions unsuitable for cryptographic purposes that still have potential in terms of identifying data. The simplest hash function for this is probably **Fowler-Noll-Vo**, which is used in combination with Adler's checksum for the **Spamsum** algorithm. [19] It's main virtues are two things: it's performance, and it's ease of implementation [20].

FNV-1a (pseudocode)

```
hash = FNV_offset_basis
  for each byte_of_data to be hashed
       hash = hash XOR byte_of_data
       hash = hash * FNV_prime
  return hash
```

There is no free lunch, however. The fact that the algorithm revolves around multiplications and EXCLUSIVE OR operations, means that it is sensitive to zeroes. [20] If the hash value of one round is zero, then the hash value of the next round will be XORed with zero, and as such, the first line inside the for loop will make no difference whatsoever. In FNV-1, XOR and multiplication happens in the opposite order of FNV-1a. The latter will have much better avalanche capabilities than the former, while using no more power. FNV-1a will still have the same bias towards zeroes, and as such, it will lead to a high collision risk if the numbers are not very large.

Two better suited alternatives are 64 bit **cyclic redundancy check** and **Rabin's Fingerprinting Scheme**.

Mathematical Basis of Rabin's Fingerprinting Scheme

Rabin's Fingerprinting Scheme is a hashing algorithm owing it's collision resistance to the use of irreducible polynomials over $GF(2)$. This was first presented by Michael O. Rabin in 1981. While any hash function can technically be used to fingerprint files or tables, this is the most well-known algorithm that was designed specifically to do so.

If random polynomials are used, it can be considered a universal hashing scheme. Generally speaking, Rabin's Fingerprinting Scheme can be considered a more accurate and a more large-scale checksum. While most checksums (for example Adler's Checksum) do not have to be accurate (it is enough that a small file or a message that is corrupted differs from the original file or message), Rabin's Fingerprinting Scheme was designed to be robust against collisions.

Suppose $A = (a_1, a_2, ..., a_m)$ is a binary string, which in turn can be represented as a polynomial $A(t)$ of degree $m - 1$. Now assume that $P(t)$ is an irreducible (over $GF(2)$) polynomial of degree k, over the field $GF(2)$. Note that $A(t)$ does not have to be irreducible.

With this in mind, a fingerprint according to Rabin's Fingerprinting Scheme can be found [16]:

$$f(A) = A(t) \mod P(t) \tag{12}$$

As an example, let $P(t) = 1 + x + x^4 = 10011$ and let $A(t) = x^4 + x^3 + x + 1 = 11011$. These are polynomials of degree 4. In finite field arithmetics, the modulus is calculated

the same way as it would have been with ordinary binary numbers. The fingerprint therefore becomes $f(x) = x^4 + x^3 + 1 \; mod \; (x^4 + x + 1) = x^3 + 1$=1001. Naturally, the numbers used are typically much larger than five bits in practical applications. A straightfoward, pen and paper implementation of Rabin's fingerprinting is still slow on modern hardware. Therefore, Andrei Z. Broder's implementation is used in this project (which among others uses a lookup table). This implementation is described in detail in chapter III and IV. In Andrei Z. Broder's impementation (which is most commonly used), an input is not hashed byte-by-byte, but by splitting an input into fixed chunks of four bytes. Each chunk is hashed and ANDed with a mask, before being XORed with previous chunks.

Due to excellent performance, good collision resistance, and the ability to handle negative numbers (Java only has signed long integers — not unsigned long integers), Rabin's Fingerprinting Scheme was an obvious candidate.

The Workings of CRC

A **Cyclic Redundancy Check** is a checksum type and a type of error-detection codes. The difference between CRC and an ordinary checksum is that CRC is based on the remainder (redundancy) after cyclic polynomial divisions, hence the name. This makes CRC more accurate than for instance Fletcher's or Adler's Checksum, while only making the CRC algorithms a little slower to calculate.

A CRC can have many sizes. The number usually refers to the order of the polynomial used and not the bit size (CRC-32 is actually 33 bits, for instance).

Java's CRC-32 implementation is specified in RFC-1952 [21], except that the Java implementation does not use lookup tables. Due to it's small digest size, CRC-32 has a very high collision risk (as seen in table 7), and thus, CRC-64 is of greater interest.

NFHash (presented later on) will be using CRC64 as a rolling hash. A CRC needs a generator (divisor) polynomial to work as a divisor in polynomial long division. In NFHash, the polynomial $x^{64} + x^4 + x^3 + x + 1$ (the smallest 64 bit generator) as specified in the *ISO-3309* standard [22], will be used as a divisor.

The formula for CRC can be summarized the following way:

$$R(x) = M(x) \times x^n \mod G(x) \tag{13}$$

Here, $R(x)$ is the remainder, while $M(x)$ is the input. $G(x)$ is the generator (divisor). The input string is padded with n bits, represented by x^n, where n refers to the order of the polynomial.

Generally, a rudimentary CRC is similar to Rabin's Fingerprinting Scheme at first glimpse (both can be considered more powerful checksums), apart from the fact that Rabin's Fingerprinting Scheme is not cyclic. In CRC, the divisor will "slide" underneath the dividend, so that the trailing 1 in the divisor will align with the nearest 1 in the

dividend. In other words, it does not necessarily move bit-by-bit. An example can be seen below:

Table 8: How a Cyclic Redundancy Check Operates (for illustrative purposes, the four padded bits are not included)

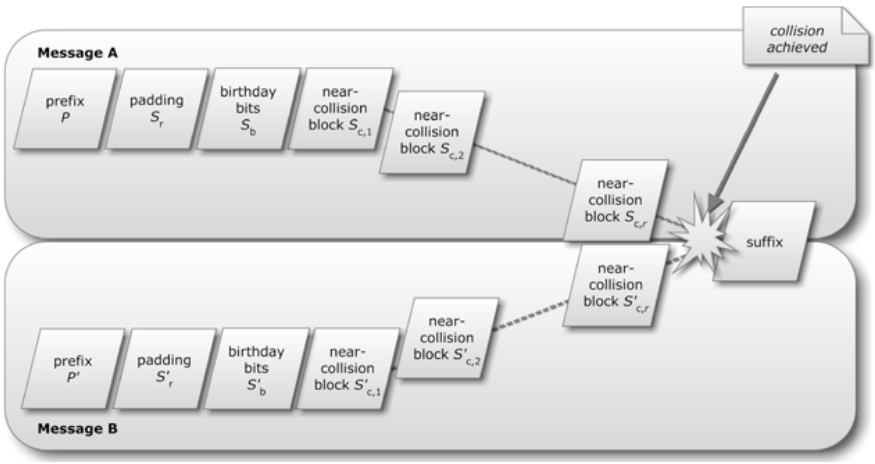
		Remainder
Dividend	101000110000	
Divisor	11001	0
Dividend	011010110000	
Divisor	11001	0
Dividend	000011110000	
Divisor	11001	0
Dividend	000000111000	
Divisor	11001	1010

From the table, it can be seen that the process ends when the number that is left is smaller than the generator. If this number is zero, then there is no remainder, but in this case the remainder is 1010_2.

2.2.1 Current Status of MD5

MD5 is probably still the most used algorithm for fingerprinting large archives and ISO files. In MD5, accidentally stumbling across collisions is not likely. However, an adversary might put malicious content into a trusted file, without MD5 detecting it.

Figure 4: Chosen-prefix attacks explained (this picture is public domain)

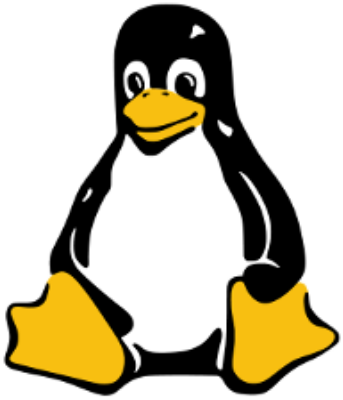


The main reason why MD5 can be considered "outdated" is because of so-called chosen-prefix collisions [23]. While this may look easy on paper and easy inside an artificial test

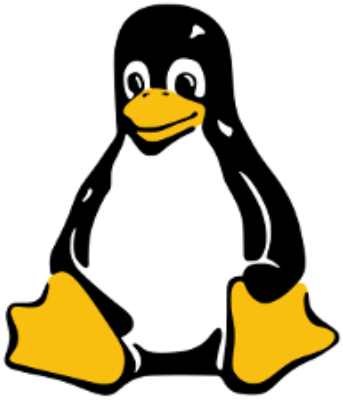
case, it is usually difficult on large archives, ISO files and so on in real life. The reason why MD5 was not used in this project, was mainly due to the fact that it is a lot slower than Rabin's Fingerprinting Scheme or CRC-64 — especially if it is going to be used as a rolling hash, or to hash each row and column of a plaintext table.

2.2.2 Fuzzy Fingerprinting Algorithms — a Unique Approach to Hashing

Figure 5: Two identical images, with two completely different SHA-256 digests, due to a few bytes differing in the file header. (This picture is public domain)



49DDA8EB2095331CD704DDBA69AC5BA
300BE748995637685FD205E58D093595C



80CC19B81145DA1BACFB983E34AEDC6
9BA0981B46EF641810C40B6F1A1F1790B

In figure 5, it can be seen that two identical .PNG files (the PNG format being lossless, without any artifacts or noise) will give two very different digests even if the headers are only differing by one byte (with the rest of the files being identical). This effect is called the Avalance effect. The Avalanche effect can be formalized as *The Strict Avalanche Criterion (SAC)* [24]. SAC states that if one bit in the input changes, each output bit will change with a probability of 50% or more. This will in turn lead to half or more of the output bits being changed. The Avalanche Effect is extremely important in terms of cryptography, but in terms of fingerprinting, it is less important. When detecting if a file was changed or not (and the extent which it was changed) is the subject, it might even become an obstacle.

Sometimes it might be useful to only let small changes in the input lead to small changes in the output. Reasons for this might be if a user wants to match files that are similar, but not identical (or files with identical content but different headers) — or discover if a certain file is a newer or an obsolete version of another file. On a different level, it might also be used to discover morphable malware, detecting spam or detecting plagiarism. This concept is called **fuzzy fingerprinting** or **fuzzy hashing**. As seen in figure 5,

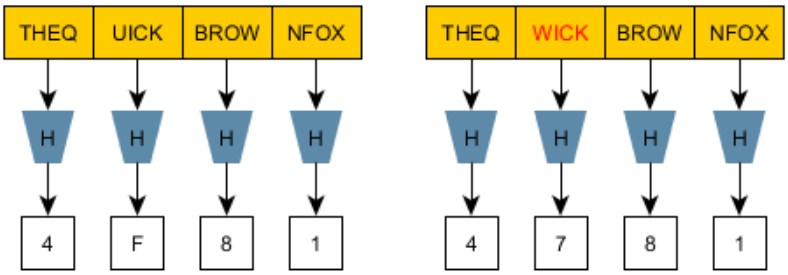
most conventional hashing algorithms are useless for identifying nearly-similar content (although nothing will beat them when it comes to matching identical content).

Two currently existing fuzzy fingerprinting schemes will be referenced a lot in this report. These are called **Nilsimsa** (often referred to as a form of locality-sensitive hashing) and **Spamsum** — and were designed to detect spam. While both are fuzzy hashing schemes utilizing a sliding window, both are neverheless very different from each other. Both will also be compared to **NFHash** in the next chapter. While these were not the only ones that were investigated, they were the ones who were the most similar to the scope of this project — and thus, both were major sources of inspiration. Moreover, the fact that these algorithms are very different, yet serve the same purpose, means that there are many ways to solve a problem.

Rabin-Karp (not the same as Rabin's Fingerprinting Scheme), SAHash and rsync were also studied, since these are other examples of rolling hashes. They were not studied in the same degree as Nilsimsa or Spamsum, however. Rabin-Karp is a plagiarism-detection algorithm and is not used for classification, and rsync is what Spamsum is derived from [25]. SAHash is — on the other hand — somewhat similar to Spamsum, but due to time constraints, this and TarsosLSH (more closely related to Nilsimsa) were not studied in detail.

Chunkwise Hashing — A Cheap Way to Do Fuzzy Fingerprinting

Figure 6: A naive chunk-wise hashing approach might sometimes give good results on two equally long inputs

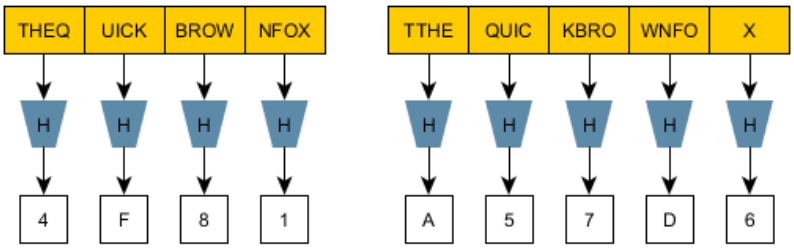


When using a fingerprinting scheme with a low overhead (eg. Rabin's Fingerprinting Scheme with a lookup table), the marginal costs of generating yet another digest are very low. Therefore, at first glimpse, it might make sense to simply cut an input into several fixed size chunks, and hash each chunk individually.

In the picture above, only one byte will change, due to the fact that the changes are confined to a single block. In the picture below, all the blocks change due to a shift, and thus, all digests will change — even though they are hashed individually. Despite the fact that the two strings only have one byte differing, they have no common characters in the digest — even though it is technically a fuzzy hash. This is a significant drawback,

considering that it has no advantage over a traditional fingerprinting scheme, while at the same time being a little slower and more cumbersome.

Figure 7: A naive chunkwise hashing approach will not give good results if there is a length disparity (even by one character)

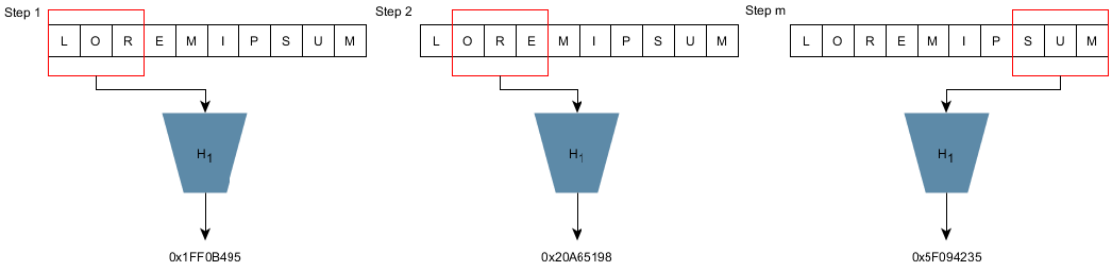


Rolling Hashes — Doing Fuzzy Hashes More Efficiently

To make a fuzzy hashing scheme that is insertion robust [26] (i.e. it will resynchronize), a sliding window is needed. When using a sliding window, a fuzzy hashing scheme can utilize a rolling hash. Typically, this rolling hash is used in combination with another hash function, such as FNV in Spamsum, but this is not a strict requirement.

A rolling hash produces a digest for every position in the file, and will only use the last few bytes of each digest — since it will hash byte-by-byte. The size of the sliding window will determine the size of the current input. A smaller input makes the result more dependent on details; if it is too small, even very similar files will have a somewhat different digest; if it is too large, files that are maybe 80% similar will have an identical digest. Sliding window sizes are also crucial in classifications and machine learning, which will be discussed in **Chapter III**.

Figure 8: A rolling hash will utilize a sliding window to hash each byte individually. If one of these digests matches or exceeds the predefined block boundariy, a block boundary is defined.



The main purpose of the rolling hash is to find block boundaries (or reset points, as they are referred to by Andrew Tridgell in the Spamsum source code). If the digest from the rolling hash matches a predefined block boundary number, a boundary is defined and a new block is started. If two hash functions are used, whenever a boundary is

reached [19], the content between the current and the previous boundary is hashed (in Spamsum, this is done by utilizing FNV).

In the case of Spamsum, this is done in two batches, calculating two halves of the digest sequentially. The right part of the digest will have a block boundary number that is twice as high as the left part. These block boundaries are also refered to as "block size" in Spamsum, even though they do not technically represent the block size itself.

Spamsum can utilize a fixed block boundary number or a dynamic block boundary number, which is calculated by iterating over the bytes before hashing and guessing an appropriate block boundariy number. Unfortunately, it may not always guess the right block boundary number, leading to a fixed block boundary number often being the better option. Another drawback of this approach is that small changes in the input text can cause the algorithm to choose a different block boundary number than previously, meaning that it can be difficult to compare two files unless several digests from the same file (with different block boundary numbers) are used.

As a rule of thumb (for all algorithms utilizing a rolling hash), the block boundary number can be adjusted to match the file size. For smaller files, small block boundaries are preferable; for large files, use large block boundaries.

What Nilsimsa Does Instead of a Rolling Hash

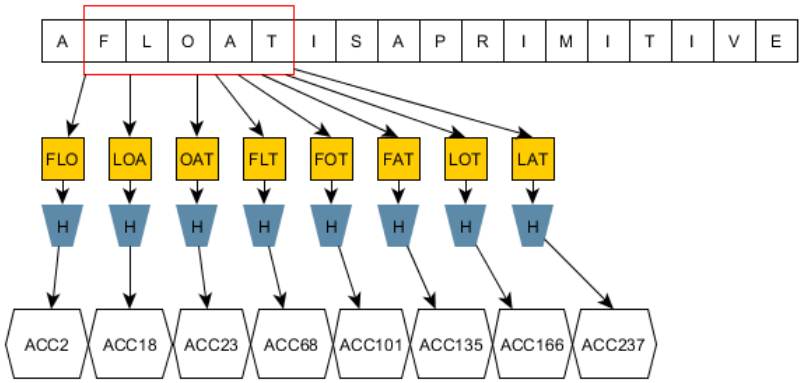
The other fuzzy hashing scheme studied in detail, Nilsimsa, also utilizes a sliding window. However, it does not employ a rolling hash layout. Instead, it uses a five character window size, and then computes all eight tri-grams that can be made from the characters inside the window. The word "float" for example has the trigrams: "flo", "loa", "oat", "flt", "fot", "fat", "lot", and "lat". When this is done, it utilizes the Trans53 hashing scheme and maps every trigram to one (for each) out of 256 accumulators. The accumulators that receive a trigram are then incremented. Finally, an expected number is calculated. [27]

In the next step, the accumulators are grouped in groups of four, so that they can represent binary numbers between 0 and 15. In total, there will be 64 groups, which in turn leads to a 64 bit digest. This approach is faster than Spamsum, but it is also less accurate. Moreover, it is more complex to implement.

It is also possible to expand Nilsimsa to Base-64, by utilizing 368 accumulators instead of 256. This will in turn lead to more memory being used.

Nilsimsa does not utilize block boundaries, and does not use a hashing scheme to slide the window. Therefore, it does not utilize a rolling hash.

Figure 9: Nilsimsa makes use of accumulators instead of taking the rolling hash approach.



3 Why Design two New Approaches?

Clearly, one of the most important aspects in designing algorithms is to never reinvent the wheel. Therefore, other existing fuzzy hashing schemes were studied carefully before **NFHash** was developed. **Nilsimsa** and **Spamsum** were chosen because spam detection is somewhat similar to matchig files, because both are well-known (for people who are familiar with fuzzy hashing algorithms), because both have different strengths and weaknesses, and because both use a completely different approach for reaching their goals. The fuzzy hashing scheme in this project seeks to achieve the performance of **Nilsimsa**, while achieving the accuracy and the robustness of **Spamsum**.

One important issue of Spamsum, is that it is relatively slow and produces very large fingerprints. Nilsimsa, on the other hand, has a higher character-wise collision risk than spamsum, and is quite rigid, in the sense that there is not much room for different settings without extensive modifications. Therefore, a new fuzzy hashing scheme named **NFHash**, which is purpose-designed for table and plaintext classifications, was created.

The second approach, by utilizing Merkle Trees, was chosen because of the fact that while a fuzzy hashing scheme can tell you *how much* a file has changed — it cannot tell directly where the change happened. Merkle Trees have already proven their usefulness in cryptocurrency [28], noSQL systems and file sharing tools to detect inconsistencies. Hence, it makes sense that they can also be used to detect updates or changes to a file. Moreover, Merkle Trees are quite resource efficient, given that no resources are wasted on comparing nodes that are the same in two or more trees.

In this project, the fuzzy hashing scheme will be coupled with k-nearest neighbor to classify files. Afterwards, the Merkle Tree-based approach is intended to continue the work, by finding out what causes the differences in each class. The two approaches here are also designed so that they can be used individually — without depending on the

other part. This is why there are two main implementations instead of one.

3.1 The Current Relevance of Non-Fuzzy Fingerprinting Schemes

In the same manner as public-key encryption did not make symmetric key encryption redundant, fuzzy hashing at it's present stage will not make ordinary fingerprinting (hashing) schemes redundant either. The main reason for this is performance. Even the fastest fuzzy hashing schemes are much slower than a typical secure hashing scheme, which in turn are much slower than Rabin's Fingerprinting Scheme and typical checksums.

4 Chapter Summary

This chapter first begins with the mathematical prerequisites required to understand how the algorithms work, namely Galois Fields and Mersenne Primes. The former are binary polynomials that can be represented as base2 numbers; the latter are any prime on the form $2^n - 1$.

Hash functions are also introduced here; most notably for data fingerprinting. Both ordinary fingerprinting approaches and fuzzy fingerprinting schemes are described. Any high-performance hashing scheme can be used to fingerprint files. With fuzzy fingerprinting it is a different story, due to the fact that the fingerprint has to change with the same degree as the input.

The next chapter will present two new ways for fingerprinting unformatted tables, but will first introduce the datastructures and algorithms (including the algorithm used for classifications) these approaches are dependent on.

> Simplicity is prerequisite for reliability.
>
> Edsger Dijkstra

Chapter III
Method and Design

This chapter will deal with the design of the two implementations, and the approaches used. The two first sections concern string-matching and dataset matching algorithms — as well as two datastructures, the third section explains machine learning, while the rest will deal directly with the design of this project (including how the string-matching/dataset matching algorithms and the datastructures are used). Readers familiar with string-matching algorithms, dataset matching algorithms, Bloom Filters, and Merkle Trees, can skip the two first sections.

Note that in-depth regression analysis will not be covered here; mostly due to lack of immediate relevance, but also due to time constraints.

1 Comparing Results After Hashing

Matching two string digests or two data sets essentially boils down to calculating how similar they are. There are many ways to do this, but some commonly used methods are **Levenshtein** (developed by Vladimir Levenshtein), **Damerau-Levenshtein** (improved by Ferederick J. Damerau), and **Jaccard** (developed by Paul Jaccard). The two former deal with the cost of transforming *string A* to *string B*, while the latter deals with the similarities between two finite sample sets.

Jaccard describes *Jaccard coefficients* (also refered to as Jaccard indices) and *Jaccard distances*. Instead of the discrete number of characters that differ, Jaccard is instead used to calculate similarity based on the binary (an element is either inside the set or it is not) difference between two sets. The Jaccard coefficient can be calculated as follows [29]:

$$S_{ab} = \frac{|a \cap b|}{|a \cup b|} \tag{14}$$

For calculating the Jaccard distance, simply subtract the coefficient from one, so that $D_{ab} = 1 - \frac{|a \cap b|}{|a \cup b|}$. The main intuition behind Jaccard is to find the number of elements

present in both datasets divided by all the elements (a AND b divided by a OR b) [30]. An example of a binary Jaccard approach can be seen below:

Table 9: A Simplified Example Showcasing the Jaccard Coefficient Between Java and C++

	Classes	**Garbage Collector**	**Compatible with C**	**Multiple Inheritance**
Java	Yes	Yes	Yes (JNI)	No
C++	Yes	No	Yes	Yes

Java can in the aforementioned example be represented as the bit string 1110, while C++ can be represented by the bit string 1011. This will in turn lead to both the Jaccard distance and the Jaccard coefficient between Java and C++ being 1, since the exact number of each element in both bit strings are the same. It does not take into consideration that the 1s and 0s are in different positions, leading to the fact that C++ and Java would be classified as identical in this example.

Because of the high risk of misclassifications, the Jaccard index will not be used for matching strings. The Jaccard algorithm will be revisited in the next chapter, where Merkle Trees are discussed, and used for matching trees based on similar nodes.

An algorithm better suited for comparing strings is *Levenshtein*. The Levenshtein distance (or edit distance) between two strings refers to how many insertions, deletions or substitutions that are required for transforming string a into string b [31]. The edit distance between "wall" and "wolf" will for instance be 2, since two substitutions (a and l are replaced by o and f, respectively) are required to transform the former into the latter. Levenshtein differs from hamming code in that it can handle shifts; while the hamming distance between "wall" and "wolf" is the same as the Levenshtein distance, "1A84BA39" and "A84BA391" has a Levenshtein distance of 2, but a hamming distance of 8, since Hamming distance only allows substitutions.

Table 10: Examples of How to Calculate Levenshtein Distance (the bottom table is the most efficient approach). Here, s means substitution, i means insertion, and d means deletion

C	**O**	**F**	**F**	**E**	**E**			
	s				*s*	*i*	*i*	Total=4
C	**A**	**F**	**F**	**E**	**I**	**N**	**E**	
C	**O**	**F**	**F**	**E**	**E**			
	s				*d*	*d*		Total=3
C	**A**	**F**	**F**	**E**	**I**	**N**	**E**	

Levenshtein takes insertions and deletions into consideration as well, and can handle strings of arbritary lengths. This means that the Levenshtein distance is always at least the length difference between the two strings — and always at most the length of the longest string. The Levenshtein distance between "Airplane and "plane" will for instance be 3, since the former is exactly three characters longer than the latter (it does not matter which characters are added). Levenshtein is generally quite robust, since

it can handle insertions and deletions (and thus shifts). Nevertheless, it cannot handle transpositions.

Table 11: Matrix representation of the most optimal solution in Table 10

		C	A	F	F	E	I	N	E
	0	1	2	3	4	5	6	7	8
C	1	0	1	2	3	4	5	6	7
O	2	1	1	2	3	4	5	6	7
F	3	2	2	1	2	3	4	5	6
F	4	3	3	2	1	2	3	4	5
E	5	4	4	3	2	1	2	3	4
E	6	5	5	4	3	2	3	4	3

An improved version of Levenshtein, which allows transpositions, is called **Damerau-Levenshtein** [32] (both fall under the umbrella term "edit distance"). While transpositions are not as important as insertions, substitutions and deletions in terms of digests, rolling hash functions nevertheless scramble the input on a character-by-character basis (every fingerprint is an x character string, made from x one character fingerprints), meaning that this extra ability is still somewhat useful, while not consuming significantly more resources. Note that most implementations of **Levenstein-Damerau** found online are in fact ordinary Levenshtein or Optimal String Alignment distance.

As an example, the edit distance between *"WINEISNOTANEMULATOR"* and *"WINEISNTOANEMULATOR"* will become 2 with ordinary Levenshtein, while it will become just 1 with *Damerau-Levenshtein*. This is because ordinary *Levenshtein* counts the swapping of "T" and "O" in "NOT" as two substitutions, while *Damerau-Levenshtein* will just see it as a single operation. Because of the aforementioned virtue, *Damerau-Levenshtein* is used in this project — since this will be useful with a more "order robust" hashing scheme. Moreover, since Damerau-Levenshtein is not fundamentally different from ordinary Levenshtein in terms of performance (both have a running time of $O(N \times M)$, where N is the length of the first string and M is the length of the second string) apart from one extra lookup and one extra calculation, there were really no significant disadvantages to employing it in this project.

Table 12: Levenshtein and how it stacks up against Damerau-Levenshtein

W	**I**	**N**	**E**	**I**	**S**	**N**	**O**	**T**	**A**	**N**	**E**	**M**	**U**	**L**	**A**	**T**	**O**	**R**	
							s	s											Total=2
W	**I**	**N**	**E**	**I**	**S**	**N**	**T**	**O**	**A**	**N**	**E**	**M**	**U**	**L**	**A**	**T**	**O**	**R**	
W	**I**	**N**	**E**	**I**	**S**	**N**	**O**	**T**	**A**	**N**	**E**	**M**	**U**	**L**	**A**	**T**	**O**	**R**	
							swap												Total=1
W	**I**	**N**	**E**	**I**	**S**	**N**	**T**	**O**	**A**	**N**	**E**	**M**	**U**	**L**	**A**	**T**	**O**	**R**	

2 Datastructures Being Introduced in this Chapter

This section will briefly introduce every datastructure used in this project that is not a part of the curriculum in most master's degrees. Later on in this chapter, their uses directly in this project will be described.

2.1 Introduction to Merkle Trees

With Rabin's Fingerprinting scheme, generating "yet another" fingerprint when everything else is initialized is computationally very cheap. Unfortunately, with chunk-wise hashing, digests will become very large and cumbersome to work on. If a table has 2000 rows and each row becomes a chunk, a potential 8 byte fingerprint per row will give a hash digest of 16 KB. For a table with 65,536 entries (the maximum number of rows in Open Office Scalc), the digest becomes one large 524 KB string — which is very cumbersome to store, use, and not to mention compare against another, equally long string. The solution to this problem is the Merkle Tree (named after it's creator, Ralph Merkle), which is a derivative of the binary tree. [33] Here, the hash value of each chunk — which for instance can represent each column or each row of a table — becomes a terminal node in the tree.

A Merkle Tree will utilize the fact that in an ordinary (non-fuzzy) hash function, the digest either changes completely or it does not change at all. Because of this, it has become a popular tool for detecting inconsistencies, regardless of how small or insignificant these might seem. Changes or updates to a table are not really any different from inconsistencies.

Merkle Trees are constructed from the bottom and up instead of from the top and down. Each terminal node is grouped pairwise under a parent node; these parent nodes are again grouped under other parent nodes, until there is only one node on top, namely the root node. Each parent node contains the hashed digest of the two digests below it [34] (i.e. the digests of the children become inputs for the hash function used to generate the parents' digests). In this project, a slightly different approach was taken, where $H(AB) = H(H(A) \oplus H(B))$. This is because $H(H(A) \oplus H(B)) = H(H(B) \oplus H(A))$ (i.e. if the sorting algorithm accidentally swaps the place of two children, it will not matter) and the fact that it is much more computationally easy to simply to an exclusive or operation than it is to first convert two long integers to strings and then merge them.

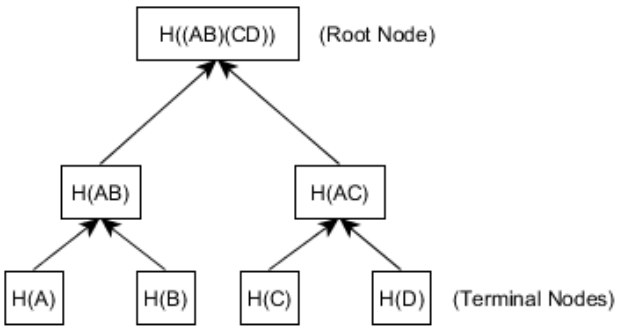
The datastructure is known for being both robust and simple, which also makes it versatile. The most famous use of Merkle Trees is probably in the scope of Bitcoins, but it is also being used by Git repositories (although not for detecting updates) and noSQL systems, such as Cassandra. Merkle Trees have very few loose parts and are fundamentally similar to LinkedLists and Binary Trees. Generally, insertion, deletion, traversal, and so on of a Tree structure is a very cheap affair — despite the fact that

a datastructure for nodes add a slight overhead. The performance of the Merkle Tree therefore relies heavily on the hash function used to fingerprint the content.

The Merkle Tree has many virtues. In this project, it is being employed for tables, where it can not only be used to calculate the Jaccard index between two documents (i.e. to what degree they have changed) — but also where the changes were made. Due to the fact that this is to be used inside networks that are already secure from outsiders, using processing power on other security meassures than collision security is of lesser importance. As previously mentioned, fingerprinting algorithms belong where purposeful data tampering is of little concern.

In this project, Rabin's Fingerprinting Scheme proved to be the best option, due to it's simplicity, collision resistance and overall good performance. Of course, any hash function can be used for this purpose. Merkle Trees generally excell in cases where the chunk-size is fixed, or if the text has obvious separators and delimiters — making it a suitable tool for CSV files, JSON files and so on.

Figure 10: Example of a Merkle Tree with a depth of 3



Merkle Trees are general purpose datastructures and also have uses outside of inconsistency detection. An example of this is MD6, which uses the Merkle Tree instead of the Merkle-Damgård construction used by SHA-1, SHA-2, and MD5.

2.1.1 Memory Usage May Become a Bottleeck

The most obvious drawback of using Merkle Trees is the memory consumption. A small Merkle Tree with for example 24 terminal nodes can easily fit in the level 1 cache of the CPU, since the number of nodes in total is $2^n - 1$ (where n is the number of nodes) — and the hash digest using Rabin's Fingerprinting Scheme is no more than 8 bytes. With thait being said, a table used in Excel often has hundreds of rows, and CSV files used in datasets are often kept as large as possible. Open Office Scalc has a maximum row count of 65 535 (or $2^{16} - 1$). A Merkle Tree with 65535 bytes of content + overhead will fit in the L1 cache of a CPU core — but if a larger hash is used, such as a fuzzy hash, it

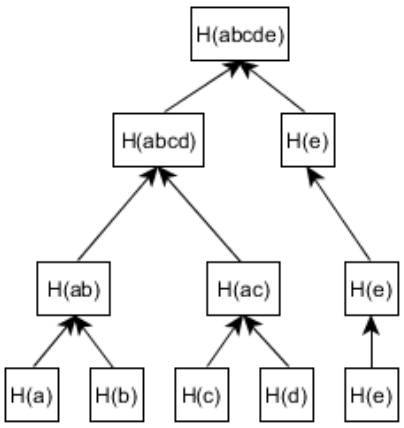
will not. If there are many files being hashed, it might not even fit into the computer's main memory, meaning that the hard drive will have to be used for caching.

Because of the potential size of a Merkle Tree (in any case, it is larger than a hash of the entire file as one chunk would be), lookup tables (these are described in detail in the next chapter) for Rabin's fingerprinting schemes, Bloom filters, a good traversal algorithm and so on are very important.

2.1.2 Odd Number Trees are Also Possible

Since some tables have an odd number of rows an columns, it is also important that a binary tree can be constructed with an odd number of terminal nodes. A Merkle Tree having an odd number of terminal nodes is not fundamentally different from one with an even number. However, it will have a duplicate of itself as a parent, which will in turn have a duplicate of itself as parent, and so it continues until it reaches a level with an even number of nodes.

Figure 11: Example of a Merkle Tree with an Odd Number of Terminal Nodes

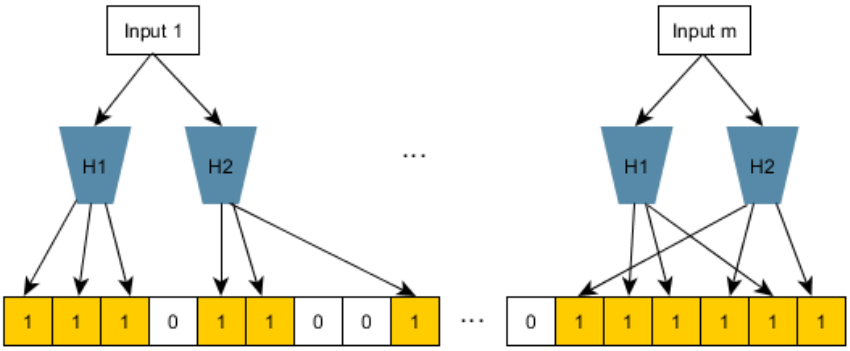


2.2 Bloom Filters — and How They Differ from Hash Tables

Sometimes it is not necessary to retrieve the object itself, just to know whether it exists or if it does not exist inside a dataset or a datastructure. This is where Bloom Filters (first created by Burton Howard Bloom) play a vital role. A Bloom filter is a probabilistic datastructure, containing bits from other objects. [35] While a Bloom filter cannot with complete certainty say that an object is in a set or not, it can nevertheless say that it *probably* is or that it *definetely* is not (i.e. there is always a small chance of a false positive, but never any chance of a false negative). [36]

A Bloom filter uses hashing to add a reference to an object, but differs from a hash table in that it will only store a few bits from the objects it is referencing. In this project, the bloom filter will be based on the hashes from the terminal nodes in the Merkle Tree. To check if a file exists in a dataset or a datastructure, the very same aproach is taken (i.e. the bit set is checked), but no bits are changed or overwritten. If there is no match in all the bits being checked, then the elements do not exist.

Figure 12: Example of a Bloom Filter with m entries, 2 digests per entry, 3 bits per digest, and n bits in the bitset



A Bloom filter is initialized with an all-zero bit set — with a length based on the expected number of entries multiplied with the number of bits per element and multiplied with the number of hash functions used per element. The higher the expected number, bits per element, and hash functions, the larger the bit set (note that a larger bit set will use more memory). Insertion and doing a lookup in a Bloom filter has a constant running time of $O(k)$, where k is the number of digests (in this project in the form of fingerprints) used.

The main reason for using a Bloom filter instead of hash tables is that insertions and lookups are computationally cheap, and at the same time, an entire bloom filter will typically fit into the level 1 or level 2 cache of a CPU core. The Bloom Filter itself was designed to avoid unnecessary disk or main memory accesses, meaning that it will be useful later on in the Merkle Tree-based implementation.

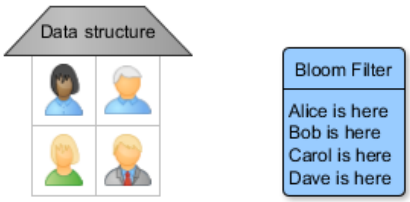
2.2.1 A Bloom Filter Still has it's Weaknesses

While a Bloom filter is very space and time efficient, it does not "know" anything about the entries, apart from the fact that they either exist or they do not exist — and it cannot be used to retrieve an object (or anything belonging to it). As previously mentioned, it does not store the objects themselves — just bits from them. Furthermore, it is not possible to delete an element from a Bloom filter.

The difference between a Bloom Filter and a tree, an ArrayList or any other datastruc-

ture that actually holds the object, can be seen as analogous to the difference between a hotel and asking the receptionist whether a given person lives in this hotel or not. This receptionist will never say that someone who lives in this hotel does not live there, but occasionally, he/she might say that someone who does not live there (but looks similar to someone who does) is in fact there. Moreover, as in real life (in accordance with the law), this receptionist is not allowed to tell where in the data structure or data set the object is located.

Figure 13: Bloom Filters will tell if an element is probably in the dataset or data structure being referenced



2.2.2 What are the Probabilities of a False Positive?

The probability of a false positive increases with the number of elements, but will decrease with the number of digests used per element or the number of bits per element. This is because false positives are statistically independent events. The odds of a false positive for an entry of one bit can be summarized in this formula [37]:

$$p = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} \approx (1 - e^{-kn/m})^{k} \tag{15}$$

Here, k is the number of hash digests per element used, n is the number of inserted elements, m is the length of the bitset (in bits) and m_x is the number of bits per element. If there are more than one bit per element, then the formula becomes:

$$p = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{km_x} \approx (1 - e^{-kn/m})^{km_x} \tag{16}$$

2.2.3 Ideal Number of Fingerprints and Bits per Element

There is no "ideal" amount of bits per element. Rather, it becomes important to find a trade-off between accuracy and space consumption. As a rule of thumb: the more elements in a Bloom Filter, the higher the amount of bits per element needed to reduce the rate of false positives. More technically, the rule of thumb can be formulated as:

$$m = \frac{-n \times ln(p)}{ln(2)^2} \tag{17}$$

Where m is the number of bits in the filter, p is the acceptable false positive rate, and n is the expected number of elements in the filter. If the expected number of elements is 100, and the acceptable false positive rate is 4%, then the equation above becomes:

$$m = \frac{-100 \times ln(0.04)}{ln(2)^2} \approx 670 \tag{18}$$

From the answer above, it can therefore be seen that 7 bits per element is a good number for 100 entries.

The ratio of false positives can also be reduced by adding more digests. This does not increase the space usage, but it will increase the CPU usage (the running time thus becoming $O(k)$). The optimal number of digests k, can be calculated as:

$$k = \frac{m}{n} \times ln(2) \tag{19}$$

If m and n from the previous equation on total number of bits used is inserted, the answer becomes:

$$k = \frac{670}{100} \times ln(2) \approx 5 \tag{20}$$

A different approach will be used in this project, due to the way the Merkle Trees are implemented. This will be described in Section 5.

Note than $ln(2)$ refers to the natural logarithm of 2, not the base 2 logarithm.

2.3 Summary

The section on data structures can be summarized as follows:

- A Merkle Tree is a binary tree containing nodes with hashes
- Each node in the Merkle Tree which is not a terminal node, contains the hashed output of the two digests underneath it
- Merkle Trees were invented to detect inconsistencies
- Bloom filters are probabilistic datastructures, meaning that they can tell if an element *is probably* in a dataset or a datastructure
- Bloom Filters only contain a few bits from each object, not the objects themselves

- Bloom Filters have a constant running time for insertions and lookups

3 k-NN is used for Supervised Machine Learning

While a system classifying few files with very clearly defined differences does not have to be very intelligent, a system acting outside of an artificial test case and inside real world applications should at least be able to "reason" or make a qualified guess rather than just act on impulse. This is where machine learning and pattern recognition comes into play.

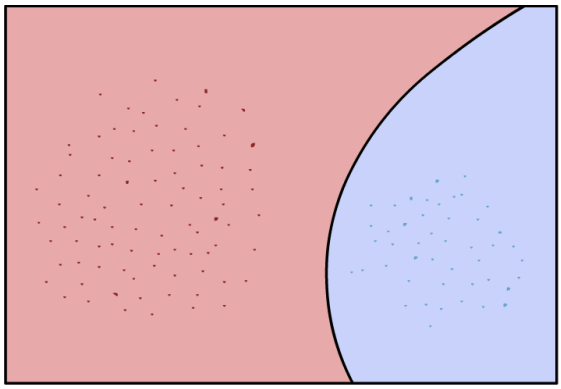
There are different models one can use. This project will utilize supervized machine learning, where one first trains the classifier using a training set with tables from each class, and the tests it's knowledge with a separate testing set [38] — with tables from each class. Essentially, one is telling the computer "this is how class 1 looks", "this is how class 2 looks" etc. and then one checks if the computer can recognize a new dataset with new tables from the same classes. For numerical classification tasks where performance and simplicity is important, the most-likely estimate (henceforth referred to as MLE), the Parzen window and the K-nearest neighbor (henceforth referred to as **k-NN**) are the most widely used (a decision tree is Boolean and not numerical).

In real life, outside of a controlled environment, the matrix/covariance and the expected value (denoted as Σ or σ and μ) are rarely known — and there are often few reasons to spend more resources on calculating these when doing classifications. Thus one cannot usually assume or "know" anything about the shape of the discriminant function for each sample or vector being classified. Therefore, a non-parametric (**k-NN** and Parzen) approach is often better suited than a parametric approach (MLE). This project uses the **k-NN**, due to the fact that the similarities are based on edit distance between text files and not the Euclidean distance between vectors. Neither **k-NN** nor Parzen windows need to know the coordinates (in this case, there are in fact no coordinates), just the distance between the test vectors and the training vectors. The vectors in this thesis are represented by plaintext tables.

Supervised machine learning demonstrated to humans typically deal with decision boundaries. These are boundaries that determine when a vector should be classified as belonging to which class ω_i. The boundaries themselves represent exactly where the probability of two or more classes is equal. An example can be seen below:

In this example, a 2-dimensional approach with two classes is used, called ω_{red} and ω_{blue} respectively. If the testing vector is on the left side of the boundary, it will be classified as ω_{red}. If it is on the right side, it will be classified as ω_{blue}. This section will typically use two-dimensional examples even though the classifier in this project is one-dimensional. This is due to the fact that the working mechanisms of a classifier are easier to illustrate with two dimensions rather than one dimension.

Figure 14: Example of a two-class, two-dimensional approach with MLE



It is a common misconception that the most advanced form of machine learning is automatically the best one. When categorizing objects based on simple attributes (eg. edit distance of fingerprints, color, shade, and so on) it is better to have a simple artificial intelligence that thinks within clear boundaries, rather than to try to program something that will make a lot of complex considerations before finally deciding; the latter will result in a lot of resources being spent on something that is not necessarily more accurate or "better" by any means.

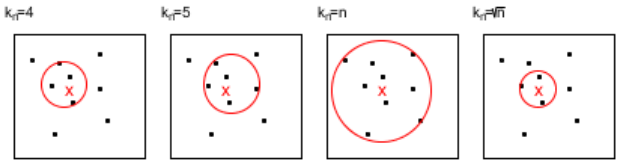
A caterpillar will for example never think that "this rutabaga tastes so much better than carrots" or "I do not like the color of these plants", or potential health benefits or ethical concerns accociated with eating something that might belong to a farmer — yet it will still have clearly defined boundaries of what to eat and what not to eat, what it might eat when there is nothing else around — and of course what could potentially eat it, signalling that it should crawl to safety. Despite not having an advanced brain, a caterpillar excels at what a caterpillar is supposed to do, which is to feed on plants and someday grow into an adult insect. Indeed, a complex brain capable of abstract reasoning will not be of significant, evolutionary advantage while doing this; instead, it might actually be a disadvantage, since an advanced brain requires much more energy to function. In the same manner, a system that classifies hash digests based on edit distance, whether the amount of money is enough based on weight, whether fruit can be sold or not based on color and dents, and so on should be kept simple, *with no more loose parts than what is strictly necessary.*

3.1 How a Non-parametric Classification Algorithm Works

In a non-parametric approach, one does not assume anything about the distribution; i.e. one does not assume that it will look like a bell curve after n samples. A lot of resources can be conserved this way, when the mean or the variance are simply not needed.

Non-parametric approaches have a probability density function function on the form

*Figure 15: How a hypersphere in a two-dimensional **k-NN** approach with $n = 9$ and various k_n values operates*



[39]:

$$p_n(x) = \frac{k_n/n}{v_n}. \tag{21}$$

Both **k-NN** and Parzen Windows are non-parametric functions utilizing a hypercube, a hypersphere or any other similar figure allowing a clearly defined voting scheme, created around the vectors (samples) being classified. The size of the figure is denoted as v_n in equation 21, while the number of elements is denoted as k_n. v_n is the same as h_n^d, where h_n is the length of each side of the hypercube, while d refers to number of dimensions. Typically, h_n is a function of the number n of samples in the dataset, eg. $h_n = \sqrt{n}$.

The class with the most vectors inside the hypercube, hypersphere or any other representation of the selection area (this can be seen as a window) "wins" [40], in the sense that the probability of it belonging in that class is higher than it belonging in a different class — and the vector becomes a member of their class. **k-NN** and Parzen windows differ in they way they achieve this goal; *the former has a fixed number of elements inside the window*, and will adjust the size of the window accordingly; *the latter has a fixed window size*, and will adjust the number of elements inside accordingly.

In Parzen windows, the number of samples (k_n) in the window is given by

$$k_n = \sum_{i=1}^{n} \phi \frac{(x - x_i)}{h_n} \tag{22}$$

where x is the vector being classified, x_i is a given sample in the dataset, and h_n is the length of each side of the hypercube, the radius of the hypersphere or any similar abstraction. Recall that this number is fixed in k-NN. Inserted into equation 16, this will lead to the probability being:

$$p_n(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{V_n} \phi \frac{x - x_i}{h_n} \tag{23}$$

Typically, prior distribution is used as well in the form of discriminant functions, to even out the odds some more. Discriminent functions with prior distributions are written the following way:

$$g_i(x) = P(\omega_i)\frac{k_n/n_i}{v_n}. \tag{24}$$

Here, n_i is the number of elements in the given class. The prior distribution $P(\omega_i)$ is simply calculated as $\frac{n_i}{n}$. Bayes theorem states that one should choose $g_i(x)$ if $g_i(x) > g_j(x)$, and $g_j(x)$ if $g_j(x) > g_i(x)$.

For **k-NN**, with a $k_n = m$, the hypercube or hypershpere size v_n is calculated as r^2 or $\pi * r^2$ respectively, where r is the Euclidean distance between the vector being classified and the m-th nearest vector in the dataset. Calculating the Euclidean distance if the number of dimensions is more than one is normally quite expensive on the x86 architecture, but on some architectures (most famously on IBM's PowerPC architecture), the square root is not calculated directly, rather it is approximated discretely. This is, however, of lesser importance here, as the edit distance between two digests is used instead, The edit distance can be inserted into equation 24, so that:

$$g_i(x) = P(\omega_i)\frac{k_n/n_i}{DamLev(x, x_i)} \tag{25}$$

Where, $DamLev(x, x_i)$ is the Damerau-Levenshtein distance between the vector being classified and a given vector in the dataset. The classifyer used in this project will also use prior distributions when classifying samples.

Due to the inherent simplicity of **k-NN** and due to the fact that it computes much more rapidly than Parzen Windows when using a string matching algorithm, this approach was chosen to do the classification in this project.

Apart from the fact that the prior distribution is used when calculating the discriminant function in this project, the vectors are all uniformly weighted; that is, there are no additional weights except for the prior distribution. In theory, nothing is "known" about the files used for testing the classifyer and as such, going by Occam's Razor makes the most sense.

Example of How k-NN is Calculated in this Project

Consider an example where there is two classes, each with 8 samples, for a total of 16 samples in the dataset. Any number is possible, but it is preferred if there is an equal number of samples in each class, since this is a file- and table matching tool using prior distributions. Each class has the following digests:

The testing vector used in this case has the digest "1x4H1F+a". The real **NFHash** uses 64 characters in the digest, but the digests here are 8 characters for illistrative purposes. Both classes have a prior distribution of $P(\omega_1) = \frac{4}{8}$ and $P(\omega_2) = \frac{4}{8}$, respectively.

Table 13: A Two-Class Dataset

ω_1	ω_2
x\41F+aa	\\4D76fa
x+41FHaa	3D76fa90
1x\41F+a	3D76ccJB
1x331F+a	5H76ccJB
x331F+aA	8H76cdJB
+aAx331F	HH76cdJH
x+41FHaa	H+41FdjH
x\41gHab	H841gdjH

Table 14: Damerau-Levenshtein distance between test sample and samples in dataset

ω_1	ω_2
4	6
5	8
4	8
2	8
4	8
8	8
5	6
6	8

Normally, there are a lot more elements in a training set, and a lot more elements in a testing set than this example might indicate. The numbers are kept low for illustrative purposes in this example. A naive (but often efficient) way to choose k_n, is to simply set it equal to $\sqrt{n} = 4$. For illistrative purposes, in this example, $k_n = 3$. With this, the discriminant functions for each class can be calculated:

1. $g_1(x) = \frac{4}{8}\frac{3/4}{4} = 0.09375$ ($v_n = 4$, which is the edit distance to the 3. nearest sample. Consider it the radius of a one-dimensional window required to trap the three nearest samples)
2. $g_2(x) = \frac{4}{8}\frac{3/4}{8} = 0.046875$

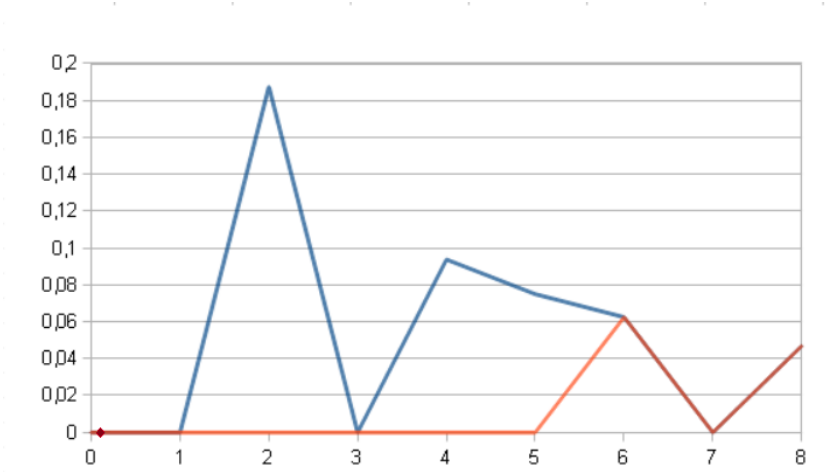
Since $g_1(x) > g_2(x)$, *the test vector is classified as a member of class* ω_1 in accordance with Bayes' theorem. A density function with the different values can be seen here:

In the aforementioned graph, point 0 is where the testing vector is. The coordinates are not fixed; if a different testing sample was used, the graph would look completely different.

Pitfall: k-NN and K-means Refer to Two Completely Different Things

k-NN is often confused with K-means, which is a clustering tool. Apart from the fact

Figure 16: Class ω_1 is represented by the blue line, while class ω_2 is represented by the orange line. The purple line is where the two probability of the two classes intersect



that both algorithms are pattern recognition tools classifying samples based on ditance, they have nothing in common. K-means does not use supervised learning; i.e. there are no external classifications (nobody tells it how to tell the difference between several buckets or classes). Instead, it groups a vector into the appropriate cluster based on which cluster is the nearest one. K-means is not used in this project—and would not work well since the classifier in this project does not use actual coordinates, just edit distance.

Intuitively speaking, K-means has more in common with graph clustering based on peer pressure than with k-NN.

3.1.1 How is This Implemented in the Project?

All that a Java function classifying a vector in this project needs to do, is to tell which $g_i(x)$ has the highest probability by utilizing simple mathematics and rudimentary data structures. This is described in detail in chapter IV.

3.2 Precision and Recall

The two most important ways to evaluate a simple classifier is by finding the **precision** and **recall** rate. **Precision** is the percentage of retrieved elements that are actually relevant, while **recall** is the percentage of releveant elements that are actually retrieved. [41] Typically, a good way to evaluate a classifier after training is to use a confusion matrix. [42] This will feature the times a prediction of a given class is right, and the times it is wrong (and what the actual result was instead). Matrix dimensions represent

the number of classes in the dataset (a two-class dataset will be represented by a 2×2 matrix, a three-class dataset by a 3×3 matrix and so on).

The formulas for accuracy and recall can be generalized as:

$$Accuracy = \frac{(relevant\ entries) \cap (retrieved\ entries)}{retrieved\ entries} \tag{26}$$

$$Recall = \frac{(relevant\ entries) \cap (retrieved\ entries)}{relevant\ entries} \tag{27}$$

In table 15 an example of a two-class confusion matrix is shown. The cell a represents the times when ω_1 is both the predicted and actual result — while d represents the time this is the case for ω_2. b represents all the time what is predicted to be ω_1 is really ω_2, while c represents vice versa.

Table 15: A confusion matrix explained

		Predicted	
		ω_1	ω_2
Actual	ω_1	**a**	**b**
	ω_2	**c**	**d**

For ω_1, b is a false while c is a. From this, the accuracy AC can be calculated as:

$$AC = \frac{a + d}{a + b + c + d} \tag{28}$$

While the recall rate for each class becomes:

$$RC_{\omega_1} = \frac{a}{a + b} \tag{29}$$

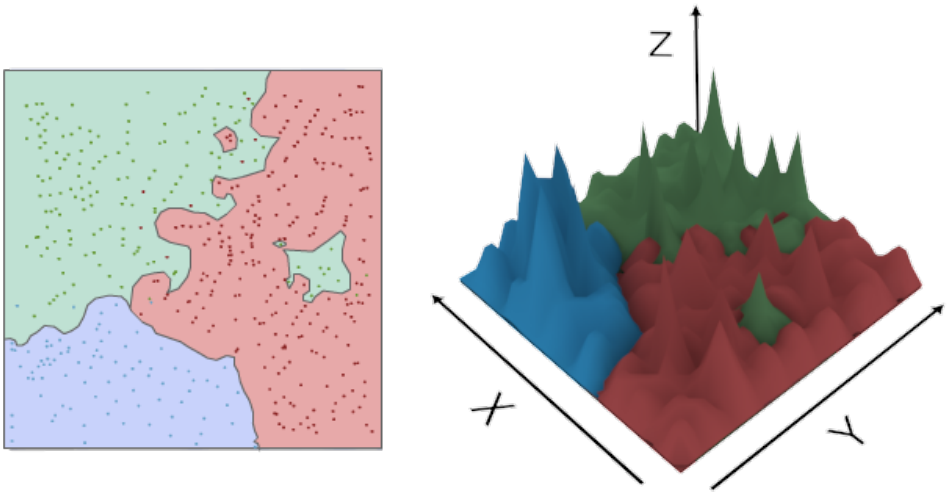
$$RC_{\omega_2} = \frac{d}{d + c} \tag{30}$$

Accuracy and recall will serve as a leitmotif in the Testing and Analysis chapter.

3.3 Overfitting Might Lead to Undesireable Results

Overfitting refers to when a classifier that is trained in a manner that makes it take exceptions, noise, inaccuracies and so on into consideration [43]. This can lead to undesireable results, leading to a sample being classified as something it is not, due to it's superficial resemblance to an exception from another class.

*Figure 17: Example of a two-dimensional dataset with three classes using the **k-NN** approach. The jagged, spiky appearance of the decision boundaries and the graph, can be both a fatal flaw and an advantage. This picture shows an example of overfitting. The more elements that are used inside the window, the less spiky the decission boundary will be*



Overfitting can partially be overcome by using a fuzzy hashing scheme, since this will also compress all the dimensions into one dimension. Nevertheless, it is not a cure for overfitting if all other steps are ignored. These steps all work with **k-NN**, but different appraoches may sometimes be taken with Parzen windows or MLE.

3.3.1 Avoiding Overfitting

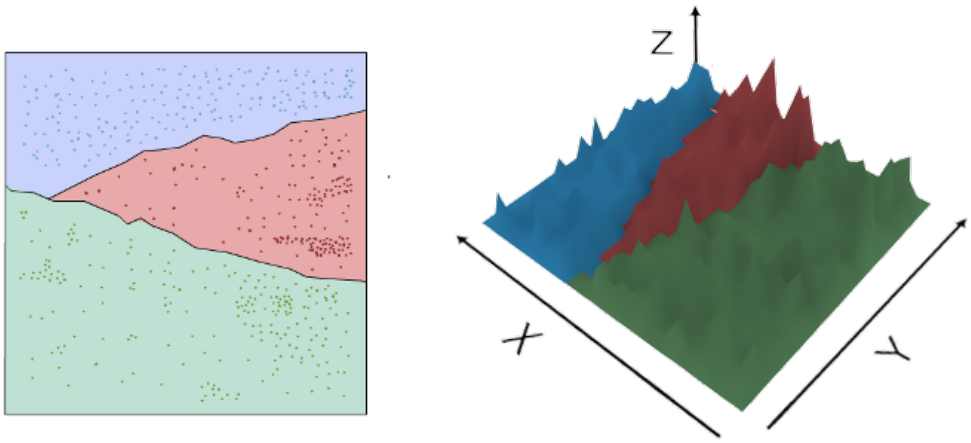
Choose an appropriate k_n value

The most important step is to choose a good k_n number. Unless the dataset is very separable (preferably linearly separable, with close to linear decision boundaries) — or the number n of elements in the dataset is very small, one should not simply base the classification on *the* nearest neighbor ($k_n = 1$). The more elements inside the window, the less the outliers and the noise among the samples will matter, and the more accurate the result will be. At the same time, the higher the number of elements in the dataset, the higher the amount of resources used. A tradeoff is therefore necessary. Generally, the smaller the difference between n and k_n (in terms of ratio), the more the density function will resemble a continuous distribution.

Typically, k_n is a function of n (the number of elements in the dataset). This could for instance be $\sqrt{n}$, $\sqrt[3]{n}$ or $n/10$.

Use Enough Training Sets

Figure 18: Example of a two-dimensional dataset with three classes where overfitting is avoided



One of the most important theorems in basic statistics, is the *Weak Law of Large Numbers*, stating that after a large number of tries, the average should converge towards the expected value. There are no expected values in ***k-NN***, but the law will still apply indirectly, since using enough datasets will mean that exceptions and outliers are of less significance. Note that the larger the training set, the more important a correct k_n number becomes, meaning that using enough training sets is not a perfect "cure" against overfitting.

Keep Dimensionality at a Reasonable Number

As previously mentioned, a classifier uses one dimension for each feature. Typically, adding more features (and thus more dimensions) makes it easier to tell several classes apart, since this will reduce the error rate $P(e)$.

As an example, consider classifying combustion engines. In this example, there are two classes in total ω_{petrol}, and ω_{diesel}. First, assume there is just one dimension, namely compression ratio (this is lower for petrol and higher for diesel). Typically, a diesel engine has a higher compression ratio than other engine types, but there are exceptions; a petrol-powered race car will have a higher compression ratio than a diesel-powered tractor, and accidentally filling petrol in a diesel-powered engine is an expensive mistake (which is why loss functions can also be used as well). More dimensions can be added, such as weight/cylinder volume ratio (diesel-powered engines are heavier and more robust than petol-powered engines), rotations per minute of the crankshaft, and torque/horsepower ratio (a diesel-powered engine usually has a higher torque). Four dimensions should in theory give a better result than just one, but if the number is too high (typically over 20 dimensions), it will also lead to the machine *learning exceptions that are specific to the training dataset and that do not necessarily reflect the real world.* This is called "the Curse of Dimensionality". [44]

A fuzzy hashing scheme (such as **NFHash**) can compress a multidimensional table into one string, that can be used in a one-dimensional case. This approach is already being used in various forensic tools utilizing **Spamsum** or **Nilsimsa**. This does not make a classifier immune to overfitting, but it will still significantly reduce the risk of this.

Use the Sliding Window Size Wisely

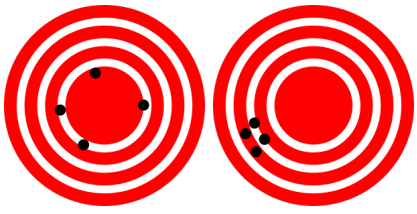
In this project, the k-NN classifyer is one-dimensional (it makes use of NFHash to flatten the dimensions). However, this does not make it immune to overfitting. One important factor is the sliding window size. The smaller this is, the more sensitive the algorithm is in terms of details. If there are many training samples per class, then a coarse granularity (large sliding window size) will lead to more misclassifications; if there are few samples in the training set, a too small one might lead to misclassifications instead. Both are examples of overfitting.

Resampling is Sometimes Necessary

Resampling simply refers to moving one element from one class in the training set over to another, based on it's discriminant function. This is no different than when a trained classifier is given a fresh vector to classify, except that the vector is already from the training set used to train the classifier. Resampling methods can be grouped into *Boosting* and *Bagging* [45], among others. (**b**ootstap **agg**regat**ing**). Bagging means that you take samples (with replacement) at random from the training set and use them to further improve the classifier. Bagging decreases the variance, and thus improves generality and precision. Boosting trains a new classifyer (or trains the same one with completely blank sheets), with special emphasis on the elements the previous classifier misclassified. Boosting decreases the bias, and thus improves accuracy.

Note that there is a tradeoff between variance and bias. Errors due to bias are due to how far off the predictions of the classifier are from the correct value. Errors due to variance refers to how much the predictions for a given point x_i differs from the expected value. High variance will typically lead to overfitting, while high bias will typically lead to underfitting (which means that the classifier is not trained to detect details on a satisfactory level). Note that low bias and low variance are not mutually exclusive, but this can often be hard to achieve.

Figure 19: Left: Errors due to high variance. Right: Errors due to high bias (this picture is public domain)



Due to the fact that the aforementioned resampling meta-algorithms are computationally

expensive to do automatically, and the fact that the approach used in this project is based on edit distance and not actuall coordinates, resampling needs to be done manually on the datasets used in this library (i.e. no code doing it exists here), by filtering the training set and reclassifying entries manually. Resampling is mostly relevant when the sample sizes are very small.

3.4 Other Potential Problems with the k-NN Algorithm

Large ranges usually dominate smaller ones, so that some dimensions gain more influence than others. In the aforementioned example on engines, the range between rotations per minute for the crank shaft for a gasoline engine can be anything from 3000 (some "malaise era" engines) to over 12,000 (high performance normally-aspirated engines). By comparison, the torque-to-horsepower ratio is usually between 1 and 2.6. Clearly, the former will have more influence than the latter. Without a fuzzy hashing scheme, one can attach weights to certain dimensions to improve their ability to influence the decision.

A high variance might make it difficult to tell classes apart, leading to the fact that a very large sample size might be necessary. This is also a problem with Parzen windows and MLE.

Some misclassifications are also more expensive to make than others. Misclassifying a diesel engine as a gasoline engine is much more expensive than the other way around, and if there is a close to equal chance of classifying a vector as either gasoline or diesel, the decission boundary needs to be moved just in case (so that diesel becomes the more likely class). This is done by loss functions.

Loss functions can be calculated in many ways, but will not be used in this project. There are a number of reasons for this, but first and foremost, it is difficult to predict how much more expensive it is to misclassify something as a table from class A when it really is a table belonging in class B than it is to misclassify it as a table of class B when it really should be in class A. A secondary reason is that it is computationally costly, and complex to implement — meaning that it adds a lot of development time to something the user might not even need.

In a few rare cases, k-NN might give an ambiguous result; imagine there are 3 classes, ak_n value of 5 and a total of 24 samples in the dataset. Now imagine that each class has 8 samples, leading to an identical prior distribution $P(\omega_i)$ for all classes. Suppose ω_1 and ω_2 has an equal number of samples within a hypercube or hypersphere, and that the edit distance between the reference vector and each vector in ω_1 corresponds to the edit distance between the reference vector and each vector in ω_2. This will lead to the unfortunate situation where the classifier cannot decide which class to choose. Although a very unlikely scenario if overfitting can be avoided, it is nevertheless an interesting thought experiment.

3.5 Summary

- For a non-parametric algorithm, one does not assume anything about the final shape of the density function. MLE is a parametric algorithm — Parzen Windows and **k-NN** are not
- Supervised learning means that a classifier is taught using external classification. A programmer "shows" it a training set to teach it to classify correctly
- k-NN and Parzen Windows both use a hypercube (a so-called window) with a majority vote to tell the classifier where to put the reference vector
- Overfitting means that a classifier is taught to pay too much attention to noise, exceptions and other inaccuracies. This is not a desireable trait
- k-NN is inaccurate for small sample sizes
- A good way to test a classifier is to calculate the accuracy and the precision rate

4 Design I: Design of No-Frills Hash

The name of the fuzzy hahing scheme designed in this project is No-Frills Hash. This was designed to better suit the needs of file and table classifications rather than spam filtering and forensics — which is what most currently existing fuzzy hashing schemes are used for. Some design criteria were needed before the algorithm could be designed. While being heavily inspired by both Nilsimsa and Spamsum, NFHash is not based on any of them.

4.1 Main Design Criteria:

The two most important properties in Spamsum, according to the creator Andrew Tridgell, are defined as follows [26]:

> "**non-propogation**
>
> In most hash algorithms a change in any part of a plaintext will either change the resulting hash completely or will change all parts of the hash after the part corresponding with the changed plaintext. In the spamsum algorithm only the part of the spamsum signature that corresponds linearly with the changed part of the plaintext will be changed. This means that small changes in any part of the plaintext will leave most of the signature the same. This is essential for SPAM detection as it is common for varients of the same SPAM to have small changes in their body and we need to ensure that the matching algorithm can cope with these changes.

alignment robustness

Most hash algorithms are very alignment sensitive. If you shift the plaintext by a byte (say by inserting a character at the start) then a completely different hash is generated. The spamsum algorithm is robust to alignment changes, and will automatically re-align the resulting signature after insertions or deletions. This works in combination with the non-propogation property to make spamsum suitable for telling if two emails are 'similar'."

The aforementioned criteria has served as the main criteria when designing NFHash — as they can be considered corner stones of any fuzzy hashing scheme. A table on the form [A, B, C, D, E, F], needs to be recognized as "very similar to" a table on the form [G, B, A, C, D, E, F].

Furthermore, performance and accuracy has been crucial. The secondary criteria are listed below:

- Less overhead than Nilsimsa — due to the fact that typically, more than two files are compared. The final version is similar in terms of overhead, leading to this being the only criteria not met
- Customizability, so that a user can decide to which extent minor details affect the digest. Spamsum already has this ability, Nilsimsa does not
- Better overall performance than Spamsum, hopefully close to Nilsimsa
- Signatures that are portable — and not dependent on any datastructure, keys or deserialization mechanism
- Lower memory consumption than Nilsimsa
- To a lesser extent robust against shuffling of the elements. If several rows are moved into new positions without any content being altered, the digest should at least look recognizable, but not necessarily to the extent where one can pinpoint exactly where the change was made
- Few loose parts, so that the result is predictable and the algorithm will perform well — and the code will be easy to port and debug between various platforms and various programming languages (hence the name "No-Frills Hash")

These criteria are also important, due to the fact that it will be used for classifying multiple plaintext files at one — files that are often larger than a spam e-mail. Note that this does not make NFHash "better" than Nilsimsa and Spamsum; it just means that it is used for different things, and as such some features in Nilsimsa and Spamsum are *redundant for file and table classifications*. NFHash is designed for plaintext files and tables (for software similar to Avito Loops), and files with very little bytewise repetition in them; Nilsimsa and Spamsum are designed for spam detection and forensics (NFHash should not be used for the latter).

4.1.1 "Order Robustness"

While Andrew Tridgell coined the terms "alignment robustness" and "non-propagation", he makes little mention of the order of the elements. Obviously, this is less important than the two main criteria, since the elements in a table are rarely moved without any insertions or deletions.

This can be demonstrated with the two phrases "Stolen This Code Is" and "This Code Is Stolen" (a digest in NFHash cannot be longer than it's input string). These two strings will give the digests "iRKJALxlFEOx2KBAx8OD" and "lFEOx2KBAx8OxiRKJALD", where the two digests will have exactly the same content, but in different order. Note that Damerau-Levenshtein will not take this into consideration if the substrings are too many and too short. This problem can be solved by Jaccard, but since Jaccard does not take into consideration where the characters are located, it might lead to a high collision risk.

Spamsum, Rabin-Karp, and rsync also have this property, courtesy of utilizing rolling hashes. Because of this order robustness, Damerau-Levenshtein will also serve it's purpose well in this project.

4.2 Outline of NFHash

Data: A byte array of arbritary length, window size and boundary size
Result: A 64 byte base64 digest
while *Byte array has more bytes* **do**
  Slide n size window over input;
  if *CRC64 of elements inside window* $\wedge$ *boundary size* -1 = *boundary size* -1 **then**
    Create a boundary and start a new block.;
    if *More than one boundary exists* **then**
      Hash elements between this and the previous boundary — as well as the CRC-64 digest in the current window with Mersenne Primes; i := Mersenne Prime digest $\wedge$ base number - 1; Next byte in digest = base 64 alphabet[i];
      if *64 blocks created* $\wedge$ *input still has more bytes to visit* **then**
        Go back to beginning of fingerprint and start from there, but only update 1/4 of the characters this time; If necessary, the algorithm will go back to start again and update 1/16 of the characters.
      end
    end
  end
end
return digest

Algorithm 2: NFHash explained

The non-propagation and the insertion robustness problem is solved by using a sliding

window, that will automatically resynchronize to compensate for insertions or deletions — and by having the hashing scheme create the digest character-by-character. Both are non-issues in Spamsum and Nilsimsa (they handle this very well), but are nevertheless important to remember.

NFHash revolves heavily around Boolean algebra and uses CRC-64 as a rolling hash. This is more accurate than Adler's Chekcsum, while at the same time only being a little slower if lookup tables are used. In this project, a hard-coded lookup table with partial seeds based on the ISO-3309 polynomials is used. This polynomial is:

$$x^{64} + x^4 + x^3 + x + 1 \tag{31}$$

Since NFHash utilizes a sliding window, all the elements inside the window will be used as input for the CRC hashing. This sliding window will slide one byte at a time, as demonstrated in figure 20. The size of the sliding window (use a power of 2 for best results) will determine to which extent minor changes will affect the digest. A too large sliding window will not catch small changes at all, while a window that is too small will lead to a digest that changes very much due to small changes in the input file.

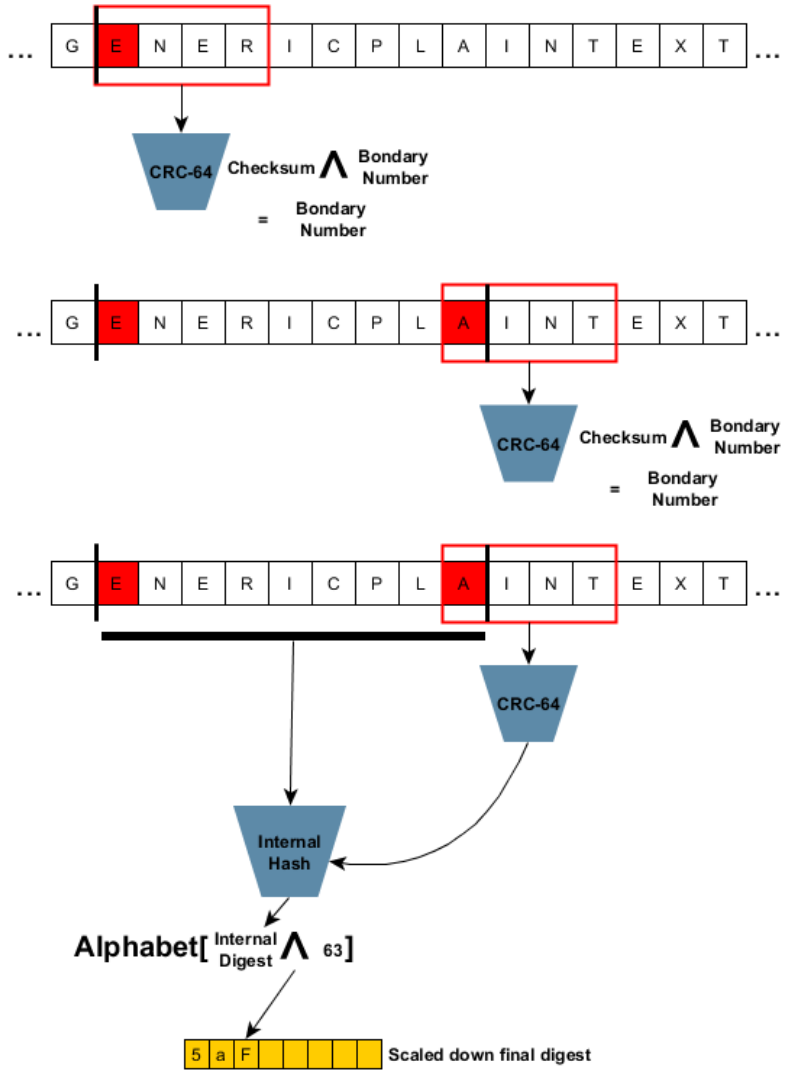
Whenever the digest ANDed with the predefined boundary number matches this boundary number, a boundary is drawn. If there are more than one boundary, the bytes between the current boundary and the previous boundary are hashed, together with the digest from the CRC-64 algorithm. The digest each time will be one base64 character. The risk of this byte matching another byte (regardless of input) is never better than $\frac{1}{64}$, meaning that even a very simple hash function is sufficient. The total digest size of NFHash is 64 characters; the odds of the entire digest string colliding with another digest string is therefore $(\frac{1}{64})^{64}$, due to statistical independence.

In NFHash, the rolling hash approach was chosen instead of accumulators, due to the fact that while Nilsimsa is fast on larger filer, the accumulator approach nevertheless causes a high overhead. If not implemented well, it will also be quite memory-intensive.

While not a strict requirement, NFHash works better if the boundary number and the sliding window size are both a power of two. This is due to the fact that no rudimentary arithmetic operations (except plus and minus) are used. Instead, it will make use of bitwise shifts, AND operations and XOR operations. This greatly improves performance, while not having any significant drawbacks, given that on a byte level, the collision risks can never be better than 1/64 when using the base64 approach.

NFHash utilizes a fixed boundary number, rather than one that is guessed by an algorithm. In Spamsum, if no boundary number is specified, it will try to approximate a boundary number that is equal to the length of the input byte string divided by the total digest length. It will do this by doubling the boundary number until it is bigger than the input length divided by the digest length. Unfortunately, using an algorithm that will guess the boundary number uses a lot of resources.

Figure 20: NFHash explained in three steps



Even setting the boundary number equal to the input length divided by the digest length cubed will give a fairly accurate result in a *NFHash*. Therefore, if the size differences are not too big, it is best to set the boundary number to the smallest previous number that is a power of two relative to the size of the smallest file in the dataset divided by the digest length. If the smallest file is 100 KB, and the digest size is 64 bytes, a good boundary number is 1024.

4.2.1 Setting a Good Sliding Window Size

Let b denote the boundary number, n the input length, and h the digest length. b will ideally be the largest power of two number smaller than $\frac{n}{h}$. For sliding windows, the answer becomes a little more blurry. A small window size makes the algorithm more sensitive to smaller changes, but might cause it to "miss the bigger picture". In other words, if the window size is to small, it might give two very different digests for files that are not that different; if the window size is too big, it might give very similar digests for files that are somewhat different.

4.3 Hashing and Mersenne Primes

Take the character c from the byte array and the long h from the CRC-64 (the digest from the current sliding window) as an input. If $h > 2147483647$ (*the* 31 bit Mersenne prime), let $h_1 = h \wedge 2147483647 + h >> 31$. If $2147483647 > h > 524287$, let $h_1 = h \wedge 524287 + h >> 19$. Repeating this process with an even smaller number in Java will unfortunately make it less collision resistant than 1/64, but due to the fact that C allows twice as big positive primitives, it can be done here (the number $2^{61} - 1$ can be used). Finally let the return value be $h_1 \oplus \neg c$, that is h_1 XORed with the inverse of the character c.

If p is the Mersenne prime largest mersenne prime smaller than h, and b is the bit size of the Mersenne prime, the formula can be summarized as:

$$h_1 = h \wedge p + h >> b \tag{32}$$
$$h_1 \oplus \neg c$$

Obviously, *this is not a strong hash function*, but it is very fast — and sufficient for something that does not require (or allow) a better collision resistance than 1/64.

The digest from the internal hash is ANDed with 63 (base number − 1) to select the character to be added to the fuzzy digest. As an example: 2765 ANDed with 63 will be 25, meaning that character 25 ("Y" in the RFC 4648 implementation) should be chosen from the alphabet and added to the digest.

4.4 Larger Base Numbers are Not Necessarily Better

NFHash uses a base64 encoding for the fingerprint. While primitives such as longs or integers are not capable of holding such numbers, no arithmetic operations are performed directly on the strings after the hashing is done (Damerau-Levenshtein does not care what a character represents, just if it is different or nonexistant in any of the two digests currently compared). This also means that the digests can be represented by strings.

The main advantages of a base64-based approach, is that significantly larger numbers can be represented in a much smaller space — and that collision risks are lowered. A base16 64 byte digest, can represent 16^{64} different values. At first glance, a base64 encoding might therefore seems very redundant — given that for reference the universe itself is "only" $4.32 * 10^{18}$ seconds old. However, a fuzzy hashing scheme is, as previously mentioned, designed to match similar digests, not necessarily identical digests, meaning that the false positive risks are always much higher than the collision risks. Base64 will therefore make the algorithm much more robust, since it can display 64^{64} unique values.

In theory, there is no upper limit to the base number one can use in NFHash. It is also fairly straightforward to modify Spamsum so that is uses an arbritary base number higher than 64. One could potentially use base128, base256 or for that matter base512, but there is one significant pitfall of doing this. The ASCII system has a total of 95 printable characters [46]. While UTF-8 has replaced ASCII, UTF-8 is also entirely backwards compatible with ASCII, and so are most other standards. [47] In theory, UTF-8 will allow an arbritary base number, but a hash digest has to be something that will look the same in any operating system, any program, any forum, any language and any database. For the very same reason, characters not found (or commonly used) in the modern Latin alphabet or modern Arabic number set should be avoided in the digest.

There are many standards for implementing a base64 encoding, but the most used is RFC 4648. In accordance with RFC 4648, the Base64 alphabet is specified as follows [48]:

Table 16: Base64 Encoding Table

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v	Pad	=
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

Note that base16 is not obsolete or outdated by any means. A numeric primitive—such as an integer, a long integer, or a float—cannot hold base64 numbers, but they will in fact hold a hexadecimal number while using no more memory than a decimal number. Furthermore, since hexadecimal numbers are four bits while base64 numbers are six bits, the former will scale better with the binary number system (six is not a power of two). Fuzzy hash digests are too large to fit into a numerical primitive, and thus they are required to use a string. A base64 string will use no more memory than a hexstring (it does not matter what kind of characters the string is composed of).

The ASCII standard does not limit the digest to base64, however. Adobe uses a base85-based approach for some of it's applications (commonly referred to as ASCII85), and there are open-source standards using base91 [49] with good results. The currently existing base91 standard (named "basE91") is freely available under the BSD license. Base91 is not as widely used as base64 (i.e. less is known about how well it performs on a large scale) and lacks a unified standard, which is why it was not used in NFHash.

4.5 Potential Weaknesses

NFHash works best if the boundary number and the window size are both powers of two. It will still work if they are not, but the result will not be optimal. The algorithm is also sensitive to repeating patterns in the text, meaning that it is not suitable for fingerprinting archives, compressed files, and so on.

If the boundary number is too small for the file size, insignificant, but scattered changes will have an impact that is too big on the digest. This is due to the fact that it will go back to the first character of the fingerprint if the 64 character fingerprint is full before all bytes from the input are traversed by the sliding window. If the opposite is true, that is that the boundary number is too big, it might ignore minor changes completely. A rule of thumb in a rolling hash, is to set the boundary number equal to the input length divided by the digest length. Usually, setting it a little too small is less severe than setting it a little too big. Note that in no algorithms utilizing rolling hashes can digests with different boundary numbers be compared.

At it's present stage, the algorithm can not create a fingerprint of a file shorter than the fingerprint itself; if for example a 20 character string is hashed, then the fuzzy fingerprint (if NFHash is configured correctly) will also be 20 characters. A significant issue might also be that using NFHash requires some practice — and some knowledge about the working mechanisms used by it. Therefore, the source code for a lightweight classification tool using it is included as well — in addition to the main source code.

One final weakness is that it might run slow if the window size is set to a large number (larger than 64). Fortunately, very large window sizes are not usually needed.

4.6 Similarities and Differences Between Other Fuzzy Hashing Schemes

Nilsimsa and NFHash have generally nothing in common apart from the outcome, the digest size and the fact that both utilize a sliding window. Nevertheless, studying this algorithm was important, as it was a reference point NFHash could be compared against — and because it is one of the best fuzzy hashing schemes for spam detection.

Spamsum is somewhat similar in the sense that it utilizes a rolling hash, and an internal hash (Adler's checksum and FNV, respectively) — and in the sense that it utilizes the base64 alphabet. Moreover, most practical applications (eg. SSDeep) will combine it with Levenshtein.

On the other hand, a Spamsum digest consists of two halves: one left half at 70 characters — one right half at 29 characters. The right half has twice the boundary number of the left half. This approach seems redundant when comparing plaintext files rather than e-mails (as long as the plaintext files are not deliberately written to confuse the hashing scheme or the classifier), due to the fact that this approach also means that each half needs to be compared individually to the other files in the set, doubling the time used for comparisons. A 64 character digests therefore seemed sufficient for comparing plaintext files. If the 64 character fingerprint is full in NFHash, it will go back to the first character in the fingerprint and here on only update every 4. character. Should the same thing happen again, it will update every 16. character. The sliding window will of course not be sent back to the start in the input, but will continue. As such, setting the boundary number too low in NFHash can also cause bytes early in the input to have a higher vote than bytes later in the input.

Table 17: NFHash compared to Nilsimsa and Spamsum

	Nilsimsa	Spamsum	NFHash
Mechanism	Accumulators	Rolling Hash	Rolling Hash
Rolling Hash	NA	Adler's Checksum	CRC-64
Internal Hash	Trans53	FNV	Mersenne Prime-based
Base Number	16	64	64
Digest Size (chars)	64	107	64
Parts	1	2	1
Sliding Window Size	5	Adjustable (default 7)	Adjustable (power of 2)
Boundary Number	NA	Adjustable	Adjustable (power of 2)

4.7 Summary

- **NFHash** used CRC64 as a rolling hash and Mersenne primes as an internal hash
- The digest size is 64 characters, with a base64 number system
- NFHash is designed for file and text table classifications, and will outperform Soamsum and Nilsimsa on this
- It will, however, be less suited for blacklisting spam or for forensic uses

5 Design II: How Merkle Trees and non-Rolling Fingerprinting Schemes are Used in This Project

Merkle Trees in this project are used to detect changes or updates in a table. This datastructure was chosen to do the job because it will not waste CPU power by visiting similar nodes, and because nodes can represent clearly separated plaintext contect very well. Moreover, the performance of Rabin's fingerprinting scheme, means that the Merkle Tree and Rabin's fingerprinting scheme are a good match — leading to this scheme being chosen. The main point of utilizing the Merkle Trees in this project, was so that expensive brute-force searches would not be necessary.

In this project, 64 bit signed long integers are used for digests, since Rabin's Fingerprinting Scheme can also handle negative numbers (it is entirely possible to represent a polynomial with negative numbers, where negative numbers are larger polynomials). The approach used in this report also works for larger numbers. The only modification needed, is to use two digests per node (and thereby, two irreducible polynomials).

A datastructure with hashing in a hierarchical manner is necessary because a fuzzy hashing scheme is still one-way, and because the digest sizes are fixed — leading to the fact that in a large text table, a change in a single row might not even affect one byte in the fingerprint. Furthermore, for a large table, a single byte in the fingerprint can

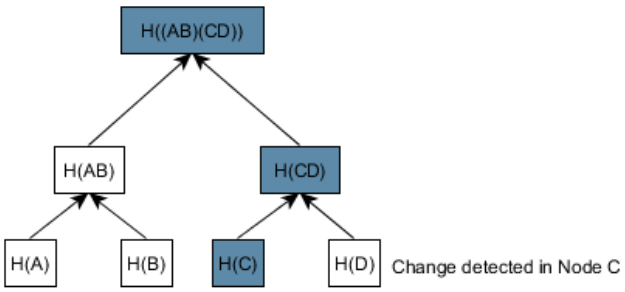
be based on more than one thousand rows, making a change in a single byte difficult to track.

The Merkle Trees in this project are constructed separately for rows and for columns, and every table has one of each. Every Tree Object in this project also has a Bloom Filter, that serves as a helping hand to get around some of the weaknesses of a Merkle Tree. This Bloom filter only holds bits from the fingerprints of each *terminal (leaf) nodes*, meaning that the Bloom filter will not take up much space.

In this project, the root node of the Merkle Tree can be considered a fingerprint for the entire file or entire table.

5.1 Comparing two Merkle Trees

Figure 21: How a change in node C will affect the traversal. The blue nodes represent nodes that are traversed, while the white nodes represent nodes that are ignored.



Comparison between two Merkle Trees can be classified as a recursive **divide and conquer** approach. Recursion will generally provide more clarity than an iterative approach, will in some cases be easier to debug, and might sometimes require less information to be stored in the computer's memory. When using a recursive approach, one does not strictly speaking have to keep anything else than the root node and some metadata (number of terminal nodes, depth, and so on) in the Merkle Tree code; the children nodes will simply be referenced internally by the parent nodes.

Since a Merkle Tree is traversed the same way as a binary tree, a comparison between two Merkle Trees can be considered a **depth first search** in level-order. The difference between the approach used in this approach and an ordinary DFS, is that in this approach, a node that is similar in both trees will not be traversed.

First, start with the root node of each tree. If these are identical, then clearly, the rest of the nodes are with a high probability as well, meaning that no further traversal is needed (best-case). If they are different, then the left and right child is checked in a similar manner in both trees. If both the left and right node are different in both trees, then proceed; if it is only different one one side, then just traverse this side. Both

children nodes cannot be different in both trees if the parent node is similar in both trees. Likewise, two identical nodes in both trees also indicate that their children are similar in both trees. Traversal stops when all terminal nodes that do not have a match in the Bloom filter of the other tree are pushed to a stack and counted. If no nodes are similar, then all nodes in both trees are traversed (worst-case). Because of this, best-case will have a complexity of $O(1)$, while worst case will have a complexity of $O(n^2)$. Average-case should in theory be $O((log\ n)^2)$. Here n is the number of rows or columns. If each cell was used instead, a worst-case comparison would be no faster than a pure brute-force search.

A possible alternative to the Bloom filter could be to do a DFS until the second last level — and then do a BFS (breadth-first search) on the terminal nodes below the nodes that have changes. The drawback of this approach is that it has poor worst-case performance, which is why the Bloom Filter-based approach was chosen instead.

It should again be noted that the Big O notation is not an exact meassurement of how long time something takes. Rather, it is a tool to show how an algorithm scales in terms of execution time; even in the worst-case, comparing Merkle Trees is s very quick operation.

Comparing Merkle Trees of different depths can pose a problem if the algorithm is not implemented properly. Initially, there was one obvious, but naive solution to this. Since a tree is basically nothing more than a collection of nodes organized in a hierarchy (i.e. apart from the fact that each parent node contains references to two nodes rather than one, it is no different from a Linked List), a tree can be split into smaller sub trees. If there is a depth disparity of n between two trees, the traversal algorithm would go down n levels in the largest tree from the root node — and start of at level n instead.

This initial approach splits the largest tree into 2^n sub trees, which would not lead to any information being lost, since only the terminal nodes contain the hash digests of the actual entries (the other nodes only contain hash digests of other hash digests). *The aforementioned approach is how the problem was initially done in this project.* After the Bloom Filter approach was added, trees of different sizes were traversed as usual, and if any of the digests of the terminal nodes in the largest tree was not referenced in the Bloom filter in the smallest tree, these were counted and pushed onto a stack. The current approach only increases the average- and worst-case performance minimally, while at the same time significantly increasing the average- and worst-case accuracy. The old approach is no longer present in the code.

Figure 22: Initial Approach: Comparing two Merkle Trees with a different number of nodes (a depth of 4 and 2 for each tree respectively)

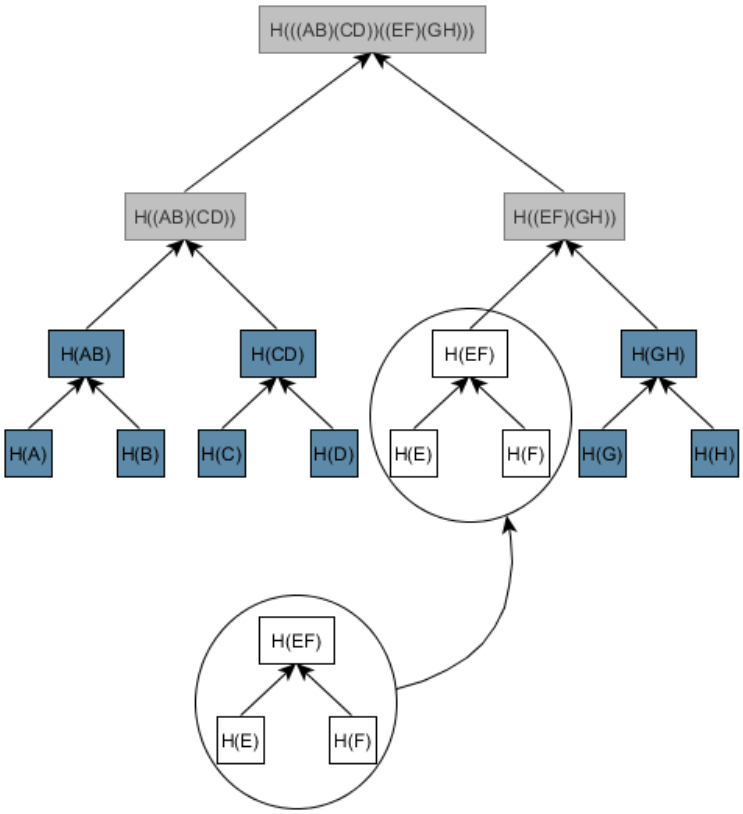
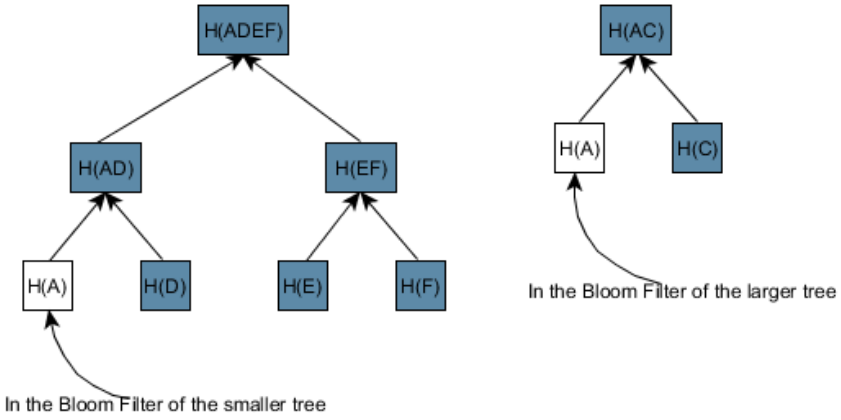


Figure 23: Final Approach when comparing two Trees of different depth: Bloom Filters are now used, so that the position of the terminal nodes are of lesser importance. Parent nodes that are similar (suppose both trees has $H(AB)$ or $H(CD)$ as a node) are still ignored



No dynamic programming is needed to implement methods for comparing two Merkle Trees, due to the fact that there are no overlapping subproblems — since it is impossible to traverse the same node twice in the same tree with a DFS. The principle of imposing restrictions on which nodes to visit and which to ignore is similar in principle to a bottoms-up dynamic programming approach, though.

Written as pseudocode, the entire algorithm can be summarized as follows:

Data: Two Merkle Trees
Result: The Number of Similar Nodes in Two Merkle Trees and their tags
initialization;
First Node of First Tree ← Root Node of First Tree
First Node of Second Tree ← Root Node of Second Tree
if *First Node of First Merkle Tree = First Node of Second Merkle Tree* **then**
 | Count=0
end
while *First Node of First Tree ≠ Second Node of Second Tree* **do**
 | Get children of both root nodes and compare them. Repeat process in every node with no identical equivalent **if** *Terminal node is reached and no equivalent found in other tree's Bloom Filter* **then**
 | | Count ← 1
 | **end**
end

Algorithm 3: Comparing two Merkle Trees

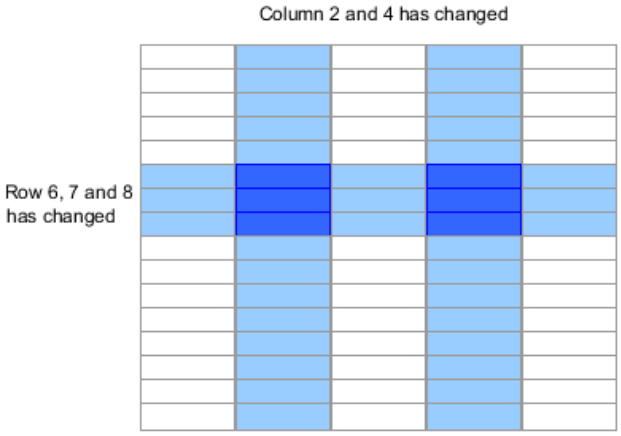
The Merkle Tree approach implemented during this project also features a method for calculating the Jaccard index, as mentioned in page 31. Since the nodes are sorted based on their respective fingerprints, this will not lead to the same weaknesses as when using

Jaccard for matching strings — due to the collision resistance of Rabin's Fingerprinting Scheme and the fact that a *Table Y* — which is a shuffled version of *Table X* — still has the same content as *Table X*. Jaccard deals with similarities, while the algorithm for comparing Merkle Trees deals with dissimilarities. The number of similar nodes is equal to the number of terminal nodes minus the number of dissimilar nodes between two trees.

The Jaccard method returns a double in this Java code. Since it is only calculated once per Merkle Tree, the extra costs of using a double instead of an integer are almost neglible. The main advantage of using Jaccard, is that it will put a coefficient the result, to make it more comprehensible to the human eye.

5.1.1 Finding Out Where the Changes Happened

Figure 24: An example of how it can be applied in a table with 5 columns and 16 rows. Changes are first discovered in row 6, 7 and 8 — and then secondly in column 2 and 4. Using Boolean logic, this means that [6,2], [6,4], [7,2], [7,4], [8,2] and [8,4] are candidates for where the change happened.



Since ordinary hash functions are indeed one-way functions, it is impossible to tell where the change happened. For a large file, this can be troublesome enough with a fuzzy hash, due to the fact that these are still very small compared to the vast majority of files (NFHash has a fingerprint size of 64 bytes).

In hash trees, this problem can be solved easily, by labelling each terminal node a tag in addition to the fingerprint. This does not have to be any more advanced than labelling them with an integer based on the row or column number. As the changes are detected, this tag can be added to an array or pushed to a stack. When the comparison is finished, the array or the stack is returned. The implementation in this library will both count the differing nodes and return their tags in a stack, so that the user will know where

the changes happened. Note that the one-way properties of hash functions still makes it impossible to find out *what caused the change*.

5.1.2 A Slightly Different Application of Bloom Filters

Recall Equation 19 in page 38 — together with the rule of thumb on bits per element. Here, a slightly different approach is used. The bit set used to implement the filter in Java, only allows digests that are no larger than what a 32 bit signed integer can hold. Since the digests from the Merkle Tree are used (to prevent redundant hashing — and to make use of Rabin's Fingerprinting Scheme) in the Bloom Filter as well, the 64 bit long integers from the terminal nodes are split into two 32 bit integers and used as two different digests per element. Moreover, 64 bits (the entire initial hash) per element is used in the Bloom Filter. The Bloom Filter is supposed to serve as a helping hand to the Merkle Tree (and not the other way around). This approach conserves a lot of CPU power (allowing for very fast comparisons) — and only adds a slight memory overhead, due to the fact that the Merkle Tree itself is much less space efficient than the Bloom Filter — almost regardless of size.

The number of digets k can therefore be summarized as 2, while $\frac{m}{n}$ becomes 32.

5.1.3 Merkle Trees Have One Significant Drawback With no Obvious Solution

A deterministic algorithm cannot be "reasoned with" — and will therefore have a fixed behaviour. This also means that a Merkle Tree in it's rudimentary form can be tricked easily.

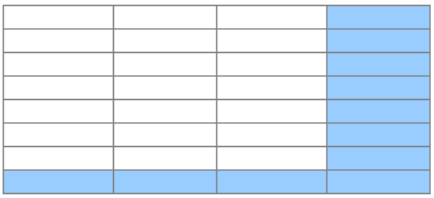
Imagine a 10×100 table. If one new column is inserted and one new row is inserted, all the fingerprins for each row and each column will be changed into a set of completely different fingerprints. Any matching algortihm, whether it is Levenhstein-Damerau, brute-force or Jaccard, will think that all entries have been changed, while in reality, only 111 entries (just shy of 10% of all the entries) have changed. This of course only applies to cases where an entire row and an entire column has been inserted (or all entries in a row or a column has been changed).

There are many slow solutions to this, but none of them are faster than an ordinary brute-force search. The first solution to this problem is to use fuzzy hashes, while hashing rows only. This is still easy to implement, but the fact that Damerau-Levenshtein is a lot slower than just comparing two hash digests by content, means that the string matching algorithm might become a bottleneck. As previously mentioned, it will also take much longer time to create a Merkle Tree with this approach, due to the fact that fuzzy hashing schemes are much slower than conventional hashing schemes. *Generally, the point of Merkle Trees is to detect changes faster than with a brute-force approach; if the*

creation or the traversal of these Merkle Trees become slower than this, then there would be no reason for them to exist.

An insert of one or more rows will change all columns, and likewise, an insert of one or more columns will change all rows. *However, if one or more rows are inserted without inserting any columns — or one or more columns are inserted without inserting any rows, it is still fairly straightforward to see where the change happened.* Clearly, if all rows are different, but only one row at position x is different, this indicates that a new row was inserted at position x. The same approach also holds water for columns; if all rows were changed, but only one column at position y was changed, then obviously, a column was inserted at position y.

Figure 25: Inserting a row and a column into a table (blue color). If a "common" hash function with avalanche properties is used, a string matching algorithm will think all cells have been changed.



5.1.4 Recognizing Rows that Have Been Shifted

An issue that could potentially render the Merkle Tree useless if not taken care of, is if there is a shift in the table — for instance if a row or column is inserted other places than at the end of the document. Special care has been taken to ensure that the Merkle Tree is in fact alignment robust.

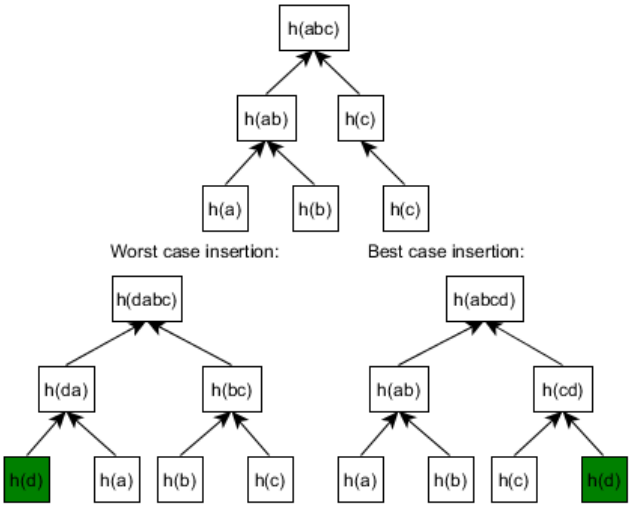
There are of course other ways to do a comparison of two or more Merkle Trees, with the most obvious one being Levenhstein-Damerau on all the terminal nodes together as a string, but this will cause the Merkle Tree to lose it's main advantage, as this would require too much resources. *The best way to solve this is to sort the nodes based on their hash values before constructing the tree.*

If the nodes are tagged (eg. with row or column numbers) in addition to their hash values, it does not matter if the nodes are not in the same order as the rows or the columns in the table the tree is based on.

In the worst-case scenario, a newly inserted row or column is placed furthest to the left after sorting the entries used to create the terminal nodes. This will shift all the other nodes to the right. This is the primary reason for why each tree is also equipped with a Bloom Filter. Without the Bloom Filter, a shift to the right at terminal node x,

would also cause all nodes to the right of and including terminal node x to be counted as changes, regardless of whether they had a match in the other tree or not.

Figure 26: Worst-case insertion compared to best-case insertion of a row or a column when represented by Merkle Trees



5.2 What about other uses than tables?

The simplicity of the Merkle Trees, means that they have several uses. While this project deals with hashing table entries, Merkle Trees can also be used in detecting changes in general plaintext. A simple way to do this would be to split a document into many fixed size chunks. Each chunk would then become a terminal node. This might make it easy to roughly tell where a change was made, but it is still pretty much useless for inconsitency detection. Moreover, it can make it troublesome if new sections, paragraphs or even sentences are inserted (i.e. no alignment robustness). There are of course several ways around this; if one only needs to detect changes on a course-grained level, it is enough to be able recognize sections and henceforth assume that anything below the header is part of that given section, until the next header starts.

This is very easy to do with a Latex document. First, the entire document can be read into a string and the whitespaces trimmed. All sections start with `\section{}`. One can then hash each section and add it into an array (the implementation in this library uses ArrayLists for simplicity), before creating a Merkle Tree from this array. PDF parsers are also improving, which means that text can be extracted from a pdf. Here, it is necessary to look for "signs" of a new paragraph (such as larger fonts), since the parser will still not decompile it into a *Latex* document or an *MS Office* document.

5.3 Limitations

In terms of data fingerprinting, the use of Merkle Trees should be limited to data with clearly defined delimiters, breakpoints and so on — or data that can easily be broken into fixed-size chunks.

6 Merging the Two Approaches in This Project

Both NFHash and the Merkle Tree-based approach have different strengths — and different weaknesses. Since both are designed for good performance, they can also be merged into one solution.

The most efficient way to do this is to first have NFHash deal with the inputs, then classify it — and finally have the Merkle Tree-based approach take over, by detecting where the changes happened, and by finding the Jaccard index on row-wise and column-wise basis inside each class ω_i.

7 Chapter Summary

This chapter has introduced two designs. The first one is called No-Frills Hash, and is a self-designed fuzzy hashing scheme utilizing CRC-64 as a rolling hash. The other design is based on Merkle Trees, and will be used to tell where the change in a table happened, by hashing each row and each column into two separate trees, where the fingerprints are placed in a hierarchy. The latter design also makes use of Bloom Filters to achieve it's goal. The trees are compared with a DFS, and every node not having a matching entry in the Bloom Filter belonging to the other tree, will be pushed onto a stack and counted as a change or an update.

Bloom Filters are a little similar to hash functions, but only stores a few bits from the elements, not the elements themselves or their whereabouts. It is useful for telling whether an element exists in a dataset or a different datastructure.

Fingerprints from NFHash are classified with k-NN, which will utilize Damerau-Levenshtein to calculate the edit distance (cost of transforming one string into another) between a given plaintext table and other plaintext tables used to train the classifier. This in turn makes it possible to classify this given plaintext table accordingly.

Chapter IV will focus on how this is implemented in Java, as well as what has been done to optimize the library. Furthermore, it will also show how the different parts of the library can interact when used together.

> You might not think that programmers are artists, but programming is an extremely creative profession. It's logic-based creativity.
>
> John Romero

Chapter IV
Implementation and Problem Solving

While the previous chapter dealt with the design phase of the project, this chapter deals with the implementations, the connections between the classes and the problem solving. The majority of this chapter consists of code listings in Java — and the corresponding explanations. Definitions that are difficult to implement in a conventional language or without very large figures (such as the difference between CRC and Rabin's Fingerprinting Scheme), will hopefully be less blurry here as well. Due to the fact that all implementations are explained theoretically in the previous chapter, this chapter will be fairly brief and straight to the point.

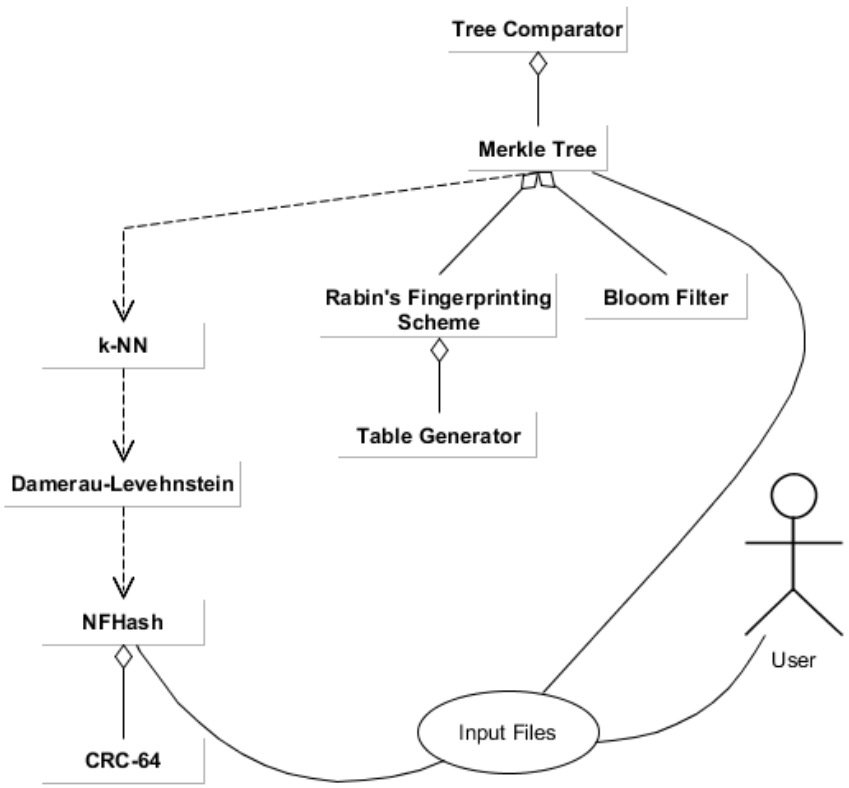
This chapter can also serve as a precursor to the user manual in Appendix A, and as an additional layer of documentations beyond the comments in the Java code. Readers that are not interested in code and microoptimizations after reading the previous chapter, may skip this chapter, as it is mostly intended for the thesis committee.

1 Intermodular Interaction

An overview of the intended usage of entire library can be seen in figure 27. The Merkle Tree will work fine without the k-NN classifyer, and thus, it is not directly dependent on it. For best results, it is better to track changes in files that are already classified, however.

The classes in this project are coded to be modular, so that for example one can use just a few modules without worrying about any dependency issues. To prevent large and bloated classes, tree traversal and the Jaccard calculations are placed in a class called *TreeComparator* — rather than in the Merkle Tree class itself. A complete Java code can be found in Appendix B.

Figure 27: UML class diagram of the entire project



2 Matching Digests Based on Damerau-Levenshtein Distance

The method for calculating the Damerau-Levenshtein Distance in this project is based on a matrix with the dimensions $(N+2) \times (M+2)$, where N is the length of the first string and M is the length of the second string.

Damerau-Levenshtein Distance

```
1     public static int DamLev(String first, String second){
          final int init = 0;
3         int[][] dist = new int[first.length()+2][second.length()+2];
          dist[0][0]=init;
5
          for(int i = 0; i<=first.length(); i++) {
7             dist[i+1][1] = i;
              dist[i+1][0] = init;
9         }
          for(int j = 0; j<=second.length(); j++) {
11            dist[1][j+1] = j;
```

```
            dist[0][j+1] = init;
13      }

15      int[] DA = new int[256]; // 256 printable characters
        Arrays.fill(DA, 0);
17
        for(int i = 1; i<=first.length(); i++) {
19          int DB = 0;
            for(int j = 1; j<=second.length(); j++) {
21              int i1 = DA[second.charAt(j-1)];
                int j1 = DB;
23
                int d=0;
25
                if (first.charAt(i-1)==second.charAt(j-1)){
27                  d=0;}
                else d=1;
29
                if(d==0) DB = j;
31              dist[i+1][j+1] =
                        Math.min(Math.min(dist[i][j]+d,
33                              dist[i+1][j] + 1),Math.min(
                                dist[i][j+1]+1,
35                              dist[i1][j1] + (i-i1-1) + 1 + (j-j1-1))); //
                                    This line separates Damerau-Levenshtein
                                    from Levenshtein
            }
37          DA[first.charAt(i-1)] = i;
        }
39      return dist[first.length()+1][second.length()+1];
    }
```

This is done in accordance with Table 11. This approach is both simple and offers good readability. Furthermore, it is easy to debug, platform neutral, and easy to port into any other C-like language. Total running time becomes $O(M \times N)$.

The final distance is returned as the lower right corner of the matrix `dist`.

To compare two digests from NFHash, simply pass them into the method as two strings. The difference between Levenshtein and Damerau-Levenshtein can be seen in line 36 (`dist[i1][j1] + (i-i1-1) + 1 + (j-j1-1)));`).

3 Optimizations on all Levels

A poorly optimized code can look very similar to a well-optimized code. Most modern compilers in any C-like language will do their best to optimize the code, allowing the programmer to think about other things. Therefore, readability and performance no longer have to be mutually exclusive. In the past, ternary conditions in C-like languages were faster than common if/else statements. Today, a modern compiler will treat

these in a similar manner. Clearly, it is easier to read Listing 3 than `int number = condition ? new value : other new value` — especially if it is surrounded by more code. Code is often worked on by many people, and according to the open/-closed principle (part of the SOLID principles), a class should be open for extension. There will be some terniary code in the NFHash class to keep the class short, but this will be explained carefully with comments.

Terniary conditions no longer offer many advantages in terms of optimizations

```
1 if (condition is true) {
  Do something.
3 } else {
  Do something else.
5 }
```

In the past, statements such as final (denoting that a value is never changed) would speed up the code, since the compiler would not assign any unneeded resources to a value that would not change. Today, the compiler will check if the value is changed in the code or in any other class using the value, optimizing it accordingly. It might be better for the sake of readability to use it, though. This section will focus on things the compiler cannot optimize on it's own.

3.1 Overview of the CRC-64 and Rabin's Fingerprinting Scheme Class

Both CRC-64 and Rabin's Fingerprinting Scheme have a lot in common. To keep the library clean and orderly, they nevertheless have their own classes. The former is based on the CRC-32 class already present in the `java.util.zip` package, and the only reason why it was created, is that there is no CRC-64 class already existing in any stock Java library — and so that the project would not be directly dependent on any 3. party library. Thus, it is implemented according to the RFC-1952 implementation of CRC-32 — but with the ISO-3309 polynomials. Rabin's Fingerprinting Scheme in this project is based on Andrei Z. Broder's implementation, including the lookup tables. Details of both algorithms are described in detail further on in this section.

3.2 Boolean Logic Revisited

As mentioned in **Chapter I**, Boolean operations are typically much faster than common arithmetics when doing divisions or finding the remainder. Most algorithms utilizing rolling hashes, such as **Spamsum** and the **NFHash** prototype presented here will only slide one byte at a time (a necessary evil for accuracy). Even for a CSV file of a few hundred kilobytes, a lot of performance gains can therefore be achieved by implementing the rolling hash function in an efficient manner.

In the first section in Chapter II, it was shown that to most architectures, the modulus operand is a costly operation. This can be further proven by x86 assembly code, which is ultimately what all C-like programming languages are translated into before being converted to binary numbers.

The example below is taken from NFHash, line 55, which initiates the input value sent into both the rolling hash and the internal hash. This can either be written as `int character = (in[i] + 256) % 256;` or `int character = (in[i] + 256) & 255;`. This example assumes that `in[i]+256` is equal to 281, but in the code, this can be any number. 281 was chosen because it is a prime number.

How an x86-64 CPU Does Division

```
1 mov edx, 0          ; reset edx register
  mov eax, 0x119    ; move the dividend 281 into the 32 bit register eax
3 mov ecx, 0x100     ; move the divisor 256 into the 32 bit register ecx
  div ecx             ;  eax = 0x54, edx = 0x001
```

The register *eax* is an accumulator register, used for arithmetic operations. The quotient can after division be found in the eax register, while the remainder can be found in the edx register. Registers with the "e" prefix are used for 32 bit integers. In Java, all primitive numbers are signed, meaning that an integer uses 32 bits or 4 bytes for both positive and negative number (16 bits for each). The most expensive operation here is the *DIV* operation.

How an x86-64 CPU does a Boolean AND

```
6 mov eax, 0x119    ; move the dividend 281 into the 32 bit register eax
  and eax, 0x0FF     ; AND the register eax with 256-1
```

The result will also be in the EAX register after the Boolean AND shown in Listing 5. Not only does this lead to fewer instructions for the CPU, but it also avoids the expensive DIV instruction completely.

A good estimate of how well an instruction performs and scales, can be done by micro-operations (hencefort denoted as *µops*). One differentiates between *µops* for the *fused domain* and *µops* for the *unfused domain*. The former deals with issuing *µops* for an operation, while the latter deals with retiring (or freeing) *µops*. Multiplications and divisions in a CPU are not microoperations themselves, but are represented by additions coupled with shifts and subtractions coupled with shifts, respectively. AND operations often only require just one *µop* for issuing and one *µop* for freeing on Intel's Skylake architecture [50], making them much more efficient to implement.

3.3 Big Integers are not Primitives

Java comes with a builtin BigInteger library and the same goes for C# (furthermore, many open-source implementations of this datastructure are available for C and C++).

This allows for numbers larger than $2^{32}-1$ — without causing overflow. From a technical point of view, Big Integers are not primitives. Rather, they are datastructures implemented with strings and arrays. Since Big Integers are abstract data types, there is no strict or "correct" way to implement them. In Java, big integers are represented by an integer array, where the most significant bit is represented by the 0. element. [51]

BigIntegers can indeed be useful in cases where performance is not crucial, such as public key encryption, online calculators and so on. For fingerprinting, however, they are too slow to be a feasible alternative to integers or long integers.

For the reasons mentioned above, they are not used in NFHash, nor in the Merkle Tree-based approach (or any classes used by these two implementations).

3.4 Lookup Tables Speed Things Up Significantly — Even if it Leads to Larger Classes

When it comes to fingerprinting, an algorithm is only as good as it's performance. One good way to speed things up is to use lookup tables. While lookup tables will lead to more code and larger classes, they will deload some work from the CPU, due to the fact that much the work is already done. More importantly, it will significantly decrease the overhead — which is important when fingerprinting multiple tables rather than just one large table.

Lookup tables can either be generated when the object is initialized — or they can be hardcoded. Which approach to take will of course depend on what kind of algorithm it is. These lookup tables do not contain hashed values, but partial seeds, meaning that the CRC digest can be calculated the following way:

How CRC-64 is implemented in this library (the lookup table can be found in the appendix). This is mostly based on the existing CRC-32 class in the `java.util.zip` package

```
@Override
    public void update(final byte[] bytes, int offset, int length) {
        final int end = length - offset;
        long crc = 0L;
        while (offset < end) {
            crc = lookup[(bytes[offset] ^ (int) crc) & 0xFF] ^ (crc >>> 8);
            offset++;
        }
        this.crc = crc;
    }
```

Rabin's Fingerprinting Scheme is designed to make use of different irreducible polynomials — hence it makes more sense to generate a table when the object is initialized. This table can later be used to hash more files, making the "marginal cost" of hashing another file or another chunk very low.

It is important that the lookup table is small, yet at the same time that it is large enough to actually be useful. In the CRC-64 class, the lookup table has 256 entries, and is based on the ISO polynomials as previously mentioned. In Rabin's Fingerprinting scheme, it is not hardcoded, but generated based on an irreducible polynomial sent into the constructor of the class "TableGenerator". It can then be generated by the method `generateTable()`. Here, the table size is 256×4 (in accordance with Broder's implementation). In both algorithms, the table is small enough to fit into the level 1 cache. The main virtue of using a lookup table, is that calculations can now be done byte-by-byte rather than bit-by-bit. [52] In Broder's Implementation, it is done chunk-by-chunk, each chunk consisting of four bytes.

The code for Rabin's Fingerprinting scheme (which is closely related to CRC) is found in the appendix, due to the fact that the class is longer and more complex than the CRC-64 class.

How the table generator for Rabin's Fingerprinting Scheme works

```
public class tableGenerator {

    private transient static long[][] superTable;

    public long[][] generateTable(long prime) {
        long X_degree = 1L << Long.bitCount(Long.MAX_VALUE);

        final long[] preTable = new long[64];

        preTable[0] = prime;
        for (int i = 1; i < 64; i++) {
            long poly = preTable[i - 1] << 1;
            if ((preTable[i - 1] & X_degree) != 0) {
                poly ^= prime;
            }
            preTable[i] = poly;
        }
// superTable is the name of the lookup table returned
        superTable = new long[8][256];

        for (int i = 0; i < 256; i++) {
            int c = i;
            for (int j = 0; j < 8 && c > 0; j++) {
                if ((c & 1) != 0) {

                    superTable[0][i] ^= preTable[j];
                    superTable[1][i] ^= preTable[j + 8];
                    superTable[2][i] ^= preTable[j + 16];
                    superTable[3][i] ^= preTable[j + 24];
                    superTable[4][i] ^= preTable[j + 32];
                    superTable[5][i] ^= preTable[j + 40];
                    superTable[6][i] ^= preTable[j + 48];
                    superTable[7][i] ^= preTable[j + 56];
                }
```

```
                    c >>>= 1;
36              }
            }
38          return superTable;
        }
40
        }
```

3.5 Possible Advantages of Rewriting the Library in C

The Java Virtual Machine can run C code by compiling it to bytecode (just like it does with Java code). This is done via the Java Native Interface (JNI) framework. Due to the fact that it still runs on the same virtual machine (contrary to popular belief, a virtual machine that is not an emulator will not add much overhead [53]), there are normally no advantages to doing this. However, one potential advantage C could have over Java in this case, is that Java does not allow unsigned integers or unsigned long integers; these are readily available in C (denoted as `uint` and `ulong`, respectively). In Rabin's Fingerprinting scheme this does not matter (negative numbers work just as well for implementing polynomials as positive numbers), but in NFHash it might give a modest performance gain, due to the fact that the internal hashing algorithm deals with positive numbers (an unsigned integer only requires half the memory space of a signed long integer).

4 Implementing the k-NN Classifier

Recall that decision regions, decision boundaries, hypercubes and so on are for human eyes only. Therefore, a k-NN classifier can be implemented with the help of some ArrayLists holding more ArrayLists (where each outer ArrayList represent classes and inner ArrayLists represent vector number), two for loops which will iterate through this array when calculating the edit distance from the input, and a function for calculating the discriminant functions coupled with a for loop. This is of course a lot slower than the hash function itself is, but fortunately, the digests are no more than 64 characters.

k-NN discriminant function

```
    private static double discriminant(double knn, double editDist, double
        numClass){
2
            if (editDist>0){ // Do not divide by zero
4           double prior = numClass/(double)count;
            double top = knn/numClass;
6           double pnx = top/editDist; // nth estimate of p(x) (PDF)
            double disc = prior*pnx;
8            return disc;}
            else return Double.MAX_VALUE;
```

```
10      }
    }
```

The code above shows the discriminant function. Here, `prior` is the prior distribution, `top` is the divisor of the n-th estimate of the PDF, `pnx` is the PDF and `disc` is the discriminant function. Unfortunately, there is not way around doubles or the division operand in this case. One could potentially multiply the inputs with 100 and treat them as integers, but this would cause inaccuracies when there is a remainder — and would not solve the division problem. There is an if/else statement here, due to the fact that the Damerau-Levenshtein distance in some cases might be 0. If this distance is zero, it must necessarily mean that a file is being compared against it's own duplicate in the dataset. Therefore, the disciminant function returns the maximum double value, to ensure that the file is more likely to be placed in the right class.

The rest of the k-NN code can be found in Appendix B.

5 The Merkle Tree and it's Uses

The Merkle Tree is initialized by passing an ArrayList with fingerprints, an ArrayList with tags and an instance of Rabin's Fingerprinting Scheme into the constructor. Both Array Lists are then sorted according to the Array List with the fingerprints. This can be seen from the code below:

Merkle Tree constructor

```
1   public merkle(ArrayList<Long> fingerprints, RabinSingle rs, ArrayList<
        Integer>tagList) {
            this.tagList=sortTags(tagList, fingerprints);
3           this.fingerprints=fingerprints;
            this.rs = rs;
5           terminalnodes=leafSignatures.size();
            bf=new bloomFilter(512,terminalnodes);
7           create(leafSignatures);
        }
```

Sorting one list according to another

```
    private ArrayList<Integer> sortTags(ArrayList<Integer> results,ArrayList<Long
        > digests){
2           int tmp2;
            long tmp;
4           for (int k=0; k<digests.size()-1; k++) {

6               Boolean isSorted=true;
                for (int i=1; i<digests.size()-k; i++) {
8
                    if (digests.get(i)>digests.get(i-1)  ) {
10                      tmp=digests.get(i);
```

```
                        digests.set(i,digests.get(i-1));
12                      digests.set(i-1,tmp);

14                      tmp2=results.get(i);
                        results.set(i,results.get(i-1));
16                      results.set(i-1,tmp2);

18                      isSorted=false;
                    }
20              }
                if (isSorted) break;
22          }
        return results;
24
    }
```

This aforementioned method returns an ArrayList with the tags — sorted according to the digests. The reason why this approach to sorting was chosen, was due to the fact that sorting the nodes directly will lead to a higher overhead.

The class `merkle` has the inner class `node`. This is implemented in a very compact manner, as it is only intended to hold a few primitives, plus references to it's children nodes.

Note that the ArrayList representing the fingerprints for the terminal nodes need to be hashed "externally", that is outside of the Merkle Tree, before being sent into the constructor. The recommended approach for doing so is by using the enclosed Rabin's Fingerprinting Scheme, which will also be used by the Merkle Tree when creating digests for the parent nodes.

How the class representing nodes appears

```
1   public static class Node {
            public byte type;  // Terminal or not?
3           public long fingerprint; // Digest
            public Node left;
5           public Node right;
            public int tag; // ID
7           public int depth;
        }
```

Avito Loops retrieves entries from databases either cell-by-cell, row-by-row or column-by-column. If plaintext files are used instead, it might be an advantage to sort each column based on cell contents before hashing. If a the order of the rows is shuffeled, the traversal algorithm will not consider it as a change. However, since hash functions typically exhibit avalance properties, a column on the form $[adcb]^t$ will look different from a column on the form $[abcd]^t$ after hashing. This highlights why sorting the cells in a column by content might be a good idea before hashing.

5.1 Traversing the Merkle Tree in a Recursive Manner

Traversal is found in a different class named `treeCompare`, not in the class `merkle`. This was done to make the library more modular, more convenient to study or work with, and easier to debug.

Comparisons between two trees are done the following way:

Comparing Two Trees

```
public Stack<Integer> bloomCompare(merkle mfirst, merkle msecond){
      equal=0;
      nodeStack=new Stack<Integer>();
        if (mfirst.root!=msecond.root){

        int t1=mfirst.nnodes;
        int t2=msecond.nnodes;
        merkle referenceTree=mfirst;
        bloomTree=msecond;
        nodeStack= new Stack();

        if (t2>t1){
            referenceTree=msecond;
        bloomTree=mfirst;
        }
        bloomTraverse(referenceTree.root.left,bloomTree.root.left);
        bloomTraverse(referenceTree.root.right,bloomTree.root.right);
        }
        return nodeStack;
    }

private void bloomTraverse(merkle.Node root, merkle.Node root2){
        if (root!=root2)
        {
        if (root.left!=null && root2.left!=null)
            bloomTraverse(root.left,root2.left);
        if (root.right!=null && root2.right!=null)
            bloomTraverse(root.right,root2.right);

        if (root.left!=null && root2.left==null)
            bloomTraverse(root.left);
        if (root.right!=null && root2.right==null)
            bloomTraverse(root.right);

        if (root.type==0x00 && !bloomTree.bf.contains(root.sig))
        {
        nodeStack.push(root.tag);
        equal++;
        }}
    }
```

Note that `bloomTree` is simply the name given to the largest tree. Comparisons and traversals are done in separate methods to keep the methods compact and easy to deal

with. `bloomTraverse` is a recursive method, so that less information is required to be stored in the Merkle Tree, and that the working mechanisms are easier to illustrate.

The comparison only returns a stack containing the tags (IDs) of each note rather than the nodes themselves directly. Furthermore, it also causes the integer `equal` to increment. This integer is public, so that it can be retrieved in classes elsewhere in the library. The stack is used here instead of an array, due to the fact that insertion and retrieval is an $O(1)$ (constant speed) operation. When the tags for all the nodes are going to be returned anyway, there are no advantages to returning an array or a list structure instead.

6 A Brief Overview of NFHash

The entire code for NFHash is too long to be included other places than in the appendix. Nevertheless, the code will still be explained here.

NFHash is initialized by passing two integers into the constructor. These are (in this order) the sliding window size, and the boundary number. Digests can then be generated by passing a byte array into the method `hashBytes(final byte[] in)`. This method returns a 64 character base64 encoded string, which will serve as the fuzzy fingerprint of the inputted byte array.

The rolling hash uses an instance of the `CRC64.java` class, while the internal hash function based on Mersenne Primes is located directly in the NFHash class.

7 Chapter Summary

The main point of Chapter IV has been to explain Chapter III briefly by using Java code, and by showing how the different modules interact. A complete set of code, a user manual, and how to implement a simple classification tool based on the enclosed code can be found in the appendix. Certain things are difficult to explain by simple comments only, as such, this chapter has hopefully brought some more clarity to parts of the design.

As seen in this chapter, lookup tables do not take up that much space — and will lead to a much faster processing speed. Moreover, things that might seem insignificant (eg. Boolean operands) can make everything more efficient if used correctly. This chapter has also explained why (practical as the may be in other uses) Big Integers are not used.

The next chapter will put everything described in this chapter and chapter III to use in terms of testing. This will demonstrate among others the performance gains made with Mersenne Primes and lookup tables. Moreover, it will demonstrate just how well

the Merkle Tree and the Bloom Filter will work together for tracking changes in a table.

> To err is human — and to blame it on a computer is even more so.
>
> Robert Orben

Chapter V
Testing, Analysis, and Results

While the two previous chapters have described the two designs and how they are implemented, what they will be used for and what their strong and weak points are, this chapter will put them to practical use.

Some minor scale unit testing was done already at the start of the project, along with the literature review, to ensure that there were no dead-ends or red herrings in the project. These tests are not featured here, due to space constraints, and due to the fact that they do not give a clear and concise picture suited for a thesis.

The tests here are the tests done after the library was more or less complete, to ensure that the hypotheses were mostly true, and to prove that the two approaches would perform as well in practical applications as they did on a theoretical level.

This chapter focuses on testing with plaintext tables in the CSV format. Excel files in the format XLSX are basically archives holding many XML tables, and can be used as well for the classifier if the contents are extracted from them. Furthermore, if parsed appropriately, they can also be used to construct Merkle Trees. Due to time constraints, this was not done here.

There are three sections in total in this chapter. The first concerns the performance of NFHash compared to Spamsum and Nilsimsa; the second deals with NFHash coupled with the k-NN classifier, while the last one will show the capabilities of the Merkle Tree-based Implementation.

1 Performance Tests

The performance tests were set up in jUnit inside IntellijIDE. Due to the fact that the testing was done with many background processes running, it was not possible to test how many MB per second each hash function was capable of. Moreover, finding open-source libraries for doing this proved difficult.

Because of this, the performance tests meassure how each algorithm scales, and how fast they are in relation to each other. The times were logged with the builtin function in jUnit. The times shown in the graphs are based on an average after ten runs.

1.1 Testing Performance with Multiple Files

A good meassure of an algorithms overhead resource usage, is to have it do many small tasks instead of one large task. The first performance test will have Spamsum, NFHash and Nilsimsa deal with 50, 100 and 200 consecutive 30 KB tables each, to see how the much time each algorithm uses — and how each algorithm scales. The sliding window approach means that none of the three algorithms could potentially scale better than $O(n)$. Nevertheless, to a lesser extent it can be used to detect overhead consumption.

1.1.1 Initialization

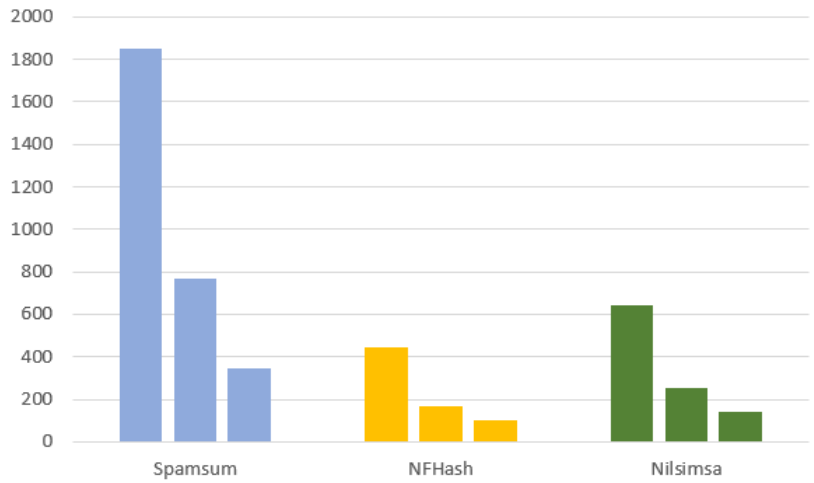
The first test was initialized by pushing the given number (50, 100 or 200, depending on the subtest) of CSV files to a stack holding byte arrays for each file.

The results given below are based on the average after 10 runs for each scenario (eg. 10 runs for 100 consecutive tables, 10 runs for 200, and so on).

The boundary number for Spamsum and NFHash was set to 512, while the sliding window size for both was set to 4 (this is fixed to 5 in Nilsimsa by default). Spamsum also ran without the boundary number guessing algorithm, which increased the performance somewhat.

1.1.2 Results

Figure 28: How the different fuzzy hashing schemes handle 50, 100, and 200 consecutive 30 KB tables. The times are given in ms



NFHash on average ran 43% quicker than Nilsimsa when considering all the scenarions

— and more than four times quicker than Spamsum — indicating that most of the performance goals have been reached.

The average scale factor in time consumption for each algorithm when doubling the number of tables, was 2.31, 2.19 and 2.13 for Spamsum, Nilsimsa and NFHash, respectively.

1.1.3 Analysis

In Figure 28, it can be seen that NFHash outperforms both of it's two main sources of inspiration. All of the algorithms scale fairly linearly, giving a running time of $O(n)$. The overhead is difficult to tell from this test, but NFHash might have a lower overhead than the other algorithms. Nevertheless, this test proves that NFHash is significantly faster than both Nilsimsa and Spamsum. Ultimately, the final bottleneck for all algorithms in terms of speed, is the same thing giving them their strength: The sliding window approach.

1.2 Testing Performance on Single Files of Varying Size

To get a better overview over how much of the performance gains are due to a decreased overhead consumption and how much was due to NFHash processing large inputs of information quickly, a second test was done. Instead of hashing multiple files at once, each algorithms instead was given progressively larger files, to meassure how much time each of them would spend on a single large file.

1.2.1 Initialization

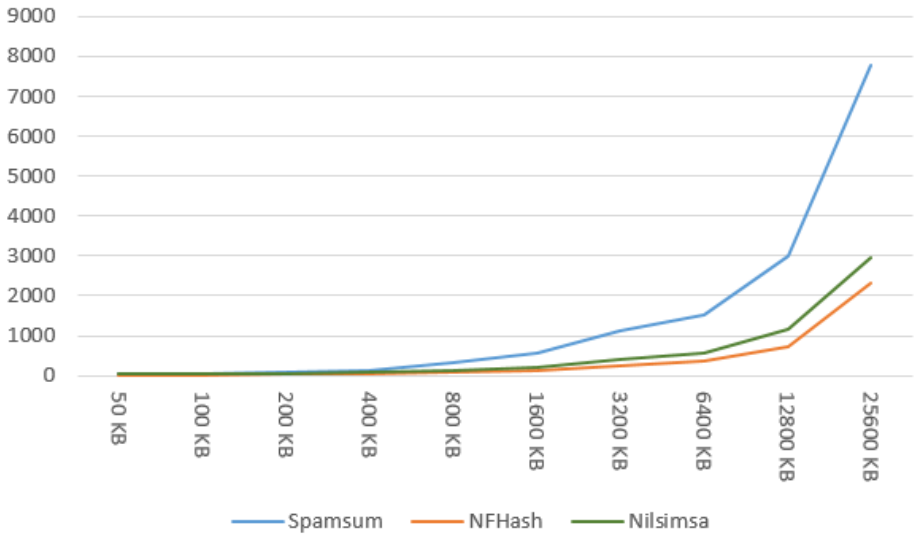
To get enough files of exact size (each file used in this test is exactly twice as big as it's predecessor), ten dummy files were created. These files contain nothing apart from white spaces (this was not trimmed away before the algorithms received the files), and were created with MyNikko.com's Dummy File Creator. The settings for Spamsum and NFHash in this test, were identical to the ones used in the previous test.

Each file size was tested ten times, before the average was calculated.

1.2.2 Results

Spamsum was on average 3.6 times slower than NFHash, while the ratio between Nilsimsa and NFHash remained the same. This suggests that while NFHash has a lower overhead than Spamsum, the overhead level is equal to that of Nilsimsa — thus meaning that the overhead is still significant.

Figure 29: Time used by each algorithm to hash a single file of the respective sizes. The times are given in ms



1.2.3 Analysis

This test shows that while NFHash is clearly the fastest algorithm, it nevertheless has a significant overhead (still lower than Spamsum). This was the only design goal that did not pass any of the tests. While NFHash does not have a lower overhead, it does not have a higher overhead either, implying that they are fairly equal in terms of overhead. Combined with the previous test, it can be concluded that NFHash is between 3.5 and 4 times faster than Spamsum — depending on the use. Compared to Nilsimsa, it is almost 40% faster — regardless of use.

2 NFHash and Classifications

While utilizing NFHash with machine learning has been shown to be theoretically possible in Chapter III, this test section will demonstrate that it is also a practically feasible solution when using it to match plaintext tables. For the formulas required to better understand this test section, refer to page 45.

A unit testing library was not used for this section. Instead, a GUI was made. This made the testing both faster and easier. Furthemore, mistakes were easier to correct.

These two tests will deal with randomly generated, yet realistic data. This section can be divided into two tests, one dealing with four clearly separable classes — the other one dealing with two classes that are difficult to separate. Both tests will illustrate

the importance of a fuzzy hashing scheme that is customizable — and of course how the problem of overfitting in k-NN can be tackled. Most importantly, however, the subtests will illustrate just how well suited a fuzzy hashing scheme and a stirng matching algorithm are to classify raw data from a spreadsheet.

In both tests, the initial hypothesis was that the classifyer would perform well on well-separated classes, but not so well on classes that were difficult to tell apart. Furthermore, part of the hypothesis was that the k_n number would be the most significant number — and not the sliding window size.

This hypothesis was only partially true, as will be demonstrated with the results.

2.1 Test Setups

In both tests, for each class, a total of 30 plaintext files with 1000 columns each were generated with the tool found on Mockaroo.com. Realistic data is important, due to the fact that in real life, this typically has a high variability (not the same as variance) and tend to be quite unpredictable. Furthermore, to get accurate results, enough sets are needed. These are time-consuming to create, and if created by a person for the purpose of testing, they might be biased in favour of the hypothesis.

The first test utilizes a boundary number in NFHash of 512 — while the second test utilizes a boundary number of 1024. The second test and it's corresponding subtests will test the training set again after classification to see if the classifier thinks any of the training samples are misclassified. If the classifier "thinks" that some training tables need to be reclassified, it might indicate a high variance (i.e. poor generality) with the current settings, making accurate classifications in the designated testing set very difficult.

While all subtests in each test had the same boundary number, the settingss k_n, number of training samples, and the size of the sliding window was adjusted to illistrate how the settings impact the result. Test I and Test II have respectively 3 and 6 subtests each.

Note that "Window Size" in these tests refer to the size of NFHash's sliding window, not the size of the one-dimensional "window" (representing the edit distance) containing the k_n number of samples.

2.1.1 Initializing Test I

The first test is based on clearly separable classes, without much in common, to illustrate that in clearly separable testcases, different settings have a less significant impact. The testcase has four different classes — all of which are very different. These classes are

$\omega_{transactions}$

Wallet	Amount	Currency	Currency Code	Type	Date
17Lj...	$74029.85	Dinar	IQD	diners-club-enroute	’8/3/2015

ω_{cities}

City	Country	Country Code	Time Zone
Tomdibuloq	Uzbekistan	UZ	Pacific/Auckland

$\omega_{employees}$

First Name	Last Name	Employer	Title
Phyllis	Williams	Trilith	Honorable

$\omega_{mailserver}$

E-mail Address	Last Used IPv4	Last Used IPv6	Last Used MAC Address
cbishop0@census.gov	44.26.167.198	8dcd:...	F9-60-F9-62-02-75

named $\omega_{transactions}$, ω_{cities}, $\omega_{employees}$, and $\omega_{mailserver}$. Tables from these classes look as follows:

The settings for each subtest can be seen in the table below:

Different settings for each subtest in Test I

Subtest	n_i	k_n	Window Size
A	20	11	8
B	10	3	8
C	3	1	8

2.1.2 Results from Test I

The results from each subtests in Test I are represented by confusion matrices. From these, the accuracy and the recall rate has been classified as well — so that there exists a normalized meassure indicating how well the classifier performs.

Subtest A

Accuracy:

$$AC = \frac{10 + 10 + 10 + 10}{10 + 10 + 10 + 10} = 1$$

		Actual			
		$\omega_{transactions}$	ω_{cities}	$\omega_{employees}$	$\omega_{mailserver}$
Predicted	$\omega_{transactions}$	10	0	0	0
	ω_{cities}	0	10	0	0
	$\omega_{employees}$	0	0	10	0
	$\omega_{mailserver}$	0	0	0	10

Recall rate:

$$RC_{transactions} = \frac{10}{10+0+0+0} = 1$$

$$RC_{cities} = \frac{10}{0+10+0+0} = 1$$

$$RC_{employees} = \frac{10}{0+0+10+0} = 1$$

$$RC_{mailserver} = \frac{10}{0+0+0+10} = 1$$

Subtest B

		Actual			
		$\omega_{transactions}$	ω_{cities}	$\omega_{employees}$	$\omega_{mailserver}$
Predicted	$\omega_{transactions}$	10	0	0	0
	ω_{cities}	0	8	2	0
	$\omega_{employees}$	0	0	10	0
	$\omega_{mailserver}$	1	0	0	9

Accuracy:

$$AC = \frac{10+8+10+9}{10+10+10+10} = 0.925$$

Recall rate:

$$RC_{transactions} = \frac{10}{10+0+0+0} = 1$$

$$RC_{cities} = \frac{8}{0+8+2+0} = 0.8$$

$$RC_{employees} = \frac{10}{0+0+10+0} = 1$$

$$RC_{mailserver} = \frac{9}{1+0+0+9} = 0.9$$

Subtest C

		Actual			
		$\omega_{transactions}$	ω_{cities}	$\omega_{employees}$	$\omega_{mailserver}$
Predicted	$\omega_{transactions}$	10	0	0	0
	ω_{cities}	0	6	4	0
	$\omega_{employees}$	0	0	10	0
	$\omega_{mailserver}$	0	0	0	10

Accuracy:

$$AC = \frac{10 + 36 + 10 + 10}{10 + 10 + 10 + 10} = 0.9$$

Recall rate:

$$RC_{transactions} = \frac{10}{10 + 0 + 0 + 0} = 1$$

$$RC_{cities} = \frac{6}{0 + 6 + 4 + 0} = 0.6$$

$$RC_{employees} = \frac{10}{0 + 0 + 10 + 0} = 1$$

$$RC_{mailserver} = \frac{10}{0 + 0 + 0 + 10} = 1$$

2.1.3 Analysis of the Results in Test I

All the subtests demonstrate a good accuracy, even in the case of subtest C, where only 3 training samples are used to train the classifier and only *the* nearest neighbor has any impact.

Recall rate is also good for almost all cases — except the recall rate for cities in Subtest C. Here, 4/10 test tables will be misclassified as $\omega_{employees}$ — instead of being classified as ω_{cities} where they belong.

The results indicate that proper configurations are less important with clearly separated classes, but at the same time, it also shows that good configurations still have a positive impact, since the recall rate still suffered significantly in the last subtest for one of the classes.

2.1.4 Initializing Test II

The second test matches tables that are not that easy to tell apart. These are classified as either ω_{wallet} or ω_{bank}, representing either Bitcoin wallet details or bank account details respectively. Many columns will look similar; three of them even contain exactly the same type of information in both table classes. Furthermore, three other columns in ω_{wallet}, "wallet", "Credit Card Number" and "Balance (BTC)", will still have counterparts with a strong resemblance in ω_{bank}, namely "Account", "IPv4" nad "Balance (USD)". A classifier without the proper settings should not be able to tell the differences apart.

The header and second row in a table from ω_{bank}, can be written as:

A small part of a CSV from the first class

```
First Name,Last Name,Account,Balance (USD),Credit Card Number,Credit Card
    Type,E-Mail
Andrew,Hanson,HU76 1253 8382 0373 6372 5504 8007,$357340.98,6333194148575331,
    switch,ahanson0@mozilla.com
```

For the ω_{wallet} class, a similar header and second row looks like this:

A small part of a CSV from the second class

```
First Name,Last Name,Wallet,Balance (BTC),IPv4,E-Mail
Thomas,Austin,13n7mxMNn8Hp4Mxcq8WC341ZkCgzxBMzu8,176.41,135.55.16.243,
    taustin0@bigcartel.com
```

The first table has 7×1000 entries, while the second one has 6×1000 entries. This test has six subtests, represented by their results in the next subsubsection.

Table 18: Different settings for each subtest in Test II

Subtest	n_i	k_n	Window Size
A	20	7	2
B	10	3	2
C	3	1	2
D	20	7	32
E	10	3	32
F	3	1	32

2.1.5 Results From Test II

The outputs from each subtest can be shown in the corresponding confusion matrices below. The results after classifying the designated testing set can be seen on the left, while the results after trying to reclassify the training sets can be seen on the right.

Each matrix also features calculations regarding the accuracy, as well as the recall rate for each class.

Subtest A

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	10	0
	ω_{bank}	0	10

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	20	0
	ω_{bank}	0	20

Accuracy for testing and training sets:

$AC_{testing} = \frac{10+10}{10+0+0+10} = 1$ $AC_{training} = \frac{20+20}{20+0+0+20} = 1$

Recall rate for testing and training sets:

$RC_{testing_{wallet}} = \frac{10}{10+0} = 1$ $RC_{testing_{bank}} = \frac{10}{0+10} = 1$

$RC_{training_{wallet}} = \frac{20}{20+0} = 1$ $RC_{training_{bank}} = \frac{20}{0+20} = 1$

Subtest B

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	10	0
	ω_{bank}	0	10

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	10	0
	ω_{bank}	0	10

Accuracy for testing and training sets:

$AC_{testing} = \frac{10+10}{10+0+0+10} = 1$ $AC_{training} = \frac{10+10}{10+0+0+10} = 1$

Recall rate for testing and training sets:

$RC_{testing_{wallet}} = \frac{10}{10+0} = 1$ $RC_{testing_{bank}} = \frac{10}{0+10} = 1$

$RC_{training_{wallet}} = \frac{10}{10+0} = 1$ $RC_{training_{bank}} = \frac{10}{0+10} = 1$

Subtest C

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	4	6
	ω_{bank}	7	3

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	3	0
	ω_{bank}	0	3

Accuracy for testing and training sets:

$AC_{testing} = \frac{4+3}{4+6+7+3} = 0.35$ $AC_{training} = \frac{3+3}{3+0+3+0} = 1$

Recall rate for testing and training sets:

$RC_{testing_{wallet}} = \frac{4}{4+6} = 0.4$ $RC_{testing_{bank}} = \frac{7}{7+3} = 0.7$

$RC_{training_{wallet}} = \frac{3}{3+0} = 1$ $RC_{training_{bank}} = \frac{3}{0+3} = 1$

Subtest D

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	8	2
	ω_{bank}	1	9

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	15	5
	ω_{bank}	3	17

Accuracy for testing and training sets:

$AC_{testing} = \frac{8+9}{8+2+1+9} = 0.85$ $AC_{training} = \frac{15+17}{15+5+3+17} = 0.8$

Recall rate for testing and training sets:

$RC_{testing_{wallet}} = \frac{8}{8+2} = 0.8$ $RC_{testing_{bank}} = \frac{9}{9+1} = 0.9$

$RC_{training_{wallet}} = \frac{15}{15+5} = 0.75$ $RC_{training_{bank}} = \frac{17}{17+3} = 0.85$

Subtest E

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	5	5
	ω_{bank}	3	7

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	4	6
	ω_{bank}	2	8

Accuracy for testing and training sets:

$AC_{testing} = \frac{5+7}{5+5+3+7} = 0.6$ $AC_{training} = \frac{4+8}{4+6+2+8} = 0.6$

Recall rate for testing and training sets:

$RC_{testing_{wallet}} = \frac{5}{5+5} = 0.5$ $RC_{testing_{bank}} = \frac{7}{3+7} = 0.7$

$RC_{training_{wallet}} = \frac{4}{4+6} = 0.4$ $RC_{training_{bank}} = \frac{8}{2+8} = 0.8$

Subtest F

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	2	8
	ω_{bank}	2	8

		Actual	
		ω_{wallet}	ω_{bank}
Predicted	ω_{wallet}	3	0
	ω_{bank}	0	3

Accuracy for testing and training sets:

$AC_{testing} = \frac{2+8}{2+8+2+8} = 0.5 \qquad AC_{training} = \frac{3+3}{3+0+3+0} = 1$

Recall rate for testing and training sets:

$RC_{testing_{wallet}} = \frac{2}{2+8} = 0.2 \qquad RC_{testing_{bank}} = \frac{8}{2+8} = 0.8$

$RC_{training_{wallet}} = \frac{3}{3+0} = 1 \qquad RC_{training_{bank}} = \frac{3}{0+3} = 1$

2.1.6 Analysis of the Results in Test II

The results clearly show that the hypothesis was partially wrong; the classifyer can in fact perform well on classes that are difficult to tell apart as well. The requirements for this to be possible are of course good settings.

From Subtests A and B in Test II, it becomes fairly apparent that an appropriate sliding window size is just as important as a good k_n value and a sufficient number of training samples. Subtest A has a perfect accuracy and a perfect recall rate, and the same goes for Subtest B — both when trying the testing sets and when attempting to reclassify the training tables. This is despite the fact that the classes are not easy to separate, illustrating that a fuzzy hashing approach can indeed be used in machine learning with good results. This is at least true if the input files are unformatted text.

Notice that with poor settings (leading to an extreme case of overfitting), the classifyer will be no better than a random guess. Setting the sliding window size to a high number, causes it to ignore changes on a smaller scale in favour of changes on a large scale. In other words, it might not be able to detect what uniquely identifies two very similar classes.

One interesting finding that can be seen from all subtests utilizing a k_n of 1, is that when trying to reclassify the training set, nothing changes. This is due to the fact that with a k_n of 1 (in which case the algorithm is simply called the "Nearest Neighbor" algorithm), only *the* most similar plaintext table is taken into consideration. Since the

plaintext table already has an *exact* copy of itself in the original class, it will thus remain there.

Note that a good sliding window size cannot "save" a classifier with a too small sample size and a too small k_n. The fuzzy hashing scheme is not *instead of* good settings, but *in addition to* good settings. This can be seen from subtest C; while it utilized the same sliding window size as the subtests with perfect results, it nevertheless performed no better than a blind guess.

Finally, the second test also proves that the less separable the classes are, the more the k_n number, sliding window size and the number of samples in the training set matter. This is evidenced by the accuracy and recall rate of Subtest A and Subtest B compared to the rest of the subtests.

3 Using the Merkle Tree-based Approach to Track Changes

This section will utilize smaller test sets than the two previous sections, due to the fact that it uses real data, and because the human tester needs to know where the updates in the table are localized. The intented use of the Merkle Tree based approach is to track changes in already classified tables. Any number of tables can be used; two are used here for every test to keep things simple. The first test will test the order robustnes of a merkle tree, by rearranging a table in two ways. The second test will change values in a different table and compare the updated version with the original version.

The test tables used are enclosed in this project as "tables.7z". Testing has also been done extensively with Avito's own tables. However, as these are confidential, they cannot be shown in this thesis or attached as enclosed sets. Cars were chosen because finding data regarding this is quite convenient and straightforward.

These tests were done with a GUI — not by unit testing. Tests in this section are meant to demonstrate the Merkle Trees ability to track changes, as well as verify it. As such, the tests will not be divided like the previous tests — nor will they focus heavily on arithmetics or feature any form of hypothesis or analysis part.

3.1 Test I: Verifying That the Order of the Elements is Irrelevant

The Merkle Tree-based approach in this project is designed to be "order robust", which means that the order of the elements should not matter. This can be tested with the two files "VolvoC70-2000" and "VolvoC70-2000-sorted". The elements here are in a different order, but the Jaccard indices and the fingerprints of the root nodes will be the same, indicating that nothing has changed.

The fingerprints can be seen in the following table:

	Fingerprint, rows (root)	Fingerprint, columns (root)
Volvo C70	100114697C2E0401	4A6DBDFF8DED9BFE
Volvo C70, order changed	100114697C2E0401	4A6DBDFF8DED9BFE1

Due to the fact that the fingerprints (taken from the root nodes) for both files are similar, no tree traversal was done.

3.2 Test II: Tracking Localized Differences

This test will verify if the Merkle Tree can detect local changes well enough. Here, two tables featuring data for the 1994 SAAB 900 will be used. The first table is based on the 2.0 turbocharged version, while the second table is based on the 2.5 V6 version. Both CSV files are in the enclosed 7z file, under the names "Saab900-20-1994" and "Saab900-25-1994", respectively.

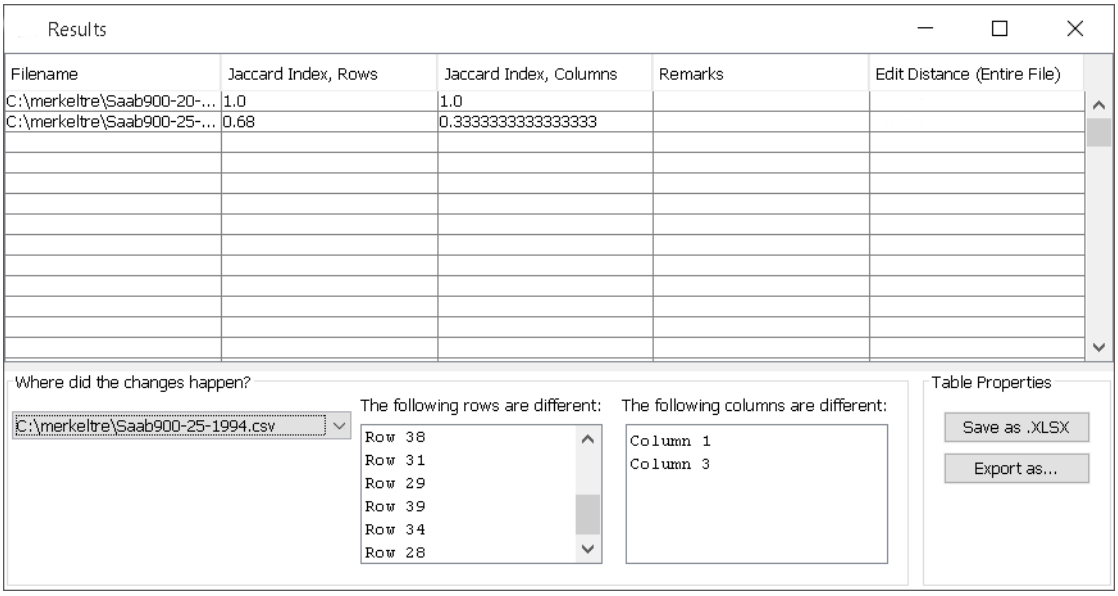
The following fingerprints are given to each table, based on the root node of the corresponding Merkle Trees:

	Fingerprint, rows (root)	Fingerprint, columns (root)
SAAB 900 2.0T	BCFDD71F7CF5DF7C	15676DFEFDFDBEF5
SAAB 900 2.5 V6	F7FDE267BFF1BF9B	2C040B6888010191

Due to the fact that two columns feature headers and two columns hold the numerical values (a change in a single entry will change the fingerprint for an entire row and column), two columns are similar in this case, giving a Jaccard Coefficient of 0.33. The Jaccard Coefficient for rows is 0.68.

The tree traversal algorithm correctly catches 12 out of 13 changes to the rows, the exception being *row 15*, where the weight of the cars are described. This is probably due to a false positive in the Bloom Filter.

Figure 30: Output when comparing two different SAAB 900 models



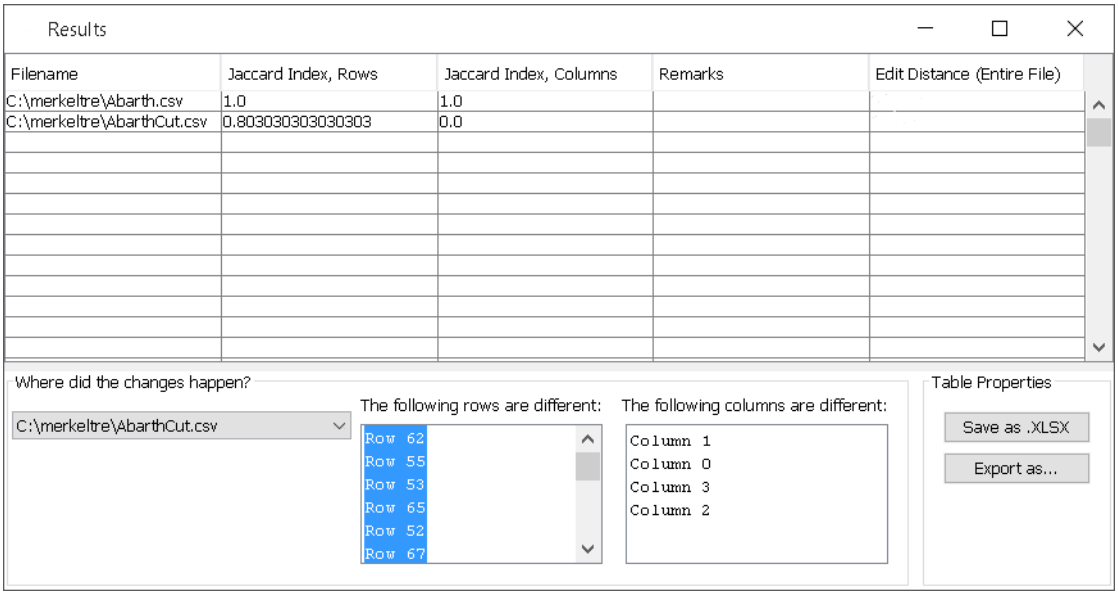
3.3 Test III: Different Length Tables

The Merkle Tree-based approach needs to be able to handle tables of different lengths, which is the main reason for employing a Bloom Filter for every tree.

This test was done with the two files "Abarth" and "AbarthCut". Both deal with the same car (the Autobianchi A112 Abarth), but the former is 20 rows longer than the latter. A total of 17 rows were successfully identified, out of 20.

Not suprisingly, all columns changed, due to the avalanche effect. The Jaccard Coefficient for rows and columns thus become 0.803 and 0, respectively.

Figure 31: Comparing two Different-Length Tables



4 Chapter Summary

The first test in this chapter was about performance. Here, NFHash performed faster than the algorithms that inspired it. When hashing multiple files, it was more than four times faster than Spamsum. In the next test section, it's capabilities as a helping hand for k-NN was demonstrated. On very separable classes, the settings hardly mattered. For classes that were difficult to tell apart, however, correct settings — in terms of sliding windows size, k_n value and number of training samples — proved important. In cases with high overfitting (as a result of the wrong settings), k-NN is no better than a random guess. On the other hand, if the settings were correct, recall rate and accuracy was spot on.

The final tests revolved around the Merkle Trees — and how these efficiently can discover updates or inconsistencies. Here, tables representing cars were used — and it was shown that the algorithm works well on inputs of different lengths or if the nodes were reorganized. Moreover, detecting changes was shown to be fairly straightforward.

The next chapter will conclude this report. It will be about discussion, and will feature future works, various principles and so on. No new technical facts directly related to the implementation or design are presented.

> Be able to defend your arguments in a rational way. Otherwise, all you have is an opinion.
>
> Marilyn vos Savant

Chapter VI
Discussion

This final chapter will discuss the research and outcome of this project — as well as summarize everything that was done. No new facts directly related to the project, nor any background material will be presented here. Rather, this chapter is ment to wrap things up. However, this chapter will include a section on further works, explaining what can be done to improve the implementations here — as well as what can be done to expand them.

1 Originality

The most recently developed data structure in this thesis, is the Merkle Tree, which was designed back in 1979. Fuzzy hashing using rolling hashes have also been available since 2002, while fuzzy hashing algorithms in general have been available since 2001 — and probably even earlier. The target with this thesis has therefore been to make something new that builds on already existing concepts. In the same manner as Spamsum was designed by studying Nilsimsa, NFHash was developed by studying both Spamsum and Nilsimsa. Since it was designed to match files based on content rather than spam recognition, it has fewer loose parts and is optimized for performance. Nevertheless, it achieves good accuracy because of CRC64. The main goal of the algorithm has been to combine Spamsum's accuracy and reliability with the performance of Nilsimsa.

There are quite a few theses out there dealing with fuzzy hashing. The most famous one is probably "mvHash — a new approach for fuzzy hashing", by Knut Petter Åstebøl (Gjøvik University College, 2012). Here, an unusual, but very efficient approach is taken when a file is hashed by majority votes and an RLE compression scheme — rather than by using a rolling hash. The advantage of this approach is that performance is very efficient (it is theoretically impossible to make a rolling hash that scales more quickly than $O(n)$), even though the algorithm is very aligment robust. MVHash and MVHash-B were desgined for forensic uses, but might offer a high collision risks due to producing small digests with base-10 or base-16 numbers. Moreover, it might become problematic that comparison time is very long. More about this can be found in Åstebøl's thesis [54].

Example of how RLE combined with majority vote works in MVHash:

- Output of majority vote: 0.0.FF.0.0.0.0.0.FF.FF.0.FF.FF.FF.FF
- Result after RLE compression: 1|1|4|2|1|4

If MVHash-B is as fast as described in Åstebøl's thesis, it might in fact be a perfect tool for hashing the nodes in a Merkle Tree.

No thesis on Merkle Trees to detect and track changes in a file were found. As previously mentioned, the underlying principle is still the same as bitcoins, P2P applications, and NoSQL systems. There are — hoewever — many theses dealing with cryptographic uses of Merkle Trees. Moreover, the Merkle Tree itself, and several uses for it (most famously as an alternative to DSA) was part of Ralph Merkle's Ph. D. dissertation. [55]

2 Why Study Open Standards and Open Source Code?

A lot of time can be saved by not wasting time on reinventing the wheel. After all, few (if any) scientific papers or discoveries are created out of thin air. Neither Copernicus nor Galilei discovered the heliocentric model; they simply continued where others (most notably Aristarchus of Samos) left of. Charles Darwin did not discover evolution, he simply improved upon the research done by his grandfather, Erasmus Darwin. By studying freely available content, one does not waste time on developing something that already has a simple solution.

This project has very little to do with security, except in the fields of collisions. Nevertheless, security is often the main reason for the misconception that something has to be secret or hidden to be useful. There are many commonly used fallacies and misconceptions concerning secret vs. open standards. The practice of using secrecy as a mean to achieve security is referred to as **security through obscurity**. While keeping part of the source code obfuscated and "hidden" can be a good thing to prevent bypassing various security meassures, the algorithms themselves (hash functions, encryption algorithms, and so on), should not rely on secrecy to achieve good security, good performance or good accuracy. The **security by design** principle means that an algorithm should be designed to be secure from the ground, and going by **Linus' law**, a security vulnerability, a performance bottleneck or other undesireable features will be discovered more quickly if the algorithms are open. [56] Moreover, if an algorithm is freely available and widely used, weaknesses and strengths are also more widely known — as well as alternative uses for the algorithm.

Decompilers for C# and Java are easily available, and decompilers for C and C++ are also improving. On a lower level it is also possible to disassemble the code, for more experienced hackers. Secrecy for hash functions will at best stall the time an atacker needs if it is a secure hashing scheme, although proprietary algorithms might have their place in intranets, as demonstrated by Cisco. Keep in mind that secrecy and security are not mutually exclusive, but if a hashing algorithm or something similar is secure

from the ground, it does not matter if an attacker knows how it works or not, as long as he/she does not have the necessary supplements (eg. keys, corresponding plain and cipher texts and so on). The bottomline is that an algorithm should not *depend on* keeping any functionality hidden to be efficient, easy to expand or secure — although these traits are certainly not mutually exclusive with secrecy either.

This project had nothing to do with either cryptocurrency or noSQL systems. Nevertheless, the open bitcoin standard and the Cassandra source code proved useful to study.

Lastly, even if an open standard is brand new, one can still check the credentials of the author. If a datastructure or an algorithm has not gained widespread usage yet, but is merely a theory from a respected scientist, a concept from a high ranking member of the open-source community or for that matter a beta version larger company like Google or Microsoft, it might deserve a chance after all.

2.1 Old Mathematical Theorems Might be a Priceless Source

Clearly, a mechanic does not want to make the tools himself if there are already tools out there that will get the job done; in the same manner, a baker does not wish to make the oven himself and a hairdresser does not want to spend time designing the scissors herself. Likewise, there is very little point trying to reinvent the wheel when trying to design algorithms or designing new approaches in software-development. Mathematical theories can be considered analogous to a toolbox. If something can be done efficiently mathematically — without using division or remainders — then it can also be done efficiently in a C-like programming language.

It does not matter if the mathematical theories are well-known or not. Asymmetric cryptography did not get the recognition it deserved until the mid 1970's, despite being proven possible by pen and paper in the mid 1850s.. Mersenne Primes are widely-known for anyone who look for them, but have not gained widespread use in fuzzy hashing algorithms. Lastly, the theories of George Boole and Evariste Galois did not catch on until the rise of digital technology, long after both had passed on.

2.2 Datastructures and Algorithms Almost Always Have Many Uses

A lot of the modern technologies found in 2016 either began their lives inside a very limited scope or inside a very different scope. Several of the improvements of the diesel engine (turbochargers, the common-rail fuel injection system, and so on) during the last three decades began as technologies designed to make submarines in the early 20th century more powerful without increasing the fuel consumption. And while there are billions of smart phones out there, these all built upon technology that was initially

developed for the military (ENIAC — the first Turing-complete digital computer — was originally designed to calculate artillery firing tables).

Algorithms are no different. An example is Dijkstra's algorithm for finding paths in a graph; this was later developed into the *Fast Marching Algorithm*, which today is an invaluable tool in many CAD programs and picture manipulation tools for detecting boundaries. Merkle Trees were originally designed to detect inconsistencies and have until now for the most part been used in peer-to-peer protocols, cryptocurrency and no SQL database systems, such as Cassandra. With a sorting mechanism on the terminal nodes and a Bloom filter to be used if potentially matching nodes are not in the same place, a Merkle Tree can also be used to track changes in plaintext with clearly defined boundaries and delimiters.

2.2.1 Security Might Present Exceptions

As demonstrated, hash functions are both versatile and very useful. There is, however, no free lunch. The output will almost always be incomprehensible to human eyes — which is a good thing in terms of security — but as previously mentioned, a hash-function is one-way. Thus, a hash function cannot be used to *encrypt* large documents, E-mails, and so on, just used to identify them. Moreover, since the output is irreversible in a reasonable amount of time, it cannot be used for compressed archives.

Purpose-designed fingerprinting algorithms will not offer the same security as cryptographic hash functions (except in the form of collision resistance). Typically, they are much more vulnerable to preimage attacks — especially for algorithms that are being used for fuzzy fingerprinting. Generally, fingerprinting algorithms are kept simple and no-frills, for the purpose of performance. Because of these properties, algorithms typically used to identify files, such as **CRC**, **Adler's Checksum**, **Rabin's Fingerprinting Algorithm**, **Fowler-Noll-Vo** and so on should not be used to hash keys, passwords, and sensitive information in general. They can, however (as previously mentioned), be used for sensitive information internally within a secure environment for identification purposes.

An example of what might happen if a fuzzy fingerprinting scheme was used to hash sensitive data:

1. Bob authenticates content from Alice based on the edit distance of file hashes received by Alice and entries in his whitelist
2. If the edit distance is low, he will accept the file. Otherwise, he will reject it
3. Should Eve get hold of the whitelist Bob compares the files against, she could make a malicious file with a similar digest (as in low edit distance), masquerade as Alice and trick Bob into installing the file

Clearly, a whitelist (which contains digests of good or approved files) using nothing more

than a simple fuzzy hash cannot tell the difference between a game file that has been modified by a gamer for fun or a game file with a trojan trapped inside it. Moreover, it cannot tell the difference between an excel file that has been updated or a similar XLSX file that contains a macro virus. In theory, a blacklist (which contains hashes of malicious files) can make use of preexisting fuzzy hashing algorithms such as Spamsum. A simple example can be a keygenerator (an infamous source of malware) with a trojan or a setup file that installs adware or spyware; if this keygenerator or this setup file has a digest resembling that of a previous keygenerator or setup file with malware, a virus program can then assume that it is, in fact, malware. Fuzzy hashing might be a viable "medicine" against morphable malware. This can be taken further by using a k-nearest neighbor or Parzen window approach.

NFHash is in theory secure against anti-blacklist attacks at first glimpse, but it has not been tested for this, and therefore, it should be treated as if it is not safe for this purpose (i.e. not used to filter out spam or malware). The simplicity of the algorithm means that in practice, it might not be secure against these attacks. Any adversary who knows how the algorithm works can gather enough spam (there is no shortage of this), train a k-NN classifier and write a spam email that will trick the algorithm. Generally, spam detection tools have to be lenient, as misclassifying important mails as spam is far worse than letting a few spam mails slip through the cracks.

While some cryptographic algorithms such as SHA-2 will indeed offer strong collision resitance (and of course excellent one-way properties), they are nevertheless too slow to be able to fingerprint many files, rows, and columns in a short time, which is crucial in a software tool similar to **Avito Loops**. Some older cryptographic hash functions now considered "broken", such as *MD5* and *SHA-1* have nevertheless found new life in terms of fingerprinting, due to being significantly faster than more modern cryptographic algorithms.

3 Can an Algorithm Really be One-Way?

When discussing one-way functions, it does not refer to a function that is *inherently one-way*. Indeed, the question of whether one-way functions really exist or not is an example of an unsolved question. If one could prove that there was an efficient way to reverse the output of a hash function, one would also have proved that there is a non-polynomial problem that does have a polynomial (or "easy") solution. This is also an example of one of the greater questions in computer science. Generally, a hash function can be considered one-way "enough" when it is more computationally demanding to reverse the output directly than it is to attack it by brute-force. Even in the case of the less secure fingerprinting algorithms, reversing the output requires much resources.

Checksum and fingerprinting algorithms such as CRC and Rabin's Fingerprinting Scheme can be more accurately described as trapdoor functions rather than one-way functions.

The output from the paper shredder pictured in the beginning of this chapter can be pasted together after hours of tedious work, and in theory, with sufficient computer power, the output from Rabin's Fingerprinting Scheme and CRC (as well as the output several cryptographic algorithms if no salt is used) can indeed be reversed indirectly. The most famous way to do this is probably with the help of rainbow tables.

4 No Fingerprinting Scheme is Inherently Superior

This report presents two designs. The first one is NFHash, while the second one is based on Merkle Trees. These are not superior or inferior to each other, but have different uses, NFHash's intended usage is to classify plaintext tables, while the Merkle Tree-based approach is intended to track changes in classified files. In other words, they complement each other rather than substituting each other.

4.1 Limitations of the Two Approaches Presented in This Report

The limitations are presented earlier on in the report, but the most important aspects will still be summarized here.

- The Merkle Tree is bottlenecked by the fact that a digest in a traditional hashing scheme either changes completely or it does not change at all. Therefore, a row insert and a column insert at the same time will result in the algorithm not being able to track the changes.
- Before NFHash can be put to use, a person using it needs some practice in how to set the different parameters.

5 Further Works

The elephant in the room in terms of further works, is to find a way to bypass the "either this or the other" approach of the Merkle Tree; that is that it will perceive all rows as different if there is a column insertion — or all columns as different if there is a row insertion. MVHash-B — as presented in Åstebøl's thesis — might be a possible solution. The row or column inserted will still be transparent (that is, it will not be possible for a user to pinpoint where it was inserted), but if Åstebøl's thesis is correct, it might nevertheless allow a user to tracck all other changes in a table. The drawback here is that MVHash-B is not open-source nor is it used in any commercially available applications (even if it was, it would have been copyrighted) — and thus, little is known about the algorithm itself, how to implement it, and how well it performs on different tasks. Another thing that could be added two the Merkle Tree, is two or three fingerprints

per node instead of just one. This would also help the Bloom Filter, but would probably lead to a higher memory usage.

None of the two approaches presented in this paper looks at anything other than plaintext or file content with little repetition (eg. PNG files). NFHash has already proven to be efficient for files that are similar in size to a typical png file — by looking at the bytes only. However, a hashing scheme alone does not care about what the picture represents.

Figure 32: Two similar png images that have two different NFHash digests because the latter is an inverted version of the former in terms of colors (note: some bytes are similar due to the pictures having similar size and similar file format)

8hf6rLeTCjcvL11118if5zfJfDfj3LnrlnX
hTxMM18qJDjZIMWUZ/B3V5ZqEMk7+

8hf6bTOmNBykqopZ+8Gg1NCnG111
18if5zfJfDgViDr70oaFoAvVB27aojI+3nhb
B

Recall the *Fast Marching Algorithm* earlier in this chapter. If figure 32 instead featured contoures represented by 1 (black) and 0 (white) rather than the picture itself, NFHash could potentially classify it based on what it represents — if the classifyer was trained with many pictures. The *Fast Marching Algorithm* would play a vital role here, since it traces edges, rather than decide whether to paint a pixel black or white based on the initial colors. The key here is of course to make sure the 1s and 0s matrix does not get too large.

On a lower level, a random polynomial generator based on Rabin's Irreducibility Test could be implemented, so that the user would not have to manually input the polynomials himself/herself.

Lastly, expanding the k-NN classifyer to counter overfitting better would be one of the top priorities. Fuzzy hashing helps a lot against overfitting since it flattens the

dimensions, but it is still not a perfect cure against it. Steps that can be taken to reduce overfitting are described in pages 45–48. Examples could be mechanisms for resampling, or letting the classifier decide the k-NN number based on heuristics.

Generally, due to the modular layout of the library, these changes can be implemented by adding more classes; there is no need to modify any existing code, except for possible microoptimizations.

6 Learning Experience

Arguably, the most important experience in this project has been non-cryptographic hash functions and their uses — as well as thinking outside of the box by finding new uses for content previously learned during the master's studies and even the bachelor's studies. This also applies to subjects not directly linked to (but still relevant to) computer science, such as discrete mathematics and pattern recognition.

One of the most valuable lessons learned in this project apart from data fingerprinting, is that small scale testing (eg. unit testing) might not reveal the entire truth. Something that looks and performs perfectly inside a small, artificial test case, might show several bugs in real-world applications using actual tables. Optimizations and learning how to do this at a lower level also proved a useful thing to learn.

Another important thing that proved to be more valuable than previously thought, is the literature study — so that it became possible to avoid pitfalls other people had fallen into, and possible to build upon research done by others. An important part of the literature study was also to learn how to apply discrete mathematics to solve problems that were not written on paper, but were actual, real-world problems.

Initially, the Merkle Tree led to some difficulties due to a lack of alignment- and insertion robustness. The learning outcome here was to keep an open mind about datastructures and different data structures were useful. If the Merkle Tree-based approach was not proven possible for tracking updates with "pen and paper" before it was implemented in Java code, it would probably be scrapped as soon as problems were encountered.

Lastly, the importance of good explanations, and good documentations were learned. After all, a library nobody knows how to use, nobody knows how works, and everybody finds confusing, is pretty much useless. At the end of the day, this is a project that might be useful to somebody, and not just an exam dealing with artificial problems.

7 Conclusion

Since hash functions and discrete mathematics have been of interest for a long time, this project proved to be a golden oportunity. Extensive work had already been done

by others in this subject of hash functions — highlighting that they in fact have many uses. Moreover, the work done by mathematicians in the past, essentially ment that the road taken in this project was already paved — even though it was challenging at times.

Chapter 3 focused on the design part — as well as datastructures and algorithms not typically part of the curriculum in most degrees. Two approaches were designed; on of them was a fuzzy hashing scheme designed from scratch, but with heavy influence from Spamsum and Nilsimsa. The other one was Merkle Tree-based, and was inspired by a lot of things not related to fingerprinting files, such as Bitcoins. In this chapter, figures and formulas proved more important than words. This might be especially true for the 2. design, utilizing Merkle Trees. For the 4. chapter, explaining the previous chapter with Java code was the most important part. Furthermore, Boolean algebra was revisited, so that optimizations on all levels could be explained.

The "Testing and Analysis" chapter proved that NFHash is almost four times faster than Spamsum on average — and 30% faster than Nilsimsa. What gives these three algorithms their strength is also what ultimately holds them back performance-wise. A sliding window requires a lot of resources, after all. The Merkle Tree would also prove to be an excellent tool for finding out where the changes happened, as long as the files compared had an equal number of rows or an equal number of columns. For this purpose tables with car data was used. The Merkle Trees correctly produced the same fingerprints for two table where the order of the elements was completely different, but the elements themselves were identical. The algorithm also tracked most changes, and was also able to do comparisons of two tables with an uneven length.

This chapter has focuses on discussing the outcome of the project — and giving a little more depth to the background chapter. Overall, the project has been very interesting, and has dealt with both computer science and discrete mathematics to a large degree. The information in this report has largely been given by figures, tables and equations, with the written words for the most part backing these up. Hopefully, this has made the report interesting to read.

References

[1] Stanford lecture. URL:http://web.stanford.edu/class/cs345a/slides/04-highdim.pdf
Retrieved: May 27. 2016.

[2] Dustin Hurlbut. Fuzzy hashing for digital forensic investigators. 2009.

[3] Agner Fog. Instruction tables. URL: http://www.agner.org/optimize/instruction_tables.pdf
Retrieved: February 26. 2016.

[4] Federal standard 1037c. URL: http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm
Retrieved:March 4. 2016.

[5] W. Sierpinski. *Elementary Theory of Numbers: Second English Edition*, volume 2. PWN-Polish Scientific Publishers, 1988. Page: 360.

[6] Eric Temple Bell. *The Queen of Mathematics*, volume 3. Springer, 1964. Page: 227.

[7] Some irreducible polynomials. URL: http://www.math.umn.edu/ garrett/m/algebra/notes/07.pdf
Retrieved: February 15. 2016.

[8] Michael O Rabin. Probabilistic algorithms in finite fields. *SIAM Journal on Computing*, 9(2):273–280, 1980. Pages: 273–280.

[9] William Stallings. *Cryptography and Network Security: Principles and Practice*, volume 5. Pearson Prentice Hall, 2011. Page: 276.

[10] William Stallings. *Cryptography and Network Security: Principles and Practice*, volume 5. Pearson Prentice Hall, 2011. Page: 328.

[11] Michael O. Rabin. Fingerprinting by random polynomials. 1981.

[12] Thomas H. Cormen et. al. *Introduction to Algorithms 3rd Edition*, volume 3. The MIT Press, 2009. Page: 257.

[13] William Stallings. *Cryptography and Network Security: Principles and Practice*, volume 5. Pearson Prentice Hall, 2011. Page: 336.

[14] Fred B. Schneider David Gries. *A Logical Approach to Discrete Math*, volume 1. Springer, 1993. Page: 355.

[15] Cliff Wang Mohammad Tehranipoor. *Introduction to Hardware Security and Trust*, volume 1. Springer, 2012. Page: 28.

[16] Andrei Z. Broder. Some applications of rabin's fingerprinting method. 1993.

[17] URL: http://betterexplained.com/articles/understanding-the-birthday-paradox/
Retrieved: 02.02.2016.

[18] Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009. Page: 185.

[19] Andrew Tridgell. Spamsum source code (written in c), 2002. License: GNU General Public Licese v. 2.

[20] Title: FNV hash history
Retrieved: February 28. 2016
Address: http://www.isthe.com/chongo/tech/comp/fnv/index.htmlFNV-1.

[21] LP Deutsch. Rfc 1952: Gzip file format specification version 4.3. *Internet Engineering Task Force*, 1996.

[22] David P Schultz and Christopher D Ebeling. Method and apparatus for implementing a cyclic redundancy check circuit, July 17 2012. US Patent 8,225,187.

[23] Marc Stevens, Arjen Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and colliding x. 509 certificates for different identities. In *Advances in Cryptology-EUROCRYPT 2007*, pages 1–22. Springer, 2007. Pages: 1–22.

[24] Sheelagh Lloyd. Counting functions satisfying a higher order strict avalanche criterion. In *Advances in Cryptology—EUROCRYPT'89*, pages 63–74. Springer, 1989. Pages: 63–74.

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

[26] Andrew Tridgell. Spamsum readme, 2002. URL: https://www.samba.org/ftp/unpacked/junkcode/spamsum/README.

[27] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. An open digest-based technique for spam detection. *ISCA PDCS*, 2004:559–564, 2004.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system bitcoin: A peer-to-peer electronic cash system.

[29] Hinrich Schütze Christopher D. Manning, Prabhakar Raghavan. *Introduction to Information Retrieval*, volume 1. Cambridge University Press, 2008. Page: 56.

[30] Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971. Pages: 34–35.

[31] Jan Van Leeuwen. *Handbook of theoretical computer science (vol. A): algorithms and complexity*. Mit Press, 1991. Page: 294.

[32] Gregory V Bard. Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In *Proceedings of the fifth Australasian symposium on ACSW frontiers-Volume 68*, pages 117–124. Australian Computer Society, Inc., 2007. Pages: 117–124.

[33] Peter Eades and Petra Mutzel. Algorithms and theory of computation handbook. 1998. Section 30.5.1.

[34] Pedro Franco. *Understanding Bitcoin: Cryptography, engineering and economics*. John Wiley & Sons, 2014. Page: 117.

[35] Kent D Lee and Steve Hubbard. *Data Structures and Algorithms with Python*. Springer, 2015. Page: 206.

[36] Mahmoud Parsian. *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark*. " O'Reilly Media, Inc.", 2015. Page: 694.

[37] Bloom filters - the math, 1998. URL: http://pages.cs.wisc.edu/ cao/papers/summary-cache/node8.html.

[38] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[39] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012. Page:174.

[40] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012. Page: 183.

[41] Lior Rokach. *Pattern classification using ensemble methods*, volume 75. World Scientific, 2009. Page: 160.

[42] Ashish Gupta. *Learning Apache Mahout Classification*. Packt Publishing Ltd, 2015. Page: 16.

[43] Henry Garner. *Clojure for Data Science*. Packt Publishing Ltd, 2015. Page: 226.

[44] David J Marchette. *Random graphs for statistical pattern recognition*, volume 565. John Wiley & Sons, 2005. Page: 16.

[45] J Ross Quinlan. Bagging, boosting and c4.5. In *AAAI/IAAI, Vol. 1*, pages 725–730, 1996. Pages: 725–730.

[46] Reference: Nonprintable and printable ascii characters. URL: http://www.juniper.net/documentation/en_US/idp5.1/topics/reference/general/intrusion-detection-prevention-custom-attack-object-extended-ascii.html
Retrieved: May 12. 2016.

[47] Francois Yergeau. Utf-8, a transformation format of iso 10646. 2003.

[48] Simon Josefsson. The base16, base32, and base64 data encodings. 2006.

[49] base91 encoding. URL=http://base91.sourceforge.net/.

[50] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Den-*

mark (Lyngby): Technical University of Denmark, page 220, 2016. URL: http://www.agner.org/optimize/instruction_tables.pdf.

[51] Oracle. java.math.biginteger class.

[52] Henry S Warren. *Hacker's delight*. Pearson Education, 2013. Page: 328.

[53] GP Nikishkov, Yu G Nikishkov, and VV Savchenko. Comparison of c and java performance in finite element computations. *Computers & structures*, 81(24):2401–2408, 2003. Pages: 2401–2408.

[54] Knut Petter Åstebøl. mvhash: a new approach for fuzzy hashing. 2012.

[55] Ralph C Merkle. *Secrecy, authentication, and public key systems Ph. D*. PhD thesis, dissertation Department of Electrical Engineering. PhD thesis, Stanford University, 1979. Page: 41B.

[56] Eric S. Raymond. *The Cathedral and the Bazaar*, volume 1. O'Reilly Media, 1999. Page: 30.

Appendix

A User Manual

To better understand how the code works, a simple user manual is included. This will not explain every part of the code in detail, but it will show the simplest way to get the implementations up and running.

A.1 Creating and Using a Merkle Tree

The easiest way to create a Merkle Tree in this project, is to pass an ArrayList holding fingerprints of the terminal nodes, an irreducible polynomial and an ArrayList holding the IDs of the nodes into the constructor, like this:

```
merkle mkl= new merkle(TerminalFingerPrints, 0xB, TagList);
```

It will then create an instance of Rabin's Fingerprinting Scheme and a lookup table on it's own (transparent to the user if this approach is taken). Note that in a typical case, an irreducible polynomial larger than B_{16} is preferred.

Two Merkle Trees can be compared the followig way:

```
treeComparator tc = new treeComparator();
Stack<Integer> differentNodes =  tc.bloomCompare(mkl_a, mkl_b); // Comparing
    two trees names mkl_a and mkl_b
int numDiff = tc.neq; // Number of differing nodes
double jacc = tc.jaccard(mkl_a.depth, mkl_b.depth, numDiff); // Jaccard
    Coefficient
```

A.2 How to Use NFHash

NFHash requires some practice from the user to actually do a good job. An NFHash object can be created the following way:

```

int windowSize = 4;
int boundaryNum = 128;
NFHash nfh = NFHash(windowSize, boundaryNum);
String fingerprint = nfh.hashByte(inputByteArray);
```

A.3 Classifying Content With k-NN

```
kNN classifyer = new kNN();

for (int i=0; i<inputFiles.length(); i++){ // Train the classifyer with a
    class
classifyer.add(inputFiles[i],i); // Array of input fingerprints
}

for (int j=0; j<inputFiles2.length(); j++){ // Train the classifyer with
    another class
classifyer.add(inputFiles2[j],j); // Array of input fingerprints
}

int kNum = 9;
int classnum=classifyer.classify(testFingerprint, kNum); // Pass the testing
    sample you want to classify into the classifyer
```

In the above case, `classnum` becomes the class as an integer. If there is a datastructure holding the class names as well, this number can be used to index it to get the name.

B Complete Code

Complete code can be found in the enclosed 7z file "sourcecode.7z". It is also included as text in this appendix. Note: You need to specify a package name before you use these classes.

B.1 Bloom Filter

bloomFilter.java

```
import java.io.Serializable;
import java.util.BitSet;

public class bloomFilter<E> implements Serializable {
    private final BitSet bitset;
    private final int bitSetSize;
    private final int bitsPerElement;
    private final int expN; // expected (maximum) number of elements to be
        added
    private int numberOfAddedElements; // number of elements actually added
        to the Bloom filter
    int mask=0;

    /*
    bitSize = amount of bits per element
    expN = expected number of elements in bloom filter
     */

    public bloomFilter(final int bitSize, final int expN) {
        this.expN = expN;
        this.bitsPerElement = bitSize;
        this.bitSetSize = (bitSize * expN)*2;
        this.bitset = new BitSet(bitSetSize);
    }

    // Reset filter

    public void clear() {
        bitset.clear();
        numberOfAddedElements = 0;
    }

    // Add a digest

    public void add(long num) {

        int left = (int)(num >> 32); // Split the long into two
        int right = (int)num;

        System.out.println("adding");
```

```
            bitset.set((left & bitSetSize-1), true);
            bitset.set((right & bitSetSize-1), true);
            numberOfAddedElements ++;
        }

        // Check if the Bloom Filter contains a digest:

        public Boolean contains(long num) {

            int left = (int)(num >> 32); // Split the long into two
            int right = (int)num;

            if (!bitset.get(left & bitSetSize-1) && !bitset.get(right &
                bitSetSize-1)) {
                System.out.println("not retrieving");
                    return false;
                }
            System.out.println("retrieving");
            return true;
        }
}
```

B.2 Cyclic Redundancy Check

CRC64.java

```
import java.util.zip.Checksum;

public final class CRC64 implements Checksum {

    // Based on ISO polynomial:

    private static final long[] lookup = new long[]{
            0x0000000000000000L, 0x01b0000000000000L, 0x0360000000000000L,
            0x02d0000000000000L, 0x06c0000000000000L, 0x0770000000000000L,
            0x05a0000000000000L, 0x0410000000000000L, 0x0d80000000000000L,
            0x0c30000000000000L, 0x0ee0000000000000L, 0x0f50000000000000L,
            0x0b40000000000000L, 0x0af0000000000000L, 0x0820000000000000L,
            0x0990000000000000L, 0x1b00000000000000L, 0x1ab0000000000000L,
            0x1860000000000000L, 0x19d0000000000000L, 0x1dc0000000000000L,
            0x1c70000000000000L, 0x1ea0000000000000L, 0x1f10000000000000L,
            0x1680000000000000L, 0x1730000000000000L, 0x15e0000000000000L,
            0x1450000000000000L, 0x1040000000000000L, 0x11f0000000000000L,
            0x1320000000000000L, 0x1290000000000000L, 0x3600000000000000L,
            0x37b0000000000000L, 0x3560000000000000L, 0x34d0000000000000L,
            0x30c0000000000000L, 0x3170000000000000L, 0x33a0000000000000L,
            0x3210000000000000L, 0x3b80000000000000L, 0x3a30000000000000L,
            0x38e0000000000000L, 0x3950000000000000L, 0x3d40000000000000L,
            0x3cf0000000000000L, 0x3e20000000000000L, 0x3f90000000000000L,
            0x2d00000000000000L, 0x2cb0000000000000L, 0x2e60000000000000L,
            0x2fd0000000000000L, 0x2bc0000000000000L, 0x2a70000000000000L,
            0x28a0000000000000L, 0x2910000000000000L, 0x2080000000000000L,
            0x2130000000000000L, 0x23e0000000000000L, 0x2250000000000000L,
```

```
28              0x2640000000000000L, 0x27f0000000000000L, 0x2520000000000000L,
                0x2490000000000000L, 0x6c00000000000000L, 0x6db0000000000000L,
30              0x6f60000000000000L, 0x6ed0000000000000L, 0x6ac0000000000000L,
                0x6b70000000000000L, 0x69a0000000000000L, 0x6810000000000000L,
32              0x6180000000000000L, 0x6030000000000000L, 0x62e0000000000000L,
                0x6350000000000000L, 0x6740000000000000L, 0x66f0000000000000L,
34              0x6420000000000000L, 0x6590000000000000L, 0x7700000000000000L,
                0x76b0000000000000L, 0x7460000000000000L, 0x75d0000000000000L,
36              0x71c0000000000000L, 0x7070000000000000L, 0x72a0000000000000L,
                0x7310000000000000L, 0x7a80000000000000L, 0x7b30000000000000L,
38              0x79e0000000000000L, 0x7850000000000000L, 0x7c40000000000000L,
                0x7df0000000000000L, 0x7f20000000000000L, 0x7e90000000000000L,
40              0x5a00000000000000L, 0x5bb0000000000000L, 0x5960000000000000L,
                0x58d0000000000000L, 0x5cc0000000000000L, 0x5d70000000000000L,
42              0x5fa0000000000000L, 0x5e10000000000000L, 0x5780000000000000L,
                0x5630000000000000L, 0x54e0000000000000L, 0x5550000000000000L,
44              0x5140000000000000L, 0x50f0000000000000L, 0x5220000000000000L,
                0x5390000000000000L, 0x4100000000000000L, 0x40b0000000000000L,
46              0x4260000000000000L, 0x43d0000000000000L, 0x47c0000000000000L,
                0x4670000000000000L, 0x44a0000000000000L, 0x4510000000000000L,
48              0x4c80000000000000L, 0x4d30000000000000L, 0x4fe0000000000000L,
                0x4e50000000000000L, 0x4a40000000000000L, 0x4bf0000000000000L,
50              0x4920000000000000L, 0x4890000000000000L, 0xd800000000000000L,
                0xd9b0000000000000L, 0xdb60000000000000L, 0xdad0000000000000L,
52              0xdec0000000000000L, 0xdf70000000000000L, 0xdda0000000000000L,
                0xdc100000000000000L, 0xd5800000000000000L, 0xd4300000000000000L,
54              0xd6e0000000000000L, 0xd7500000000000000L, 0xd3400000000000000L,
                0xd2f0000000000000L, 0xd0200000000000000L, 0xd1900000000000000L,
56              0xc3000000000000000L, 0xc2b0000000000000L, 0xc0600000000000000L,
                0xc1d0000000000000L, 0xc5c0000000000000L, 0xc4700000000000000L,
58              0xc6a0000000000000L, 0xc7100000000000000L, 0xce800000000000000L,
                0xcf300000000000000L, 0xcde0000000000000L, 0xcc500000000000000L,
60              0xc8400000000000000L, 0xc9f0000000000000L, 0xcb200000000000000L,
                0xca900000000000000L, 0xee000000000000000L, 0xefb0000000000000L,
62              0xed600000000000000L, 0xecd0000000000000L, 0xe8c0000000000000L,
                0xe9700000000000000L, 0xeba0000000000000L, 0xea100000000000000L,
64              0xe3800000000000000L, 0xe2300000000000000L, 0xe0e0000000000000L,
                0xe1500000000000000L, 0xe5400000000000000L, 0xe4f0000000000000L,
66              0xe6200000000000000L, 0xe7900000000000000L, 0xf5000000000000000L,
                0xf4b0000000000000L, 0xf6600000000000000L, 0xf7d0000000000000L,
68              0xf3c0000000000000L, 0xf2700000000000000L, 0xf0a0000000000000L,
                0xf1100000000000000L, 0xf8800000000000000L, 0xf9300000000000000L,
70              0xfbe0000000000000L, 0xfa500000000000000L, 0xfe400000000000000L,
                0xfff0000000000000L, 0xfd200000000000000L, 0xfc900000000000000L,
72              0xb4000000000000000L, 0xb5b0000000000000L, 0xb7600000000000000L,
                0xb6d0000000000000L, 0xb2c0000000000000L, 0xb3700000000000000L,
74              0xb1a0000000000000L, 0xb0100000000000000L, 0xb9800000000000000L,
                0xb8300000000000000L, 0xbae0000000000000L, 0xbb500000000000000L,
76              0xbf400000000000000L, 0xbef0000000000000L, 0xbc200000000000000L,
                0xbd900000000000000L, 0xaf000000000000000L, 0xaeb0000000000000L,
78              0xac600000000000000L, 0xadd0000000000000L, 0xa9c0000000000000L,
                0xa8700000000000000L, 0xaaa0000000000000L, 0xab100000000000000L,
80              0xa2800000000000000L, 0xa3300000000000000L, 0xa1e0000000000000L,
```

```
            0xa050000000000000L, 0xa440000000000000L, 0xa5f0000000000000L,
            0xa720000000000000L, 0xa690000000000000L, 0x8200000000000000L,
            0x83b0000000000000L, 0x8160000000000000L, 0x80d0000000000000L,
            0x84c0000000000000L, 0x8570000000000000L, 0x87a0000000000000L,
            0x8610000000000000L, 0x8f80000000000000L, 0x8e30000000000000L,
            0x8ce0000000000000L, 0x8d50000000000000L, 0x8940000000000000L,
            0x88f0000000000000L, 0x8a20000000000000L, 0x8b90000000000000L,
            0x9900000000000000L, 0x98b0000000000000L, 0x9a60000000000000L,
            0x9bd0000000000000L, 0x9fc0000000000000L, 0x9e70000000000000L,
            0x9ca0000000000000L, 0x9d10000000000000L, 0x9480000000000000L,
            0x9530000000000000L, 0x97e0000000000000L, 0x9650000000000000L,
            0x9240000000000000L, 0x93f0000000000000L, 0x9120000000000000L,
            0x9090000000000000L
    };


    @Override
    public void update(final byte[] bytes, int offset, int length) {
        final int end = length - offset;
        long crc = 0L;
        while (offset < end) {
            crc = lookup[(bytes[offset] ^ (int) crc) & 0xFF] ^ (crc >>> 8);
            offset++;
        }
        this.crc = crc;
    }

    long crc = 0L;

    @Override
    public long getValue() {
        return (crc ^ 0xFFFFFFFFFFFFFFFFL); // Invert the output
    }

    // Redundant methods (only present because this class implements an
        interface):

    @Override
    public void reset() {
        crc = 0;
    }
    @Override
    public void update(int i) {
    }
}
```

B.3 Damerau-Levenshtein Distance Calculator

DamerauLevenshtein.java

```
import java.util.Arrays;
```

```
3  public class DamerauLevenshtein {
       public static int DamLev(String first, String second){
5          final int init = 0;
           int[][] dist = new int[first.length()+2][second.length()+2];
7          dist[0][0]=init; // This is the matrix holding the edit distance for
               each letter

9          for(int i = 0; i<=first.length(); i++) {
               dist[i+1][1] = i;
11             dist[i+1][0] = init;
           }
13         for(int j = 0; j<=second.length(); j++) {
               dist[1][j+1] = j;
15             dist[0][j+1] = init;
           }
17
           int[] DA = new int[256]; // 256 printable characters
19         Arrays.fill(DA, 0);

21         for(int i = 1; i<=first.length(); i++) {
               int DB = 0;
23             for(int j = 1; j<=second.length(); j++) {
                   int i1 = DA[second.charAt(j-1)];
25                 int j1 = DB;

27                 int d=0;

29                 if (first.charAt(i-1)==second.charAt(j-1)){
                       d=0;}
31                 else d=1;

33                 if(d==0) DB = j;
                   dist[i+1][j+1] =
35                         Math.min(Math.min(dist[i][j]+d,
                                   dist[i+1][j] + 1),Math.min(
37                                 dist[i][j+1]+1,
                                   dist[i1][j1] + (i-i1-1) + 1 + (j-j1-1))); //
                                       This line separates Damerau-Levenshtein
                                       from
39                 //ordinary Levenshtein
               }
41             DA[first.charAt(i-1)] = i;
           }
43         return dist[first.length()+1][second.length()+1];
       }
45 }
```

B.4 k-Nearest Neighbor

kNN.java

```
1  package knn;
```

```
3  import java.util.ArrayList;
   import java.util.Arrays;
5  import java.util.Collections;

7  /**
    * Created by Ole on 06.06.2016.
9   */
   public class kNN {
11
       static DamerauLevenshtein cmp = new DamerauLevenshtein();
13     static ArrayList<ArrayList<String>> classes = new ArrayList<ArrayList<
           String>>();
       static ArrayList<ArrayList<Integer>> distances = new ArrayList<ArrayList<
           Integer>>();
15     static double prior;
       int numClass;
17
       // Create Class Based on Input Files:
19
       public void createClass(String[] digests){
21         ArrayList<String> digestArray = new ArrayList<String>(Arrays.asList(
               digests));
           classes.add(digestArray);
23     }

25     // Remove Class:

27     public void remove(int classNum){
           classes.remove(classNum);
29         try {
           distances.remove(classNum);
31     }
       catch (Exception e){
33         e.printStackTrace();
       }
35     }

37     // Add element to class:

39    public void add(String digest, int classNum){
           ArrayList<String> temp= classes.get(classNum);
41         classes.remove(classNum);
           temp.add(digest);
43         classes.add(classNum,temp);
       }
45
      // Classify digest:
47
       public static int classify(final String candidate, final int knn){
49
           count();
51         distances = new ArrayList<ArrayList<Integer>>();
```

```
53          for (int i=0; i<classes.size(); i++){
                ArrayList<String> aList = classes.get(i);
55              ArrayList<Integer> theClass = new ArrayList<Integer>();

57              for (int j=0; j<aList.size(); j++){
                    int leve = cmp.DamLev(candidate,aList.get(j));
59                  theClass.add(leve);
                }
61              Collections.sort(theClass);
                distances.add(theClass);
63          }

65          int classify=0;
            double edit=0.0;
67
            for (int i=0; i<distances.size();i++){
69              ArrayList<Integer> aList = distances.get(i);
                double newEdit = discriminant((double)knn, (double)aList.get(knn
                    -1), (double)aList.size());
71
                if (newEdit>edit) {
73                  edit = newEdit;
                    classify=i;
75              }
            }
77          return classify;
        }
79
        static int count=0;
81
        // Count all elements in data set:
83
        private static void count(){
85          for (int i=0; i<classes.size(); i++)
                count+=classes.get(i).size();
87      }

89      // Calculate Discriminant Function:

91      private static double discriminant(double knn, double editDist, double
            numClass){
            if (editDist>0){ // Do not divide by zero
93          double prior = numClass/(double)count;
            double top = knn/numClass;
95          double pnx = top/editDist; // nth estimate of p(x) (PDF)
            double disc = prior*pnx;
97           return disc;}
            else return Double.MAX_VALUE;
99      }
    }
```

B.5 No-Frills Hash

NFHash.java

```
public class NFHash {


    // This makes it faster to do a base 64 number:
    protected static final char[] alphabet = "
        ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/".
        toCharArray();

    final protected int digestSizeBytes = 64;
    final protected int chars = 64;

    protected char[] fingerprint;

    //Mersenne number

    protected static long hash(final long c, long h) {

        long h1=(h)&p4+((h)>>31);
        h1=h<p4 ? (h)&p5+((h)>>19) : h1; // If h<p2 then (h)&p+((h)>>13),
            else h1=h1

        return ((h1)^(~c));
    }

    protected static long p5=524287; // Mersenne primes
    protected static long p4=2147483647;

    final int cutpoint;

    public String hashBytes(final byte[] in) {
        int iter=1;
        int length = in.length;
        fingerprint = new char[digestSizeBytes];

        int j = 0;
        long h2 = 0L;
        long h = setReset();

        for (int i = 0; i < length; i++) {

            int character = (in[i] ) & 255;
            h= rollingCRC(character,in);
            h2 = this.hash(character,h);
            long cmp = h & (cutpoint-1);

            if (cmp == (cutpoint - 1)) {
                // Boundary reached
                fingerprint[j] = alphabet[(int) (h2 & (chars-1))];

                if (j < digestSizeBytes - iter) {
                    h2 = 0x28021967;
                    j+=iter;
```

```
50
                    }
52
                    else {
54                      j=0;
                        iter*=4;
56                  }
                }
58          }

60          if (h != 0) {
                fingerprint[j] = alphabet[(int) (h2 & (chars-1))];
62          }

64          return String.valueOf(fingerprint);
        }
66
        final int winSize;
68      protected long[] rolling_window;

70      public NFHash(final int windowsize, final int cutpoint){
            this.cutpoint=cutpoint;
72          this.winSize=windowsize;
            newByte=new byte[winSize+1];
74      }

76      int start=0;
        byte[] newByte;
78      private long rollingCRC(long c, byte[] b) {
            CRC64 c64 = new CRC64();
80          System.arraycopy(b,start,newByte,0,winSize);
            c64.update(newByte,0,winSize);
82          if (start<b.length-winSize)
                start++;
84          return c64.getValue();
        }
86
        protected long setReset() {
88          rolling_window = new long[winSize];
            start=0;
90          return 0L;
        }
92
    }
```

B.6 Table Generator

tableGenerator.java

```
2   /**
     * Created by Ole on 10.02.2016.
4    *
```

```
 * Generates an 8x256 table for use in Rabin fingerprints.
 *
 */
public class tableGenerator {

    private transient static long[][] superTable;

       // Generate a lookup table by passing an irreducible polynomial into
          generateTable():

    public long[][] generateTable(long poly) {
        long X_degree = 1L << Long.bitCount(Long.MAX_VALUE);

        final long[] preTable = new long[64];

        preTable[0] = poly;
        for (int i = 1; i < 64; i++) {
            long poly = preTable[i - 1] << 1;
            if ((preTable[i - 1] & X_degree) != 0) {
                poly ^= poly;
            }
            preTable[i] = poly;
        }

        superTable = new long[8][256];

        for (int i = 0; i < 256; i++) {
            int c = i;
            for (int j = 0; j < 8 && c > 0; j++) {
                if ((c & 1) != 0) {

                    superTable[0][i] ^= preTable[j];
                    superTable[1][i] ^= preTable[j + 8];
                    superTable[2][i] ^= preTable[j + 16];
                    superTable[3][i] ^= preTable[j + 24];
                    superTable[4][i] ^= preTable[j + 32];
                    superTable[5][i] ^= preTable[j + 40];
                    superTable[6][i] ^= preTable[j + 48];
                    superTable[7][i] ^= preTable[j + 56];
                }
                c >>>= 1;
            }
        }
        return superTable;
    }

    }
```

B.7 Rabin's Fingerprinting Scheme

Rabin.java

```
import java.util.Stack;
```

```
2
  /*
4
  This class can use a precalculated table, or generate one on the fly. The
      former is far more efficient, as Rabin's algorithm
6 on it's own is very simple and efficient.

8
   */
10
  public final class Rabin {
12
      private static final int degree = Long.bitCount(Long.MAX_VALUE) + 1; //
          Degree is always 64 bits
14    private static final long X_degree = 1L << Long.bitCount(Long.MAX_VALUE);
      private final long Prime;
16    private transient static long[][] superTable;

18    public Rabin(final long Prime) {
          this.Prime = Prime;
20        initializeTables();
      }
22
      public Rabin() {
24        Prime=0L;
      }
26
      public void setSuperTable(final long[][] superTable){
28 this.superTable=superTable;
      }
30

32 // This is the easiest constructor to use:

34    public Rabin(final long[][] superTable) {
          this.superTable=superTable;
36        Prime=0;
      }
38
      private void initializeTables() {
40        final long[] preTable = new long[degree];

42        preTable[0] = Prime;
          for (int i = 1; i < degree; i++) {
44            long poly = preTable[i - 1] << 1;
              if ((preTable[i - 1] & X_degree) != 0) {
46                poly ^= Prime;
              }
48            preTable[i] = poly;
          }
50
          superTable = new long[8][256];
52
```

```
            for (int i = 0; i < 256; i++) {
54              int c = i;
                for (int j = 0; j < 8 && c > 0; j++) {
56                  if ((c & 1) != 0) {

58                      superTable[0][i] ^= preTable[j];
                        superTable[1][i] ^= preTable[j + 8];
60                      superTable[2][i] ^= preTable[j + 16];
                        superTable[3][i] ^= preTable[j + 24];
62                      superTable[4][i] ^= preTable[j + 32];
                        superTable[5][i] ^= preTable[j + 40];
64                      superTable[6][i] ^= preTable[j + 48];
                        superTable[7][i] ^= preTable[j + 56];
66                  }
                    c >>>= 1;
68              }
            }
70
        }
72
    // Using an AND mask of 255 keeps the last 8 bits while discarding the rest:
74
        private static long shiftDigest(final long digest) {
76
            long sD = superTable[0][(int) (digest & 0xFF)] ^
78                  superTable[1][(int) ((digest >>> 8) & 0xFF)] ^
                    superTable[2][(int) ((digest >>> 16) & 0xFF)] ^
80                  superTable[3][(int) ((digest >>> 24) & 0xFF)] ^
                    superTable[4][(int) ((digest >>> 32) & 0xFF)] ^
82                  superTable[5][(int) ((digest >>> 40) & 0xFF)] ^
                    superTable[6][(int) ((digest >>> 48) & 0xFF)] ^
84                  superTable[7][(int) ((digest >>> 56) & 0xFF)];

86          return sD;
        }
88
        public long hash(final byte[] preImage) {
90          return hash(preImage, 0, 0);
        }
92
        private static int pad(final int length) {
94          final int starterBytes = length & 7;
            return starterBytes;
96      }


98
        static long hash(final byte[] preImage, int offset, final int footOffset)
             {
100
            final int length = preImage.length;
102         long digest = 0L;

104         final int initOffset = offset;
```

```
            final int pad = pad(length);
106         if (pad != 0) {
                final int max = initOffset + pad - footOffset;
108             while (offset < max) {
                    digest = (digest << 8) ^ (preImage[offset] & 0xFF);
110                 offset++;
                }
112         }

114

            final int max = initOffset + length - footOffset;
116         while (offset < max) {
                digest = shiftDigest(digest) ^
118                     (preImage[offset] << 56) ^
                        ((preImage[offset + 1] & 0xFF) << 48) ^
120                     ((preImage[offset + 2] & 0xFF) << 40) ^
                        ((preImage[offset + 3] & 0xFF) << 32) ^
122                     ((preImage[offset + 4] & 0xFF) << 24) ^
                        ((preImage[offset + 5] & 0xFF) << 16) ^
124                     ((preImage[offset + 6] & 0xFF) << 8) ^
                        (preImage[offset + 7] & 0xFF);
126             offset += 8;
            }
128         return digest;
        }
130

132 }
```

B.8 Merkle Trees

merkle.java

```
  import java.io.Serializable;
2 import java.util.ArrayList;
  import java.util.Collections;
4 import java.util.List;

6 /**
   *
8  * @author Ole
   */
10 public class merkle implements Serializable {

12     bloomFilter bf;
       public static final byte typeTerminal = 0x0;
14     public static final byte typeParent = 0x01;

16     tableGenerator tg = new tableGenerator();
       private  Rabin rs;
18     public Node root;
       public int depth;
20     public int nnodes;
```

```
    public int terminalnodes;

    void setRabin(Rabin rs){
        this.rs=rs;
    }

     public merkle(ArrayList<Long> fingerprints, Rabin rs, ArrayList<Integer>
         tagList) {
        this.rs = rs;
       this.tagList=sortNames(tagList, fingerprints);
        terminalnodes=fingerprints.size();
        bf=new bloomFilter(32,terminalnodes);
        create(fingerprints);
    }

     // Using this constructor is the simplest approach:

     public merkle(ArrayList<Long> fingerprints, final long galois, ArrayList
         <Integer>tagList) {
        this.rs = new Rabin(tg.generateTable(galois));
       this.tagList=sortNames(tagList, fingerprints);
       terminalnodes=fingerprints.size();
        bf=new bloomFilter(32,terminalnodes);  // Adjust the Bloom Filter as
            needed for accuracy
        create(fingerprints);
    }


     /*
     Sort one list according to another:
     */

    public ArrayList<Integer> sortNames(ArrayList<Integer> results,ArrayList<
        Long> digests){
        int tmp2;
        long tmp;
        for (int k=0; k<digests.size()-1; k++) {

            boolean isSorted=true;
            for (int i=1; i<digests.size()-k; i++) {

                 if (digests.get(i)>digests.get(i-1)  ) {
                     tmp=digests.get(i);
                     digests.set(i,digests.get(i-1));
                     digests.set(i-1,tmp);


                    tmp2=results.get(i);
                    results.set(i,results.get(i-1));
                    results.set(i-1,tmp2);

                    isSorted=false;
                }
```

```
70              }
                if (isSorted) break;
72          }
        return results;
74
    }
76


78      ArrayList<Integer>tagList;

80      void setTags(ArrayList<Integer>tagList){
            this.tagList=tagList;
82      }

84      public merkle(ArrayList<Long> fingerprints, final long galois) {
            rs = new Rabin(tg.generateTable(galois));
86          create(fingerprints);
        }
88
        public merkle(Node treeRoot, int numNodes, int height, ArrayList<Long>
            fingerprints) {
90          root = treeRoot;
            nnodes = numNodes;
92          depth = height;
            fingerprints = fingerprints;
94
        }
96
        ArrayList<Long> fingerprints;
98
        void create(ArrayList<Long> signatures) {
100         if (signatures.size() <= 1) {
    //            throw new IllegalArgumentException("Must be at least two
        signatures to construct a Merkle tree");
102         }

104         fingerprints = signatures;
            nnodes = signatures.size();
106         ArrayList<Node> parents = bottomsUp(signatures);
            nnodes += parents.size();
108         depth = 1;

110         while (parents.size() > 1) {
                parents = internalLevel(parents);
112             depth++;
                nnodes += parents.size();
114         }

116         root = parents.get(0);
        }
118


120     public int getNumNodes() {
```

```
        return nnodes;
122     }

124     public Node getRoot() {
        return root;
126     }

128     public int getDepht() {
        return depth;
130     }

132
    ArrayList<Node> internalLevel(ArrayList<Node> children) {
134         ArrayList<Node> parents = new ArrayList<Node>(children.size() / 2);

136         for (int i = 0; i < children.size() - 1; i += 2) {
                Node child1 = children.get(i);
138             Node child2 = children.get(i+1);

140             Node parent = createParent(child1, child2);
                parents.add(parent);
142         }

144         if (children.size() % 2 != 0) {
                Node child = children.get(children.size()-1);
146             Node parent = createParent(child, null);
                parents.add(parent);
148         }

150         return parents;
    }
152
    ArrayList<Node> bottomsUp(List<Long> signatures) {
154         ArrayList<Node> parents = new ArrayList<Node>(signatures.size() / 2);

156         for (int i = 0; i < signatures.size() - 1; i += 2) {
                Node leaf1 = createTerminalNode(signatures.get(i),tagList.get(i))
                    ;
158             Node leaf2 = createTerminalNode(signatures.get(i+1),tagList.get(i
                    +1));


160
                Node parent = createParent(leaf1, leaf2);
162             parents.add(parent);

164         }

166         // in case of odd number of terminal nodes:

168         if (signatures.size() % 2 != 0) {
                Node leaf = createTerminalNode(signatures.get(signatures.size() -
                     1),tagList.get(tagList.size()-1));
170             Node parent = createParent(leaf, null);
```

```
                parents.add(parent);
172         }

174         return parents;
        }
176

178     private Node createParent(Node child1, Node child2) {
            Node parent = new Node();
180         parent.type = typeParent;
             parent.depth=depth+1;
182
            if (child2 == null) {
184             parent.sig = child1.sig;
                } else {
186             parent.sig = internalHash(child1.sig, child2.sig)|internalHash(
                    child2.sig, child1.sig);
            }
188     parent.left = child1;
            parent.right = child2;
190         return parent;
        }
192

194     private Node createTerminalNode(long fingerprint,int tag) {
            Node leaf = new Node();
196         leaf.type = typeTerminal;
            leaf.sig = fingerprint;
198         leaf.tag=tag;
             bf.add(fingerprint);
200         return leaf;
        }
202
        long internalHash(long leftChildSig, long rightChildSig) {
204         String it=(new String((leftChildSig+" "+rightChildSig).getBytes()));
            long h=rs.hash(it.getBytes());
206
          return h;
208     }

210   public static class Node {
            public byte type;  // Terminal or not?
212         public long sig; // Digest
            public Node left;
214         public Node right;
            public int tag; // ID
216         public int depth;
        }
218 }
```

B.9 Tree Comparator

treeComparator.java

```
import java.util.ArrayList;
import java.util.Stack;

/**
 * Created by Ole on 27.02.2016.
 */
public class treeComparator {

    // Get the binary Jaccard (for instance by using the number of terminal
    // nodes in each Merkle Tree

    public double Jaccard(int lengthA, int lengthB, int differ){

        int m=Math.max(lengthA,lengthB);

        int AnB = m-differ;

        if (AnB<0)
            AnB=0;

        double AuB=(double)lengthA+(double)lengthB-(double)AnB;
        double jaccard = AnB/AuB;
        if (jaccard<0)
            jaccard=0;
        return jaccard;
    }

// Compare two Merkle Trees -- regardless of height


    public static int count =0;

    // Tag with name, but order by hash

    public static ArrayList<Integer> differingOne=new ArrayList<Integer>();
    public static ArrayList<Integer> differingTwo=new ArrayList<Integer>();



    static ArrayList<merkle.Node> alNode = new ArrayList<merkle.Node>();


    merkle bloomTree;
    int terminalcount=0;

    public Stack<Integer> bloomCompare(merkle mfirst, merkle msecond){
      neq=0; // Number of different nodes

      nodeStack=new Stack<Integer>();
      if (mfirst.root!=msecond.root){

        int t1=mfirst.nnodes;
```

```
53          int t2=msecond.nnodes;
            merkle referenceTree=mfirst;
55          bloomTree=msecond;
            nodeStack= new Stack();
57
            if (t2>t1){
59              referenceTree=msecond;
                bloomTree=mfirst;
61          }
            bloomTraverse(referenceTree.root.left,bloomTree.root.left);
63          bloomTraverse(referenceTree.root.right,bloomTree.root.right);


65
        }
67
        return nodeStack;
69
    }
71

73  public void bloomTraverse(merkle.Node root){


75
        if (root.left!=null)
77          bloomTraverse(root.left);
        if (root.right!=null)
79          bloomTraverse(root.right);

81      if (root.type==0x00 && !bloomTree.bf.contains(root.sig))
        {
83          neq++;
            nodeStack.push(root.tag);
85      }
    }
87
    int neq=0;
89
    public void bloomTraverse(merkle.Node root, merkle.Node root2){
91
        if (root!=root2)
93      {
            if (root.left!=null && root2.left!=null)
95              bloomTraverse(root.left,root2.left);
            if (root.right!=null && root2.right!=null)
97              bloomTraverse(root.right,root2.right);


99
            if (root.left!=null && root2.left==null)
101             bloomTraverse(root.left);
            if (root.right!=null && root2.right==null)
103             bloomTraverse(root.right);


105
```

```
            if (root.type==0x00 && !bloomTree.bf.contains(root.sig))
107         {
                nodeStack.push(root.tag);
109             neq++;
            }}
111
        }
113
        public Stack<Integer> nodeStack;
115


117 }
```

C Brief Example of Project Implemented with k-NN and NFHash

A simple implementation can be found in the enclosed class "knn.7z". This archive features both a runnable jar file — and the NetBeans source code.