TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Pavel Tšikul 163324IVCM

ENCRYPTED DATA IDENTIFICATION BY INFORMATION ENTROPY FINGERPRINTING

Master's Thesis

Supervisor: Pavel Laptev
BSc

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Pavel Tšikul 163324IVCM

KRÜPTEERITUD ANDMETE IDENTIFITSEERIMINE INFORMATSIOONI ENTROOPIA SÕRMEJÄLJESTAMISE TEEL

Magistritöö

Juhendaja: Pavel Laptev
BSc

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Pavel Tšikul

07.01.2019

Abstract

The aim of this thesis is to develop a way of encrypted data identification by means of entropy features analysis with reliable mechanism of distinguishing it from extremely compressed information. The outcome of this work is a solid method to confidently recognize encryption patterns and a set of tools that provides the user with the developed functionality. To achieve this goal a thorough analysis of information randomness and its features will be performed. A series of experiments will identify some interesting entropy feature correlations. These results will later become a base for a machine learning approach to identify underlying principles. Finally, a forensic tool will be developed utilizing previously developed methods. A series of validation experiments conclude the work with proper evaluation and notes for future research.

This thesis is written in English and is 78 pages long, including 7 chapters, 25 figures and 7 tables.

Annontatsioon

Krüpteeritud andmete identifitseerimine informatsiooni entroopia sõrmejäljestamise teel

Selle töö eesmärgiks on luua viis krüpteeritud andmete identifitseerimiseks entroopia analüüsiga kasutades usaldusväärset meetodit selle tuvastamiseks äärmiselt kokkupakitud informatsioonist. Selle töö tulemuseks on usaldusväärne meetod tuvastamaks entroopiamustreid ning tööriistad, mis võimaldavad kasutada loodud funktsionaalsust. Selle eesmärgi saavutamiseks sooritatakse põhjalik analüüs informatsiooni juhuslikkusest ja selle omadustest. Mitmed eksperimendid identifitseerivad märkimisväärseid korrelatsioone entroopia omadustes. Nende eksperimentide tulemused saavad aluseks masinõpet kasutavale meetodile tuvastamaks fundamentaalseid printsiipe. Lõpetuseks luuakse ekspertiisvahend kasutades varemalt loodud meetodeid. Mitmed valideerivad eksperimendid võtavad töö kokku järelduste ning märkustega edasiste uuringute läbiviimiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 78 leheküljel, 7 peatükki, 25 joonist, 7 tabelit.

List of abbreviations and terms

AES	Advanced Encryption Standard
API	Application Programming Interface
CNN	Convolutional neural network
CS	Chi-Square test
DT	Decision tree
DES	Data Encryption Standard
GPG	GNU Privacy Guard
KNN	K-nearest neighbors
LZ	Lempel-Ziv
LZMA	Lempel–Ziv–Markov chain algorithm
LZSS	Lempel–Ziv–Storer–Szymanski
ML	Machine learning
PGP	Pretty Good Privacy

Table of contents

Author's declaration of originality ... 3
Abstract ... 4
Annontatsioon ... 5
List of abbreviations and terms ... 6
Table of contents ... 7
List of figures ... 10
List of tables ... 12
1 Introduction ... 13
1.1 Problem ... 13
1.2 Purpose ... 16
1.3 Thesis Overview ... 18
2 Current Works Evaluation ... 19
2.1 Entropy-based Encryption Classification Works Overview ... 19
2.2 Craig Heffner's Solution Analysis ... 20
2.3 Seunghun Cha and Hyoungshick Kim's Solution Analysis ... 22
2.4 Conclusion ... 23
3 Entropy and Randomness Tests ... 24
3.1 Information Entropy and Randomness ... 24
3.2 Randomness Tests Overview ... 26
3.2.1 Shannon Entropy ... 26
3.2.2 Monte Carlo Pi Approximation ... 27
3.2.3 Chi-Square Test ... 28
3.2.4 Arithmetic Mean ... 29
3.3 Methodology ... 30
3.4 Results ... 31
3.5 Conclusion ... 33
4 Encrypted and Compressed Data Entropy Features Research ... 34
4.1 Encryption Formats Overview ... 34
4.1.1 Block Ciphers ... 34

4.1.2 Cipher Text Indistinguishability from Random Noise 35
4.2 Basics of Data Compression and Modern Day Applications 37
4.2.1 Early Works 37
4.2.2 Huffman Coding 38
4.2.3 Lempel-Ziv Compression 39
4.3 Test Data Overview 41
4.4 Tools Selection 45
4.4.1 Entro.py Library and Tool 45
4.5 Methodology 47
4.6 Results 51
4.7 Conclusion 53
5 Machine Learning Model Training 54
5.1 Tools Selection 54
5.1.1 NumPy 54
5.1.2 Pandas 54
5.1.3 Scikit-Learn 55
5.2 Methodology 56
5.2.1 Data Selection and Preparation 57
5.2.2 Used ML Models Overview 59
5.2.3 Preliminary Experiments 60
5.2.4 Other Models and Model Fine Tuning 60
5.3 Decision Tree Analysis and Results 61
5.4 Conclusion 63
6 Sleuth Kit Autopsy Module Development 64
6.1 Autopsy Python Eco-System 64
6.2 Existing Encryption Detection Module Analysis 66
6.3 Ingest Module Functional Specification 67
6.4 Implementation Details 68
6.5 Results 70
6.5.1 Synthetic Test 70
6.5.2 Realistic Test 72
6.6 Conclusion 74
7 Summary and Conclusions 75
7.1 Conclusion and Future Work 75

References 77
Appendix 1 – Entro.py Library Source Code 80
Appendix 2 – Software and command line commands used to generate test data......... 83
Appendix 3 – Full Decision Tree structure 84
Appendix 4 – Machine Learning Tests Source Code (cropped) 85
Appendix 5 – Autopsy Module Source Code (cropped) 88

List of figures

Figure 1. Information entropy of encrypted and unencrypted data. 14
Figure 2. Information entropy of encrypted and highly compressed data..................... 15
Figure 3. Shannon's Entropy equation. 26
Figure 4. Monte Carlo Pi approximation.. 27
Figure 5. Probability of a point being placed inside a circle. 27
Figure 6. π dependency on the probability of a point placed inside a circle. 27
Figure 7. Monte Carlo estimated π. 28
Figure 8. Chi-Square equation.. 28
Figure 9. Calculating Chi Square for random byte distribution. 29
Figure 10. Arithmetic mean equation. 29
Figure 11. Comparison of Shannon's entropy and Chi-Square sensitivity. 31
Figure 12. AES encryption pipeline. 35
Figure 13. Huffman binary tree. 39
Figure 14. Entropy rate equation. 40
Figure 15. Example chi square graph for a binary blob of 600 x 32B pieces. 47
Figure 16. Maximum chi square values for CAST5 and AES file sets. 48
Figure 17. Occasional oscillation in encrypted data entropy... 48
Figure 18. LZSS sample chi square distribution. 49
Figure 19. LZMA sample chi square distribution. 50
Figure 20. Confidence Graph. Initial idea (dropped out). 52
Figure 21. Classification of ML tasks and algorithms. 56
Figure 22. k-fold cross-validation with k = 4. 58
Figure23. Trained Decision Tree structure (cropped). 62
Figure 24. Autopsy Encryption Detection module decision logic................................. 66
Figure 25. Decision Tree extracted as Python code. 68
Figure 26. Autopsy module logic flowchart... 69
Figure 27. Entro.py ingest module results in Autopsy blackboard. 71

List of tables

Table 1. Example of fixed-size ASCII coding. 38
Table 2. Example of VLC. 39
Table 3. Tested formats cut-off values. 44
Table 4. Example of entropy features data spreadsheet. 58
Table 5. Prediction precision by model. 61
Table 6. Encryption and compression detection results. 71
Table 7. Realistic test results. 73

1 Introduction

This chapter gives a brief introduction to the project, describes the problem, proposes a solution for it, and identifies possible application areas.

1.1 Problem

Identification of unfamiliar data chunks poses a serious challenge in digital forensics. When signature- or hash-based approaches fail researchers are only left with assumptions originating from different file characteristics and information entropy is one of the most important of them.

The term entropy was first used in statistical thermodynamics and refers to the amount of uncertainty or disorder. In Information Theory however, entropy of data is an average measure of some stochastic system which defines how much information it produces. This definition was proposed by Claude Shannon in his 1948 work "A Mathematical Theory of Communication" [1]. What that effectively means is that entropy defines the minimum number of bits on average needed to represent an arbitrary data stream.

Consider a stream consisting of only 2 symbols with equal probability of occurring. Calculated entropy value (entropy rate) will be equal 1 that is 1 bit per symbol is needed on average to encode this type of data. However if we shift the probability of occurrence as 70/30 the entropy will lower down to 0.88 bits/symbol. Continuing to shift the probability to 99/1 will transform entropy to 0.08 bits per symbol on average is required. Practically this means that encoding a 100-symbol string will require only 8 bits. This is due to the fact that 99% of the data will be represented by the first symbol (refer to Chapter 3 for more details on entropy).

For typical 8-bit bytes Shannon's entropy lays in range [0, 8]. Depending on the type of the data is possible to do quite accurate assumptions on the origin and/or state of the data based on its entropy value. For example English plain text file will usually have entropy value around 4, whereas a binary executable module is likely to be around 6.

The entropy distribution inside files is most of the time uneven and can be presented as a line graph or a chart (see Figure 1).

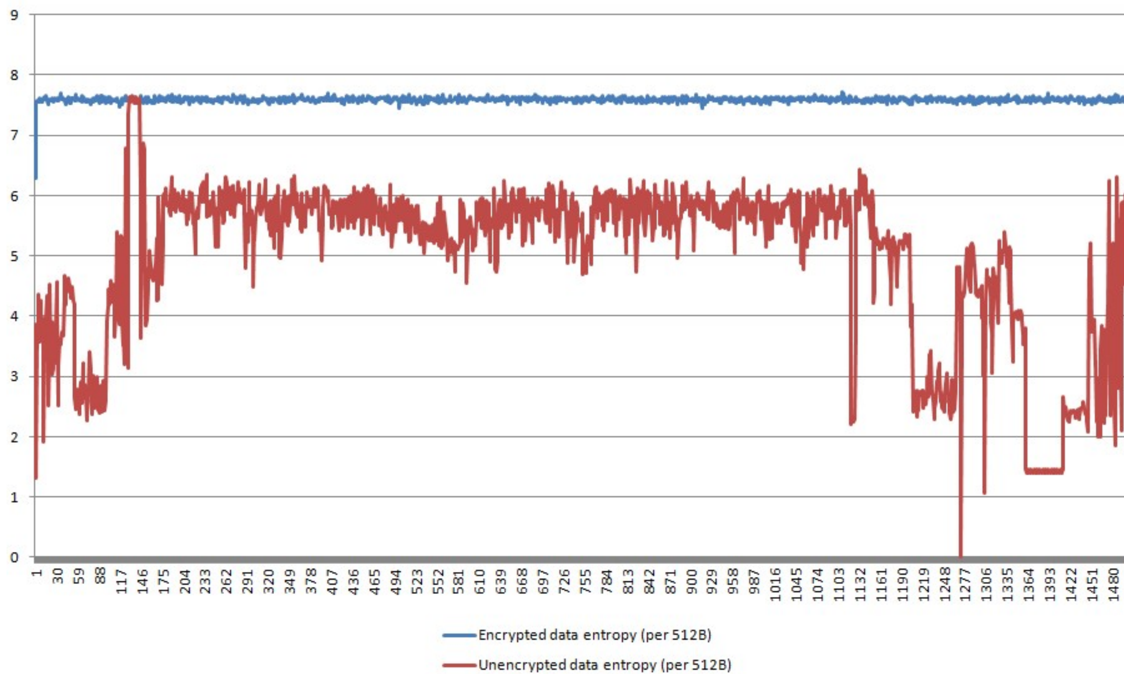


Figure 1. Information entropy of encrypted and unencrypted data.

There are three cases however which present a very challenging, yet very interesting problem. Highly compressed, encrypted, and truly random data will produce almost a flat line of evenly distributed high entropy tending to maximum value of 8 (see Figure 2) making it fairly impossible to distinguish using only graph visualization.

In this work the author tried to tackle this problem in regards to encrypted data identification and successful distinguishing it from any other types of data basing on the anomalies in entropy graphs. Since compressed data is often presented without proper headers or trailing checksums the signature-based recognition becomes useless. Examples of header-less compression are:

- Hardware firmware images, which sometimes manage compression by utilizing external functions which store compressed data sizes and checksums externally [5].

- It is estimated that 80 to 90 percent of malware samples contain some form of either encrypted or compressed data to hide its internal text or binary instructions [3].

- Any compressed data produced with “no-header”-options by tools like libzip/gzip or lzma. This approach is often used for internal software resources compression.

- HTTP traffic packets (gzip compression).

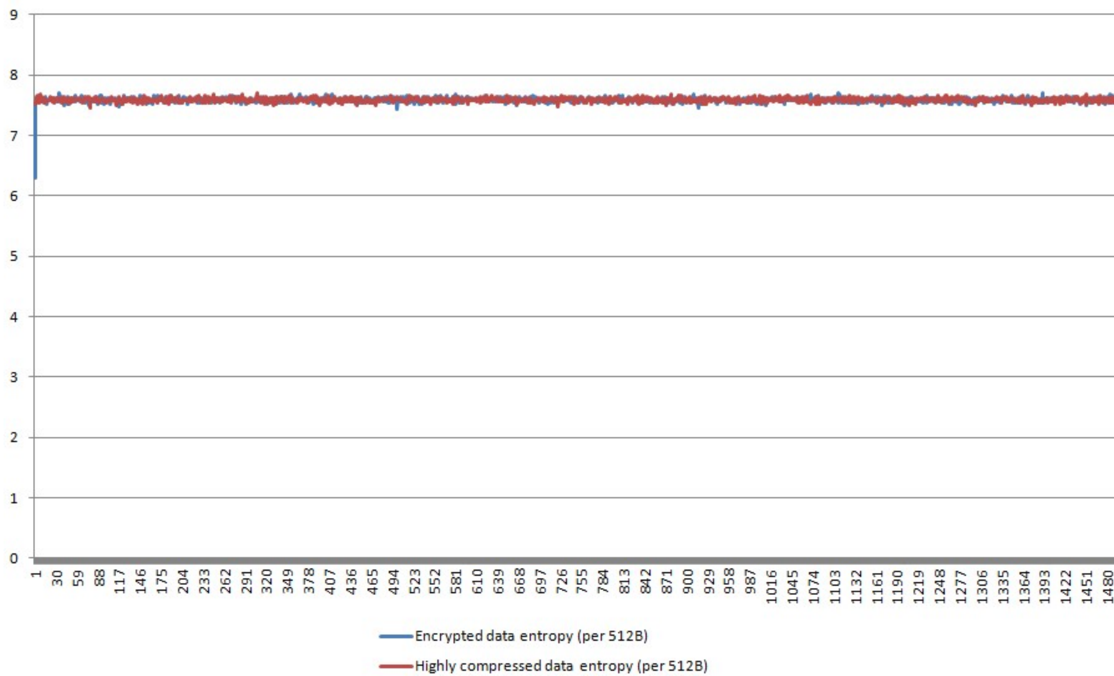


Figure 2. Information entropy of encrypted and highly compressed data.

Currently the number of studies related to distinguishing between highly compressed and encrypted data is very limited. Some preliminary studies found were performed some time ago [5][2][25][26], but they lack serious test-sample volume and their results were limited and sometimes speculative.

The idea of this work is to carefully evaluate different methods of information randomness measurement to provide a solid method of differentiation of encrypted data.

1.2 Purpose

The resulting method proposed in this work can be applied in a wide range of areas:

- Digital forensics. This area was a primary target at the moment of writing this work and mostly assumes post-mortem or offline analysis.
    - "Ransomwared" files identification. Nowadays ransomware is on the rise. It is now a mature major threat with millions of victims all over the world every year. Using the proposed method it is very easy to identify files encrypted by the malware and isolate the ones intact.
    - Hidden encrypted storage disclosure. Sometimes hidden encrypted volumes such as Hidden Volumes, RIPA, or TrueCrypt can be stored as files or as part of files and entropy tests can reveal such storages pretty easily.
    - Detection of encrypted malware. Entropy probing especially with the graph view can be successfully used to detect executables which content was purposely encrypted to be later decrypted in memory. This approach is often used by malware creators to bypass antivirus engines' signature comparison.
- Intrusion Detection Systems (IDS). With some effort put into the overall performance of the proposed method it can be potentially used for real time traffic samples analysis.
- Antivirus applications. The concept proposed for "post-mortem" digital forensics ransomwared file detection can be applied to real-time ransomware attack detection. Another AV-related field is real time sample analysis (e.g. conclusion on whether an executable in question contains any encrypted parts).

The method described in this work can be applied to all of the above-mentioned areas; however some additional work will need to be done in order for the performance to improve, as current state of the prototype does not allow efficient real-time processing due to extensive IO operations and CPU-heavy mathematical

computations. The author is currently looking for ways to bypass some of the algorithmic “bottlenecks” (see Chapter 7 for more details).

1.3 Thesis Overview

Presented thesis is divided into following chapters:

- **Chapter 2** is covering currently available works on the topic.
- **Chapter 3** focuses on the selection of appropriate randomness test for proposed solution.
- **Chapter 4** covers the case study of data encryption and compression in regards to this study, as well as practical approach to research of entropy features in encrypted and compressed data streams.
- **Chapter 5** is about utilizing the research results from Chapter 4 to build an encryption detection mechanism based on machine learning algorithms.
- **Chapter 6** describes development and testing of a sample application based on the results acquired in previous chapters.
- **Chapter 7** concludes achieved results and proposes future work directions.

2 Current Works Evaluation

Although many of the sources state that it is either very hard or even impossible to determine whether the binary blob was either compressed or encrypted [4][24], the author managed to find some works trying to approach this task. This chapter covers currently available works on the topic and their analysis.

2.1 Entropy-based Encryption Classification Works Overview

Encryption classification based on entropy is the most common method used in many different areas and numerous papers exist on the topic. The most significant of those will be covered in this section.

Dorfinger and Panholzer in their work "Entropy Estimation for Real-time Encrypted Traffic Identification" tried to approach this problem by performing an entropy comparison of a packet with the estimated entropy of an evenly distributed random block of the same size [24]. In case the total difference is above some given threshold the packet is treated as unencrypted. The numbers reported in the paper are very promising; however the tests proposed are quite unrealistic and do not account compressed traffic.

Two works that appeared after Dorfinger's are "Clear and Present Data: Opaque Traffic and its Security Implications for the Future" by White et al. [25] and "Detecting encrypted botnet traffic" by Zhang et al. [26]. They both tried to tackle the problem and did account for compressed data as well. Although compression was included in the experimental data both works indicate that effective distinguishing between two data types is not possible by their means and mark it as a limitation since both can have a very similar entropy estimation.

There are however two works that tried to approach the problem from different angles and they will be covered in more detail in upcoming sections.

2.2 Craig Heffner's Solution Analysis

Craig Heffner is a well-known security specialist, vulnerability researcher, and hacker presenting at such events as Black Hat and DEF CON. His series of articles named "Differentiate Encryption From Compression Using Math" [5][6] was the initial trigger for this work to appear.

The main idea stated in these articles is that while encrypted data is more stably distributed by entropy, compressed data tends to have some deviations. He then presents a research trying to prove this statement.

He researched two randomness tests which are Chi-Square and Monte Carlo Pi in order to identify the claimed deviations. As a result he identified Chi-Square value of 512 as a threshold for encryption entropy, i.e. files that have spikes over that value should be considered compressed.

Major flaws in Heffner's theory found by the author are:

- Feeding the whole file, including any signature, header and/or footer data, leads to inaccurate results, since those chunks usually have more predictable data, hence their entropy will be obviously less than of the data in question itself creating a disturbance in the graph.
- Heffner's test set was comprised of relatively small group of files (380 files in total) of sizes from 300KB to 15MB. All of the files were either compressed or encrypted firmware packages.
- The ranking system introduced by Craig Heffner seems to be too basic and over-simplistic. It accounts only total number of deviations occurring in a file completely ignoring other factors, such as file size and entropy distribution levels.

Already some preliminary tests on some different sizes of encrypted files showed that the "range of encryption" (Chi-Square value less than 512) is inaccurate since many of tested samples occasionally showed higher Chi-Square values. But, what is more important, compressed files can often fall under the features identified by Heffner to be encryption-only, i.e. they don't show any deviations at all, especially when the decrease

in data size. Thus, Heffner's claim of "…98% of the compressed files tested were correctly identified as compressed, and 100% of the encrypted files were identified as not compressed (i.e., encrypted)" is at least not accurate.

2.3 Seunghun Cha and Hyoungshick Kim's Solution Analysis

This fairly new work named "Detecting Encrypted Traffic: A Machine Learning Approach" was published in March 2017 [2]. It covers an effective method of encrypted network traffic separation in Intrusion Detection Systems. Since real time conventional deep packet inspection techniques are ineffective applied to encrypted traffic they propose to do the encryption classification on the fly and perform the analysis only on the regular unencrypted data, leaving the encrypted one for offline research. To achieve this goal they introduced several randomness tests to estimate the randomness of a packet and a machine learning classifier. The three randomness tests used are:

- Shannon entropy.
- Chi-Square test.
- Arithmetic mean.

Results of these tests run on a single packet are then used as a training data for ML classifiers of four different types.

In general this work thoroughly examines the topic with the only downside been the narrow application to traffic analysis and not accounting the small fluctuations of entropy inside the data stream. The latter is obviously due to the overall small size of the samples used in the experiments (the absolute maximum for a TCP packet is 64KB). So the method could potentially work well for the network analysis, but render very limited in a more general way.

2.4 Conclusion

In this chapter a thorough analysis of available solutions was conducted. Many interesting ideas to be used in this work were identified alongside the weak points and misconceptions in them. Main ideas that were highlighted include:

- There are potentially patterns that could help distinguish highly-compressed data from encrypted.
- Machine learning approaches may be very helpful when dealing with large amounts of complex data.

Now that the downsides of current solutions were identified the pathway to improve and widen the approach became more clear.

3 Entropy and Randomness Tests

This chapter contains a case study on entropy, gives an overview of evaluated randomness tests, and collected results.

3.1 Information Entropy and Randomness

Due to the nature of encryption processes (see Chapter 4 for more details) the data produced by them tends to be as random as it can theoretically be. This is done on purpose so that the produced data is indistinguishable from random noise. On the other hand compression algorithms tend to find the most optimal ways to store the data in the minimum number of bits as possible striving for the Shannon's entropy limit or the maximum compression possible according to Information Theory.

Thus, the main idea utilized throughout this work is: *we account for encrypted data to be equal in entropy to truly random one and for compressed data to be striving towards the same state but having some minor flaws or inaccuracies in randomness distribution due to imperfections of general-purpose compression algorithms in an attempt to reach maximum entropy*. With that in mind we need to check how truly random a presented data blob is and if the data tends to be not random enough treat it as compression. It should be considered encrypted or truly random otherwise.

Aside of Shannon's equation a number of other randomness tests exist. This list includes, but not limited to:

- Shannon's entropy
- Maurer's universal statistical
- Cumulative sums (cusum)
- Random Excursions

- Overlapping Permutations
- Monte Carlo Pi Approximation
- Reverse arrangements
- Chi-Square test
- Frequency test within a block
- Arithmetic Mean
- Parking lot
- Binary rank test for 32x32 matrices
- Discrete Fourier transform (spectral)

This list can be continued on and on. During the course of evaluation more than 40 algorithms were considered for the best randomness estimation.

3.2 Randomness Tests Overview

Through the years many practical ways to measure data randomness were invented and evaluated. Most of the currently used randomness tests are parts of test collections such as George Marsaglia's Diehard Battery of Tests, NIST, and ENT.

A work "Random Number Generators: An Evaluation and Comparison of Random.org and Some Commonly Used Generators" by Kenny was found very useful while selecting candidates as it provided a comprehensive analysis of the whole range of randomness tests currently available [27].

Other works in this field ([5][6][2]) provided additional information on the best algorithms and practices currently used in digital forensics and reverse engineering.

3.2.1 Shannon Entropy

Shannon Entropy equation was an obvious candidate for the proposed solution since it is a traditional method for randomness estimation. It is defined by the classic Shannon's equation:

$$H = -\sum_{i=1}^{n} p_i log_2(p_i)$$

Figure 3. Shannon's Entropy equation.

Applied to 8-bit bytes it practically means:

- Calculate every byte's occurrence count.
- Divide those occurrence counts by the data blob length.
- Multiply every resulting item by its logarithm.
- Sum up all of the results.

This will output a value in range [0, 8] with the higher values meaning higher entropy.

3.2.2 Monte Carlo Pi Approximation

The idea of approximating Pi using Monte Carlo method is pretty simple. Consider a circle of radius r inside a square with a side of *2r* (see Figure 4). With this defined the area of the circle is $S_c = \pi r^2$ and the area of the square is $S_s = 4r^2$.

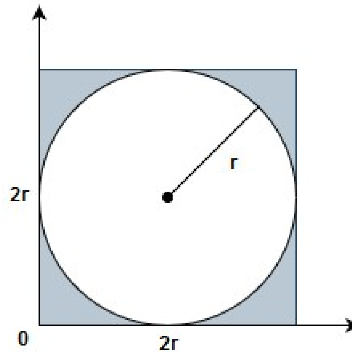


Figure 4. Monte Carlo Pi approximation.

Now if you start generating random pairs of numbers in the range [0, r] and treat them as (x, y) point coordinates inside the square then the probability of a point to be placed inside the circle is:

$$P_c = \frac{\pi r^2}{4r^2}$$

Figure 5. Probability of a point being placed inside a circle.

Simplifying this equation will result in:

$$\pi = 4P_c$$

Figure 6. π dependency on the probability of a point placed inside a circle.

At this point Pi simulation now takes pairs of bytes from the stream and checks whether or not they are inside the circle area and increment a corresponding counter N_c. The check is done with a simple comparison of whether $x^2 + y^2 \leq r^2$. At the same time the square area counter N_s is incremented every time (since all of the points are inside the square). The final equation is then as follows:

$$\pi_{MC} = \frac{4N_c}{N_s}$$

Figure 7. Monte Carlo estimated π.

The result is a Monte Carlo estimated value of Pi. And the main consequence in regards to randomness is *the closer the estimated value to canonic π the more randomly the data is distributed in a data set*. This deviation can be calculated as an absolute percentage of difference between estimated and canonic values.

One concern regarding this method that was identified on the early stage is that to get good precision the number of samples should be significant. Although the Pi simulation stops being stochastic after about 3000 samples the desired precision is acquired only around 100000-500000 samples. So even if MC Pi approximation is to be used in the future the sample count should be taken into account and the results adjusted accordingly.

3.2.3 Chi-Square Test

Chi-Square test is a universal method to check if some variation in the collected data follows a proposed theoretical prediction. The general equation for chi square is presented in Figure 8.

$$x_c^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

Figure 8. Chi-Square equation.

The meaning of the equation can be described as follows: E is a range of expected values and O is a range of practically observed values; so the sum of all the $\frac{(O_i - E_i)^2}{E_i}$ division results of corresponding expected and observed values will give the x_c^2 or the Chi Square value.

In our concrete case the null-hypothesis is that in truly random data all 256 byte values will be equally distributed and are expected to have equal probabilities of appearing in the data stream. Thus the probability for every value is the same and equals P_e = *data_length / 256*. To apply Chi Square for this occasion we need to collect the real distribution of all 256 byte values and calculate chi square as shown in Figure 9.

$$x_c^2 = \frac{(\mathrm{O}[0] - \mathrm{P_e})^2}{\mathrm{P_e}} + \frac{(\mathrm{O}[1] - \mathrm{P_e})^2}{\mathrm{P_e}} + \cdots + \frac{(\mathrm{O}[255] - \mathrm{P_e})^2}{\mathrm{P_e}}$$

Figure 9. Calculating Chi Square for random byte distribution.

General interpretation of the chi square results for this exact case is that truly random data will usually have an average chi square value of 224 with oscillations in range 170 – 490 depending on data size and histogram step. Everything above this range may be questioned for randomness.

3.2.4 Arithmetic Mean

Arithmetic mean is calculated by summing up all byte values of the data source and then dividing them by the total length of the data.

$$A = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Figure 10. Arithmetic mean equation.

In a truly random (i.e. uniformly distributed) data blob the result of arithmetic mean should lay around value of 127.5.

3.3 Methodology

The process of selection a randomness test for this particular case consisted of 3 main steps:

- Running randomness graph generation for every algorithm on a relatively small file set of 50 items of different size. File set was comprised of AES encrypted and LZMA compressed files with 50/50 distribution.
- Careful analysis of graphs side by side in order to identify a method with more sensitivity to small fluctuations in randomness.
- Iteratively decreasing graph step from 512B to 32B and re-evaluating results.

This process generated a lot of interesting results, some of which were not utilized in this work, but could potentially be used in a later research.

3.4 Results

All four algorithms in question provided more or less equal results on average and then were ranked by two criteria:

- Stability of the histogram provided.
- Sensitivity to small deviations in randomness distribution.

For the stability test a set of truly random data samples different in sizes was generated using random.org service, which uses atmospheric noise as randomness seed. As a result it was observed that Monte Carlo Pi method is not stable enough on samples less than 4KB. So it was discarded from future evaluation at this stage.

For the deviation sensitivity a procedure described in section 3.3 was used. To make the results a bit more predictable an additional series of tests were run on a subset where random noise was artificially altered with some less random data at specific locations.

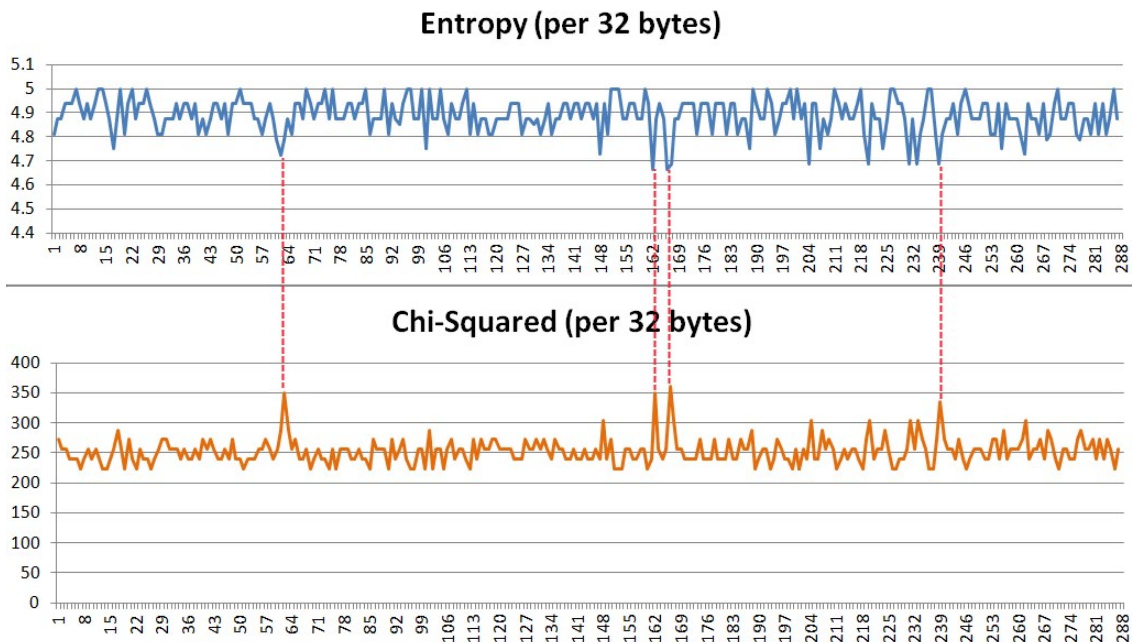


Figure 11. Comparison of Shannon's entropy and Chi-Square sensitivity.

The results of these tests have revealed all remaining three methods are quite similar in performance, yet Chi-Square test is most of the time better in identification of tiny fluctuations in randomness. This confirms hypothesis done by Craig Heffner in [5][6].

In regards to the histogram step size although it was clears that to get a more reliable symbol distribution for 8-bit byte the average size of a sample should be at least 2.5KB this size tends to hide important fluctuations in randomness. Parallel tests run for three algorithms showed that lowering the size of the sample all the way down to 32B does not affect the overall precision, but only the magnitude of the readings.

3.5 Conclusion

In this chapter a study on different randomness tests was conducted in an attempt to identify the most precise way to detect anomalous behaviour of entropy inside a data stream. As a result a Chi Square test was selected as the most promising as well as some practical aspects of its application (histogram step size). At this point the results seem to be promising; however there is a large field for future investigation in regards to randomness tests comparison.

4 Encrypted and Compressed Data Entropy Features Research

This chapter describes a case study on popular compression encryption formats, the iterative approach to entropy features research and tools improvement, as well as experimental data collection.

4.1 Encryption Formats Overview

This section covers some of the modern encryption algorithms and explains how the data transformation happens, how that affects the entropy of a resulting data stream, and what specific features these algorithms incorporate.

4.1.1 Block Ciphers

All of the algorithms used in this work, except for ENIGMA, are parts of block cipher family. The main feature of this family is that the cryptographic key and algorithm are applied to a fixed-size block of data, rather than to one bit a time as in stream ciphers.

To better understand how exactly the data inside encrypted files is composed the encryption mechanism on the example of AES or Advanced Encryption Standard will be covered. The concept of AES internal operation utilizes ideas common to the majority of modern block ciphers.

The AES main operation cycle is depicted in Figure 12. The original plain data is chopped into blocks of 128 bits each. Every such block is then transformed into 128 bits of cipher data on the output so that the final encrypted data is exactly the same size as the original. Inside the AES system there lays a series of basic cryptographic primitives: substitution, transposition, and bitwise operations [16]. Every next operation in the chain takes the result of previous one and applies transformation on top of it. A special sub-key generator takes the original cryptographic key to produce a different sub-key for every operation in the pipeline. The very first step (sometimes called “whitening”) is

a simple bitwise XOR performed on the original block data with the dedicated sub-key from the generator.

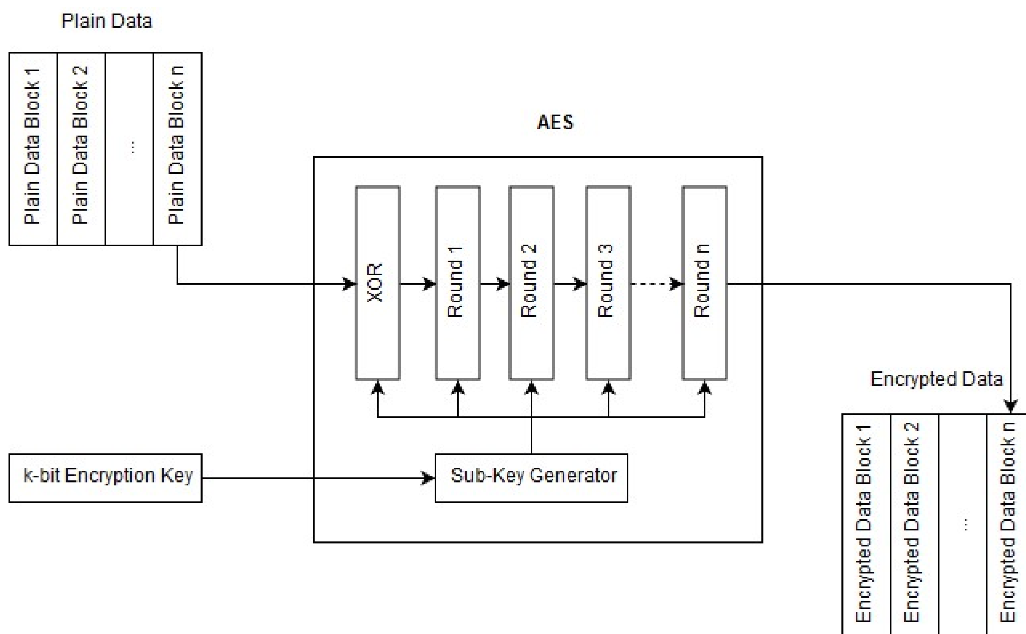


Figure 12. AES encryption pipeline.

XOR-step is then followed by a series of so called rounds which minimum count is 10 and can be infinitely increased to gain better security, but sacrificing the performance. Inside every round the following series of operations occur:

- Static Substitution – byte-wise fixed table replacement.
- Transposition – byte-wise repositioning of bytes.
- Dynamic Substitution – 4-byte-wise formula-based replacement.
- XOR – simple bitwise XOR with the sub-key.

So every round performs these operations with its dedicated sub-key making it a challenging task to roll back all the bits permutations in a brute-force attempt to crack the cipher.

4.1.2 Cipher Text Indistinguishability from Random Noise

Some ciphers intentionally try to make the cipher text to be indistinguishable from random noise bits. This is done for various purposes:

- Deniable encryption. This means that the existence of the message can be denied if the adversary cannot prove otherwise.

- To conceal encrypted data better as a random noise on hard drive. These techniques are often used by volume encryption software like TrueCrypt to hide the data as some binary garbage on a disk.

- To complicate traffic analysis.

However to achieve indistinguishability from random noise in general case it is enough to perform a XOR-operation with some pseudorandom stream of the same length as the original data. Most modern block ciphers do this in the “whitening” stage (DES, 3DES, AES, BLOWFISH, etc.) using the modified key from the sub-key generator as a random stream. The numerous bits permutations result in data effectively looking as random noise with the highest entropy possible. The fact that a secure block cipher can act as a cryptographically secure pseudorandom number generator (CSPRNG) approves this opinion [28].

4.2 Basics of Data Compression and Modern Day Applications

This subsection covers the fundamentals of the lossless data compression and some popular algorithms used in modern archiving solutions.

4.2.1 Early Works

Even starting from the Morse code there were attempts to minimise the size of a message to reduce the time needed for sending it. To achieve this goal the statistical probabilities of English letters appearing in the text were taken into account when assigning the dot-dash combinations to letters, so that the more likely it is for the letter to appear in text the shorter combination was assigned to it. That is why letters E and T were assigned the shortest combinations of single dot and single dash respectively.

What Claude Shannon developed with the concept of entropy was a mean of content measurement in a data stream. This basically means the entropy is the minimum amount of bits needed in average per symbol to represent the given data stream. So, according to Information Theory, the entropy multiplied by the length of the stream gives the minimum expected size of this stream in a compressed state. So data compression tends to minimize data size by maximizing the amount of information per bit in the data and increasing its entropy rate.

Regular approach to coding a symbol is to have a fixed-size block for each symbol and to have a mapping table for every symbol to decode it back. A good example of such approach is ASCII table. It utilizes 7 bits to encode each character, which gives a maximum of 128 symbols to be coded (see Table 1). This approach is very convenient for real-time manipulation of the data; however it is not very efficient in terms of information capacity as seen from Shannon’s equation. This comes from the excessive bits needed to be stored for every character. So the idea of variable-length codes or VLC’s appeared. The main idea is pretty straight-forward: every unique symbol is mapped to a code comprised of a variable number of bits with tendency to minimize the size of each symbol.

Letter	Binary Code
A	0100 0001
B	0100 0010
C	0100 0011
D	0100 0100

Table 1. Example of fixed-size ASCII coding.

To make use of VLC's and effectively eliminate any discrepancies during data decoding they have to possess a feature known as "prefix property" [13]. The idea is that whenever a code is assigned to a symbol, no other code may start with the same pattern. For example if letters (A, B, C) are coded with the following bit patterns (0, 10, 101), the bit stream [1010] can be interpreted dually as (BB) or (CA), which breaks the decoding process.

Peter Elias introduced a series of designs for variable-length codes [12] following the prefix property rule. The very first method, called Unary coding represents any given integer n as a $n - 1$ ones followed by a zero, so that 1 will be coded as "0" and 3 will become "110". For obvious reasons this code is not very optimal for large integer values. He went on with development of different VLC's (namely Gamma, Delta, and Omega codes) targeting at different probabilities of the data set sizes, so that the user could estimate it and pick an optimal encoding for the specific case.

4.2.2 Huffman Coding

Many other works aside of Elias' were related to optimal VLC assignment (Stout Codes, Boldi–Vigna Codes, Taboo Codes, etc. [14]). But one work became a breakthrough, so that Donald Knuth referred to it as "one of the fundamental ideas that people in computer science and data communications are using all the time" [15]. In 1952 David Huffman in his work "A Method for the Construction of Minimum-Redundancy Codes" described a mechanism of effectively minimising the number of bits needed to code an arbitrary message [11] following the prefix property rule.

The main idea is to build a binary tree from down to top basing on the probability of symbol occurring. To quickly demonstrate the method we'll take an array of letters with corresponding probabilities (A: 50%, B: 12,5%, C: 12,5%, D: 25%) and sort all the symbols by their probability of occurrence from highest to lowest. We then take the

least probable symbols and combine them into a node. Iteratively repeating this process we reach the root of the tree (see Fugure 13).

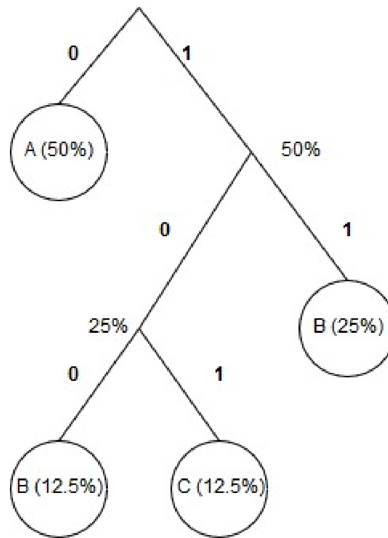


Figure 13. Huffman binary tree.

Branches of the tree are then assigned with either 1 or 0. To get a symbol code from the binary tree it is needed to collect branch values all the way from to top to the symbol in question. For our case the codes will look as follows:

Symbol	Huffman Code
A	0
B	100
C	101
D	11

Table 2. Example of VLC.

Thus the most seldom occurring symbols are coded with the longest code, effectively minimising the size of the message.

Despite the fact that Huffman has developed his method almost 70 years ago it is still widely used in modern applications, but often as auxiliary compression mechanism. One example is DEFLATE algorithm which will be tested in scope of this work. It is a combination of LZ77 (refer to next section for more details) and Huffman coding.

4.2.3 Lempel-Ziv Compression

Many different compression algorithms were developed during the last century, but one compression family really stands off. Its name is Lempel-Ziv. All of the compression

methods used in this work are part of the LZ-family and in this section we'll take a closer look at the methods it utilizes and the reasons it is dominating data compression for over 30 year.

The basic idea of any LZ-compressor is that it goes over the plain text looking for the sequences of characters that repeat throughout the text and then try to reuse them in the most optimal way [18]. Instead of coding separate characters LZ concentrates on dynamic grouping of adjacent symbols, trying to find the perfect balance between the longest tokens and the smallest *entropy rate*.

Entropy rate of a stochastic process is the average time density of information produced by the process. Shannon's entropy described in Chapter 3 provides a good method for measuring the randomness in a stand-alone or a series of independent events. However events in real systems are often not that independent and historical data often affects the behaviour of the system at present. Consider the following example: the i.i.d. (independent and identically distributed) probability of letter "U" appearing in English text is 2.76%, whereas in practice if the preceding letter was "Q" the probability of "U" following is around 80%. Shannon's entropy cannot be used directly to calculate cases like this, so the equation was modified as follows:

$$H(X) = \lim_{n\to\infty} \frac{H(X_{1,} X_{1,} \dots , X_{n,})}{n}$$

Figure 14. Entropy rate equation.

This means that the entropy rate of a process is the limit of joint entropy of all members of a process *X* divided by the number of members [29]. The very first versions of the algorithm named LZ77 and LZ78 still comprise the very backbone of modern data compression because they addressed entropy rate recursively finding the best combinations of tokens. As a fairly simplified example consider a string "*randomness is so chaotic and random*" to be a compressed subject. LZ will identify that the word "random" at the end of the string has already been encountered 29 characters before. So it will replace the 6 character entry in the word "randomness" with a 2-byte combination of a relative jump and a length of a token: "*[29,6]ness is so chaotic and random*". Going into more detail what LZ actually does it parses the original sequence into distinct phrases or tokens. The further it goes the more distinct tokens it finds, but the

thing is the shortest tokens tend to be found first and they are used as bricks for the longer ones [18].

There are currently more than 20 actively used variations of Lempel-Ziv algorithm, but one of them really stands off. LZMA or Lempel–Ziv–Markov chain algorithm is the only one of them that takes advantage of entropy encoding using Markov-chains. It uses a variable size dictionary of up to 4GB in length for the regular LZ duplicate string elimination, but after that it tries to optimize the achieved result utilizing a Markov-chain-based range coder in conjunction with binary trees [19]. This approach tends to raise the entropy level to theoretical maximum making it the hardest algorithm in terms of distinguishing it from true random noise or encryption.

4.3 Test Data Overview

This section describes the process of experimental test data preparation. To avoid mistakes done by predecessors the author decided to gather a very diverse set of test samples that would differ by the following criteria:

- Size. The files should ideally represent life-like user data so it should include very small, as well as quite large files.
- File type. Since compression algorithms often work differently with different input data it was decided to include as many commonly used file formats as possible.

Original test data is comprised of four main groups with 591 files in total:

- Random operating system files. These include executable binaries, settings files, release notes, PDF manuals, etc.
- User data. This set is comprised of carefully balanced files of different groups like images, text documents (plain text and text-processor created), spreadsheets, and binary modules.
- Images. Special set including only image and 3D data files.
- Plain text only. This set includes only large plain text files.

File sizes lie in range from 53B to 100MB.

All of the files from the above-mentioned sets were processed with different algorithms and either encrypted or compressed. The final selection of algorithms is as follows.

Encryption:

- AES – stands for Advanced Encryption Standard. A modern widely used symmetric-key algorithm developed by Vincent Rijmen. It is adopted by many countries, including US, on a governmental level. It was designed as a replacement for DES algorithm and shares many common features with the latter.

- BLOWFISH – another symmetric key encryption algorithm capable of using a key up to 448 bits. It was designed as a quicker and better alternative to DES algorithm and got a lot of attention in early 1990's. At the moment AES slowly supersedes Blowfish. Interestingly Blowfish is placed in the public domain and can be used by anyone without any licensing.

- CAST5 – another widely used symmetric-key block cipher. It is a default encryption mechanism in a number of software products, namely PGP and GPG. At some point it was considered as a candidate for AES standard, but was later rejected in favour of Rijmen's solution.

- DES – stands for Data Encryption Standard. It is a traditional block cipher algorithm designed in 1970's by IBM in partnership with US NSA. It utilizes keys of 56bit length, which is considered to be insecure taking into account modern computation powers.

- ENIGMA (UNIX Crypt) – a very simple cipher that represents a one-rotor machine designed along the lines of Enigma, but considerably trivialized. This algorithm is used in UNIX crypt(1) utility used for encryption. It is now strongly advised to avoid using it due to the ease of brute force attack.

Compression:

- DEFLATE – one of the most widely-used compression algorithms to date which is utilized in huge amount of software. Uses a combination of LZ77 for duplicate strings elimination and Huffman coding for bit reduction (see section 4.2.2).

- LZSS – stands for Lempel–Ziv–Storer–Szymanski. Another advanced compression algorithm based on LZ77 standard. In original LZ77 the dictionary reference could sometimes be longer than the string to be replaced. LZSS omits any dictionary references if they are found to be inefficient. It is a default compression algorithm for many popular archivers (PKZip, ARJ, WinRAR, ZOO, etc.).

- LZMA – stands for Lempel–Ziv–Markov chain algorithm. This is another young algorithm from Limpel-Ziv family. It is one of the best general-purpose compression algorithms to date [23][7]. It uses a variation of LZ77 algorithm for dictionary encoding followed by a complex model predicting every bit probability based on Markov-chains [19].

In order to raise the entropy level and make archived data as indistinguishable from encrypted or truly random data as possible, compression algorithms were used with their maximum compressing settings turned on (refer to Appendix 2 for complete software and settings listing).

Total number of files used during experiments is 4728 (~20GB).

For the purpose of experiment purity, data in question (encrypted and compressed) was cleaned out of any accompanying interfering data, such as file signatures and headers to the extent it made sense.

Algorithm	**Software**	**Skipped from start**	**Skipped from end**	**Comments**
AES	AESCrypt	192B	0	AESCrypt header is dynamic, but usually less than 160B.
BLOWFISH	mcrypt	32B	0	Standard mcrypt short header containing “BLOWFISH” signature. Skipping.

CAST5	Gpg4Win	0	0	Header is 11 bytes, but they seem to not interfere with results.
DEFLATE	zip	64B	128B	Zip files have both header and footer.
DES	mcrypt	0	0	Standard mcrypt short header is present, but it does not interfere with results.
ENIGMA	mcrypt	32B	0	Standard mcrypt short header containing "ENIGMA" signature. Skipping.
LZMA	7Zip	32B	4%	7Zip entry table is at the back of the file with only signature and offset in the header.
LZSS	WinRAR	5%	0	WinRAR header is of a variable length, usually not going over 1-2% in total size of the package.

Table 3. Tested formats cut-off values.

Since different software was used to generate test data sets, their formats were carefully examined and a list of cut-off values was composed [19][20][21][22]. The resulting values are presented in Table 3. Note that only separate single files were compressed or encrypted meaning that there are no file-entry headers in between the compressed data.

4.4 Tools Selection

For quick and easy execution of different types of randomness tests as well as recursive directory operations the author decided to create a simple dedicated set of tools. The following software products and technologies were selected as technology stack for this research phase:

- Python – as a main programming language, because of its variety of internal tools and speed of development.
- Entro.py – my own library and tool written in Python to perform all the calculations.
- Microsoft Excel – a powerful spreadsheet processor capable of many analytical operations as well as advanced graph visualization.

4.4.1 Entro.py Library and Tool

This library and its accompanying command line tool were designed by the author solely for this thesis work. They are distributed as open-source under MIT license and are available on Github. The library provides the following functionality:

- Shannon entropy calculation.
- Monte Carlo Pi calculation.
- Chi-Square test.
- Arithmetic mean test.
- Working with the whole file or just part of a binary blob.
- Calculation of randomness graph with a given step.
- Export of results to CSV spreadsheets.
- Recursive directory operations on many files.

Thus, this software is capable of conducting most of the needed functionality required at this stage; however there is a plan to continue with the development by adding more

useful features for entropy-related research and practical applications, including more randomness tests, extracted machine learning models for data source analysis (see Chapter 5).

4.5 Methodology

To observe entropy oscillations inside a binary blob an entropy graph is built. This means that data is split into equal parts of a given size (step) and entropy is calculated for every one of them to comprise a continuous flow of values representing internal entropy fluctuations. The selection of a step size was a result of a series of experiments in Chapter 3 during which steps in range of 512B to 16B were observed. Forensic tools on the market such as IBM QRadar Security Intelligence Platform use a step of 512 or 256 bytes to display Shannon entropy graph. In my case however this precision turned out to be not enough as a lot of smallest entropy deviations tend to remain unnoticed. So it was identified that a step of size 32B is the most suitable for this particular task and it was used throughout all of the experiments described in this chapter.

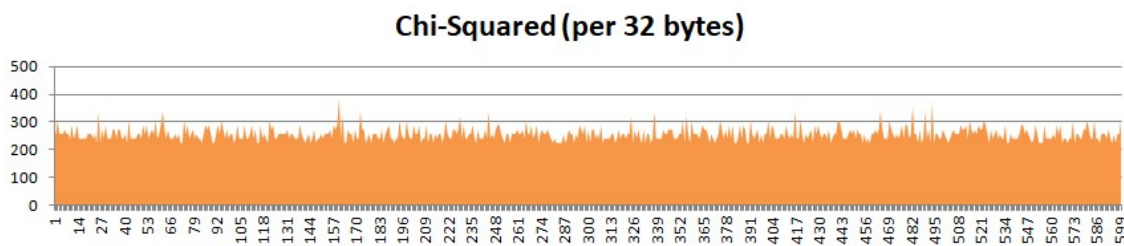


Figure 15. Example chi square graph for a binary blob of 600 x 32B pieces.

To check the assumption that encrypted data may not go over some threshold chi square value, the author decided to check the maximum chi square values for two encrypted sets as a first iteration. The selected sets were CAST5 and AES (Experiment 1 and 2 respectively). As a result it was identified that maximum chi square lies in range from 272 and 560. This maximum threshold value of 560 will be called a “Compression Marker” later in this work. Some interesting observation at that time was also the fact that for smaller files (less than 1MB) it is never above 430 (in this particular data set, see items 370-470 in Figure 15), and is never above 400 for small (less than 500KB) text files (item 325-370 on Figure 15). Comparison of maximum chi square graphs for both algorithms revealed a very similar pattern (see Figure 15).

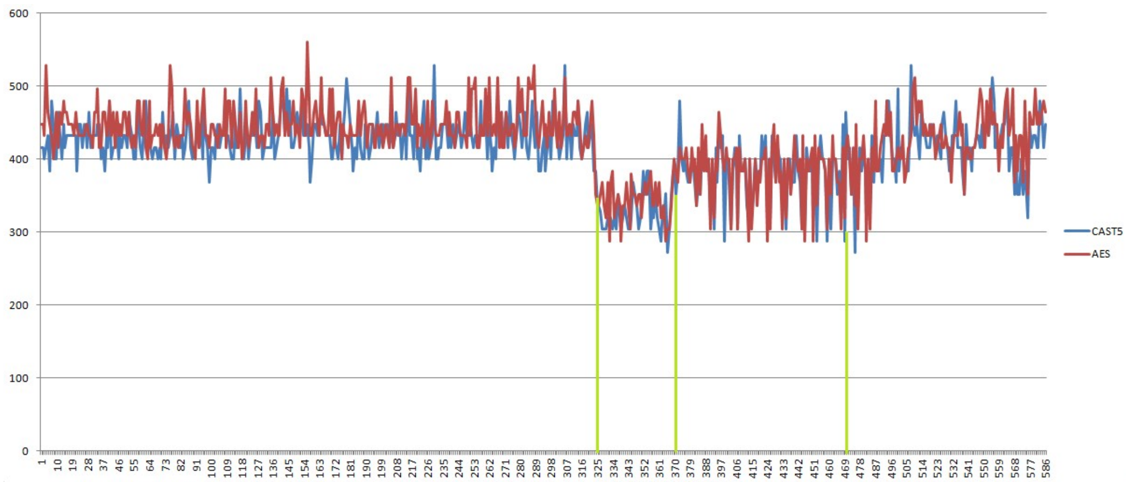


Figure 16. Maximum chi square values for CAST5 and AES file sets.

To eliminate probability of content-entropy relation it was decided to check a set of differently sized encrypted plain text files (Experiment 3). In this case complete chi square graphs were carefully examined for peak values. Resulting data clearly showed that for the large data blobs occasional peaks up to 560 can occur, with normal chi square distribution in range 224 to 400. Thus it became clear that content does not affect entropy of encrypted data, but large encrypted data is more likely to have occasional oscillations.

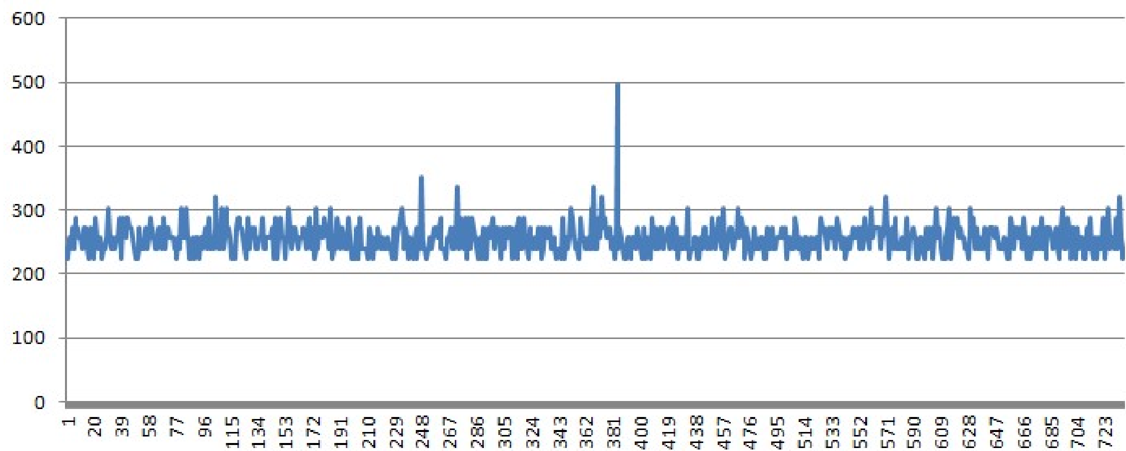


Figure 17. Occasional oscillation in encrypted data entropy.

After previous test data collection it was decided that the examined feature list should be extended to include the following:

- Minimum chi square value.
- Maximum chi square value.

- Average chi square value.
- Relative location inside the file and intensity of peak values.
- Total number of peak values.
- Data size.

Next iteration included entropy features collection for two of the archiving algorithms, namely LZMA and LZSS (Experiments 4 and 5), as they provide the best compression available and should be the closest ones to encryption or true randomness in terms of entropy distribution. Note that test samples were cleared of any header-footer data to provide clear results, as described in Chapter 4.1. The aim of these tests was to identify if the chi square values above the previously identified threshold of 560 appear commonly in compressed data and if yes is there any pattern.

The number of outstanding chi square peaks in LZSS data turned out to be pretty high on average and gave some confidence in using this feature for distinguishing at least this type of compression from encryption (see Figure 17). However results for LZMA algorithm were not that obvious and convincing. LZMA provides a much better compression level [7], hence closer entropy to pure random data.

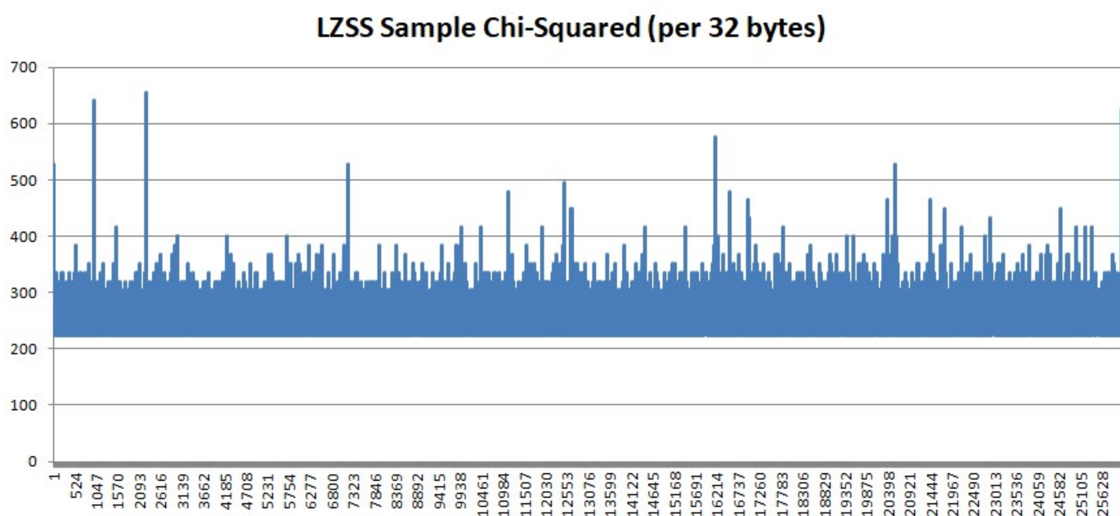


Figure 18. LZSS sample chi square distribution.

More than half of the samples did not have any Compression Markers, yet again another half showed some occasional oscillations far above 560 randomly appearing in the middle of uniformly distributed entropy (see Figure 18).

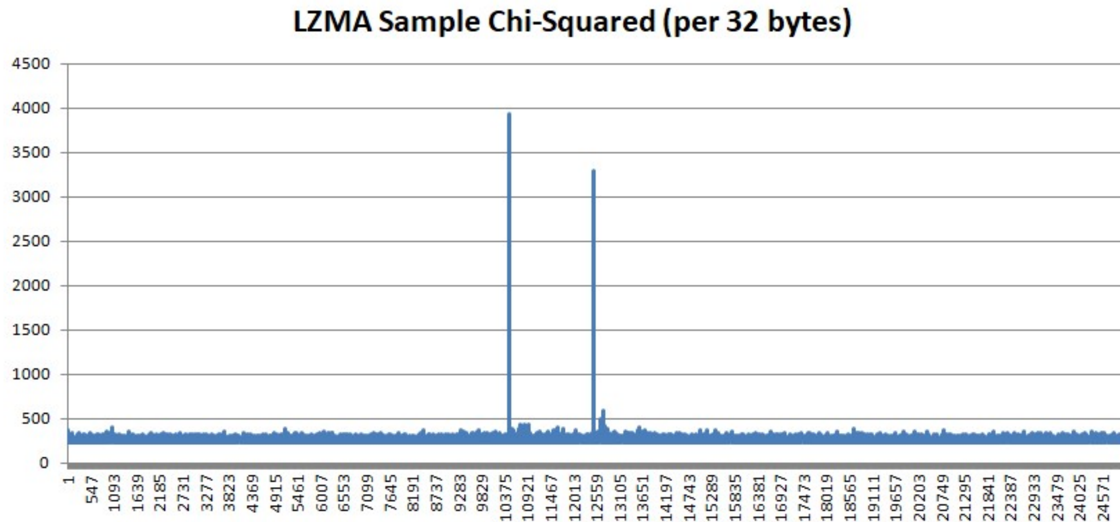


Figure 19. LZMA sample chi square distribution.

As a conclusive test (Experiment 6) for this chapter the above-described procedure was run on the full test set of 4728 files and the outcomes were carefully evaluated. Results of first two experiments were extended with missing data.

4.6 Results

Analysis of the data collected in Experiment 6 revealed some patterns as well as some expected abnormalities. Encrypted file sets were tested to fall into previously identified chi-square-range of 224 to 560:

- DES-encrypted samples had two files with single anomalies of chi square 576 and 608 randomly in the middle of the data.
- BLOWFISH-encrypted set had one file with a spike of 592.
- CAST5, ENIGMA, and AES samples' chi square values all lied in the expected range.

For compression sets the total count of Compression Markers and any additional features that could help to identify compression were looked for:

- DEFLATE-compressed set had 115 items (19.46% of total count) with no Compression Markers; 92 of them were less than 400KB in size.
- LZSS set had 72 items (12.18% of total count) with no Compression Markers.
- LZMA set had 330 items (55.84% of total count) of different sizes were missing Compression Markers.

Preliminary results identified no single-dependency on any of the selected features; however there is clearly a correlation between the size of the data and the number of compression markers in the compressed data and the results were very stable for encrypted data sets.

The initial assumption was that there is probably a linear statistical dependency between the compressed data size and the number of compression markers appearing in it. So the idea was to create a so-called confidence graph where the precision of data content prediction would rise along the data size axis as shown on the Figure 19.

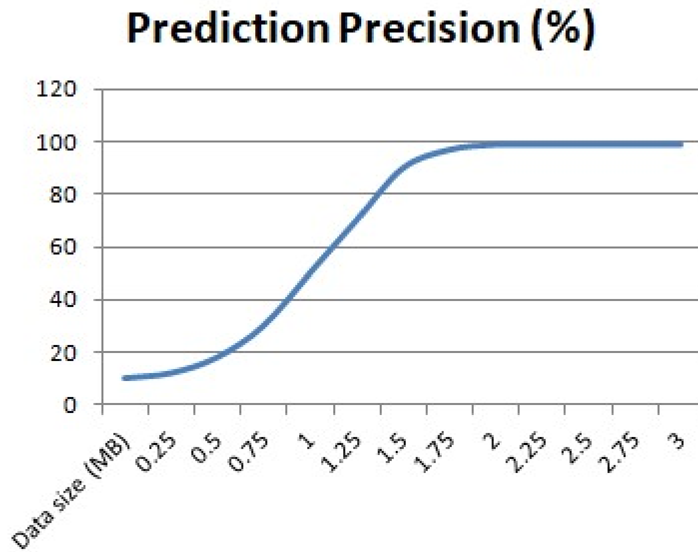


Figure 20. Confidence Graph. Initial idea (dropped out).

However, after careful analysis of the collected data it became obvious that manual recognition of features and their mutual interconnections that affect the decision will take enormous time and effort, considering that the number of affecting features was more than two. Thus some sort of machine processing must be introduced.

4.7 Conclusion

In this chapter a deep case study on the underlying principles of encrypted and compressed data formation was conducted and some popular algorithms in both domains were covered. A series of experiments was then run on a severely large dataset in an attempt to identify patterns that could theoretically help in distinguishing compressed and encrypted data. Although some patterns were found their nature and exact measurements were hard to extract due to the size of the result set and the absence of an obvious linear feature correlation. This chapter concludes major sample data collection and precedes the upcoming statistical analysis stage with machine learning approach.

5 Machine Learning Model Training

Now that a severely large collection of resulting data was collected a method to effectively process it and conclude the relationships needed to be selected. Modern frameworks for machine learning (ML) provide algorithms with this functionality: provided some data with known outcomes ("training data") they are capable of building a mathematical model that can predict the outcomes of new data with certain amount of precision [8].

5.1 Tools Selection

Since the previously selected programming language was Python it was natural to continue with it. Another deciding fact was that Python is a de-facto industry standard for machine learning and data science, because of its speed of development and a wide range of tools for almost any applied science area imaginable. According to Igor Bobriakov's research [9], SciPy (and its subsidiary Scikit-learn), Pandas, and NumPy are the top data science libraries based on the functionality provided, development activity, size of supporting community, and some other metrics.

5.1.1 NumPy

NumPy stands for Numerical Python and is a collection of useful classes and functions for convenient operations on n-arrays and matrices, linear algebra transformations, and random number manipulations. It is considered a fundamental library for scientific computing [10]. Its code is optimized for high performance scientific computing and data analysis.

5.1.2 Pandas

Pandas is a very powerful Python package, which provides easy functionality for structured data manipulation, aggregation and visualization. chi square features from Chapter 4 experiments were stored as CSV spreadsheets and those can be conveniently

fed to Pandas for later manipulations, such as columns addition-deletion, data grouping, etc.

5.1.3 Scikit-Learn

Scikit-Learn is an additional package of SciPy-family with the aim of providing easy-to-use, yet very powerful tools for machine learning and data science. The library covers most of the main ML algorithms for classification, regression, and clustering. Since it is production-ready library it was built with performance in mind, utilizing C and C++ programming languages in efficiency-critical areas.

5.2 Methodology

Machine learning is a very broad area of algorithms and mathematical models. Selecting the right algorithm requires some background knowledge on the topic and careful analysis of the data as well as the desired outcome.

There are two main areas in machine learning: the supervised and unsupervised ML. The main difference in them is that for supervised ML the data should be properly labelled in advance before the training starts. That means that the target feature is defined and set in the training set and the algorithm knowing the correct answer from the beginning iteratively tries to build a mathematical model that predicts any future inputs. Unsupervised models are fed unlabelled raw data. They are then left with no- or limited supervision to discover the underlying structure of the data.

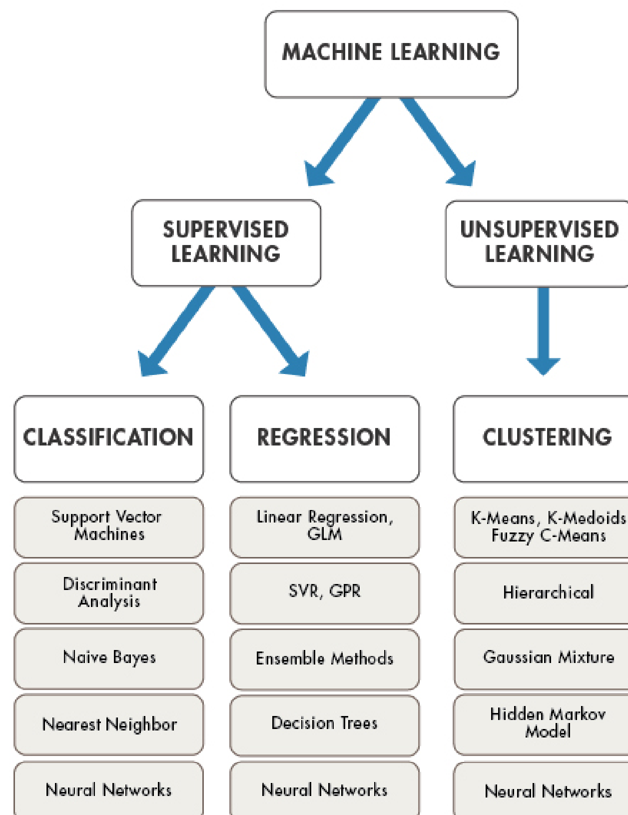


Figure 21. Classification of ML tasks and algorithms.

Supervised ML is in turns divided into two main areas: classification and regression. Classification is a type of a problem when an entity in question needs to be categorized into a finite number of classes (e.g. “red” / “green” / “blue”, “trustful customer” / “untrustful customer”, or “true” / “false”). Regression on the other hand outputs a real

number in some finite range. Most of the supervised ML algorithms can be used for both classification and regression.

5.2.1 Data Selection and Preparation

After experiments performed in Chapter 4 two resulting data sets very different by nature were collected:

- Chi-square graphs for every file in the test set. This data was stored as binary dumped NumPy arrays of chi square values. This data falls under the category of unstructured labelled data. Unstructured data like images, text, audio streams, etc. is impossible to process as is using traditional methods due to its non-uniform structure and ambiguities caused by it. Usually the pre-processing stage requires some special steps (e.g. feature extraction, clustering, etc.) before actually training a model.

- Extracted chi square features for every file in the test set. This is a more traditional structured data model stored in regular CSV spreadsheets (see Table 4). This type of data is most of the time ready for machine processing with minimal preparations.

Since a structured and carefully labelled data was in possession, it was decided to first experiment with it and only in case of failure to try the unstructured data set and a deep learning approach, which would require significantly more time and effort.

My structured results data set was labelled so supervised learning was an obvious path. The number of outputs is strictly limited to 2 values: encryption or compression. That means this task is of a Classification type. There are plenty of classification algorithms available and in general there are no strict rules in regards to what algorithm behaves better under what circumstances and it is often upon the researcher to identify it empirically.

File Name	Max Chi	Avrg. Chi	Min Chi	Peak Count	Peak IDs	Peak Positions	Block Count	Data Size	Encryp-tion
testset_002.pdf.rar	1488	260.2897	224	3	2299-2553-3655	62-69-99	3659	117088	0

testset_003.pdf.rar	560	258.8228	224	0			7233	231456	0
testset_005.pdf.rar	832	258.846	224	1	6882	99	6886	220352	0
testset_010.pdf.rar	848	256.5971	224	5	23052-23053	14-14	160998	5151936	0
testset_011.jpg.rar	544	255.8226	224	0			6138	196416	0
testset_012.jpg.rar	624	256.2706	224	1	28711	99	28715	918880	0
testset_013.jpg.rar	480	255.4601	224	0			4684	149888	0

Table 4. Example of entropy features data spreadsheet.

There are standardized ways of how to prepare the data for correct model training and validation. All results of Experiment 6 from Chapter 4 were carefully reshuffled to create an even distribution of items in the set. They were then split 70/30, with 70% of results used for training a model (training data) and 30% (so called "holdout") for later model validation. A method called stratified k-fold cross-validation was applied to the training data. The idea of this method is to split the data set into k chunks (folds) and iteratively treat every next chunk as validation set and all the remaining (k – 1) chunks as training data. In case the model is good, results of all iterations should be more or less close to each other. If the results differ greatly there is a big chance of data being unevenly shuffled or corrupted. Mean value of those k-iteration results can be considered a model prediction precision. For this exact case the author decided to go with a common value of k=5.

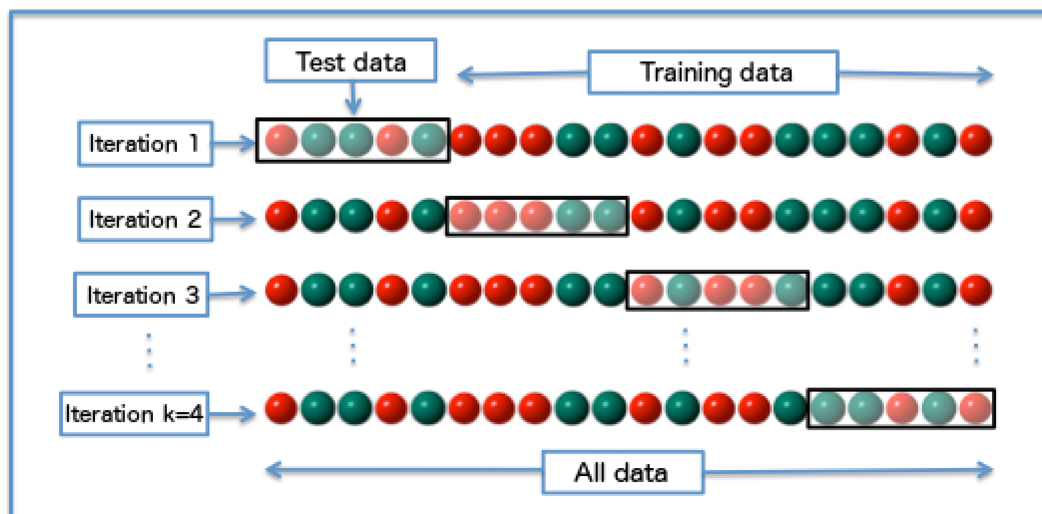


Figure 22. k-fold cross-validation with k = 4.

The following features were selected to be used in training:

- Max Chi-Squared value.

- Average Chi-Squared value.
- Min Chi-Squared value.
- Total count of compression markers.
- Total block count (i.e. file size = block count * 32B).
- Encrypted (True/False).

At least for now relative positions of compression markers were dropped out of the training data as they seem to be quite random and should not affect the decision.

5.2.2 Used ML Models Overview

There are numerous classification models available in Scikit-Learn package. Five of them were selected for this task, going from the very simplest to some more complex ones.

Decision Tree – one of the simplest ML models. It is represented by a binary tree with "branches" representing target function attributes and "leafs" holding target function values. All other nodes hold attributes that define different cases. To get a classification prediction the input value should be passed from the root through decision nodes until a leaf is reached and that will be the prediction value.

KNN – stands for k-nearest neighbours. When used for classification this algorithm assigns a class to the element basing on the most prevalent among *k* neighbours of that element which classes are already identified.

Random Forest – this is an ensemble learning method, which operates by constructing a multitude of decision trees. In case of classification it outputs the class that is a mode of the individual trees.

AdaBoost – stands for Adaptive Boosting. This algorithm amplifies weak classifiers by combining them into a weighted sum (boosted classifier).

Logistic Regression – this is a statistics based model that tries to predict the probability of event by fitting it to the logistic regression curve.

Since this is still a work in progress some other classification models being evaluated and probably will provide better results in the future.

5.2.3 Preliminary Experiments

As a first iteration Decision Tree and KNN classifier models were picked. These simplest models were selected to see if the approach is at all viable and it makes sense to continue work in this direction. No additional fine-tuning was done on them and they were used with their default settings left intact. Surprisingly even the very first results for Decision Tree and KNN gave 88.8% and 82.7% of success respectively. The mentioned success rate is for cross-validation; no holdout-validation was done on that stage.

5.2.4 Other Models and Model Fine Tuning

With pretty remarkable results of the first iteration some fine-tuning of the models was performed. A standard way of doing this is by creating special matrix of permuted settings (model hyper-parameters) for the model and iteratively observing which set outputs best results. This approach is called Grid Search and performed with a GridSearchCV tool from Scikit-Learn.

- Grid Search identified optimal hyper-parameters which are:
- Decision Tree – maximum depth 4.
- KNN – N-neighbors 1.

With these parameters set precision of tree rose up to 91.4% and for KNN to 85.6%.

A series of experiments was run using other models and the prediction precision rose even higher. Full table of results and interpretation can be found in Chapter 5.3.

5.3 Decision Tree Analysis and Results

In this section an overview of ML models performance is given as well as the attempt to interpret the gathered results. Interpreting ML models is usually a tough challenge due to the amount of interconnecting relations and minimal decision conditions. However the author believes that the provided analysis is quite thorough and gives the idea of underlying principles.

Below is the full results table collected during evaluation of different models. As seen from Table 5, Decision Tree, Random Forest, and AdaBoost showed the most impressive results. Most of the models performed slightly better on the real data than on cross-validation. At this stage it was decided that the achieved precision is enough for a prototype, but it is definitely a question for future research if any other model can give even better prediction.

Model Name	**Cross-Validation Prediction Precision**	**Holdout Validation Prediction Precision**	**Model Settings**
Decision Tree	0.91419	0.91655	max_depth = 4
KNN	0.85597	0.86704	n_neighbors = 1
Random Forest	0.92086	**0.92716**	max_depth = 8 n_estimators = 15
AdaBoost	0.92177	0.92150	n_estimators = 46
Logistic Regression	0.89176	0.89604	default

Table 5. Prediction precision by model.

Despite the good performance of the approach the question “Why does it behave like this?” remained. Most machine learning models are quite hard to visualize, i.e. it is not easy to say why a certain algorithm made a specific decision for a given input. The only one of them which stands out is Decision Tree (and Random Forest as it is a derivate of the former). This is why it was decided to visualize it in order to understand what features and values were used for prediction making.

As seen from the Figure 22, the primary divisive feature is Max Chi. Initial set consists of 3298 items, of which 1253 are compressed and 2045 are encrypted. The first split is based on the condition of Max Chi being lower or equal to 520 and it breaks the selection into parts of 2353 and 945 samples.

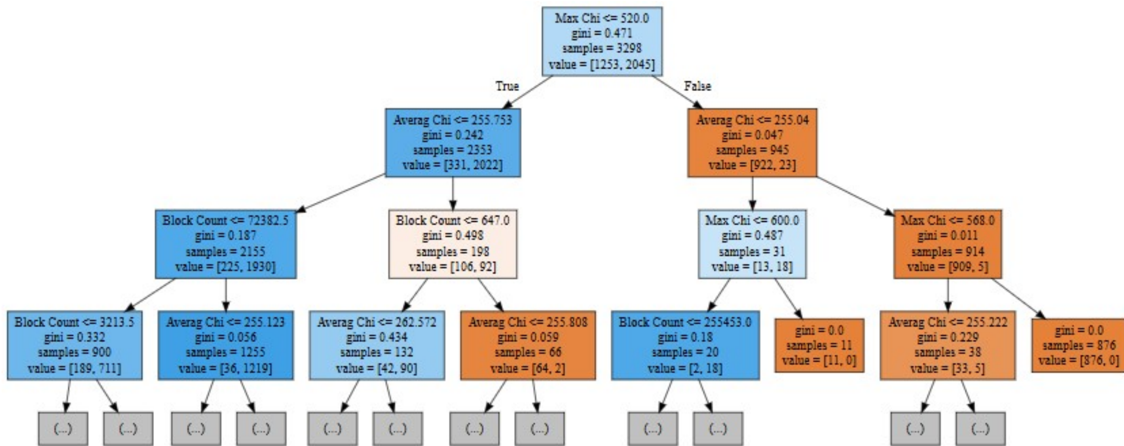


Figure23. Trained Decision Tree structure (cropped).

Lower level splits continue breaking the selection by Average Chi, Maximum Chi, and Block Count (i.e. data size). That means that only these three features actually do affect the decision making. The colour codes on the graph mean the following: the more blue the box is the more encrypted items in the sub-group, the more it is orange the more compressed items the given sub-group contains.

Apparently small differences in the average chi square value play a big role alongside obviously affecting data size and maximum chi square value. Nature of this relation is not very clear since the fragmentation of the decision most of the time tends to be in range (255, 256) and a deeper research of this pattern is needed. The maximum chi square parameter and the data size were expected to be the “key players” in decision making.

5.4 Conclusion

In this chapter a case study on machine learning techniques was conducted, followed by a series of experiments, which resulted in an effective machine learning method of encrypted data identification. The mentioned method was tested on a severely large data set and allows for a pretty high precision prediction with some limitations. Analysis of the trained ML model showed some interesting patterns inside highly compressed and encrypted data that helped with the classification. However it is still a matter of future research to recognize the true nature of these patterns.

6 Sleuth Kit Autopsy Module Development

The results of Chapter 5 needed some practical application. There are numerous areas where the method could be applied, but it was decided to implement an experimental plug-in for Sleuth Kit Autopsy platform which purpose would be to identify encrypted files. The platform selection is based on the following reasons:

- It is widely used in the digital forensics society.
- It is open source with a wide contribution community.
- Autopsy provides two extensibility API's for Python and Java programming languages.
- The eco-system of Autopsy makes the origin of file data transparent. This means that it is up to platform to provide the data whether it is originating from a local file system, a disk image, extracted from an archive, or was carved from slack space.
- Autopsy provides convenient rich UI capabilities like artefacts blackboard and report engine.
- Sleuth Kit Autopsy is distributed with a pretty wide range of different analytics modules, with many more available for installation from third parties. The results of these modules can be used inside your own plug-in, which allows you to aggregate data from them and build additional functionality on top of the other.

This chapter describes development process and testing of the Autopsy ingest module using results of previous chapters.

6.1 Autopsy Python Eco-System

Since all implementation in previous chapters was done using Python and Autopsy providing API for that, it was obvious to continue with this programming language.

However some not very convenient facts were discovered during the course of this work.

Autopsy is written in Java and it uses Jython, which is a Java-based python interpreter that compiles python-like source code to Java bytecode. The latest Jython-supported version of Python is 2.7, which imposes some restrictions on the modern Python code used in previous chapters (in fact some of the constructs required restructuring to comply with both 2.7 and 3+ versions of Python). Another unpleasant fact about Jython is that it is currently not very well supported with the latest release been done in May 2015. This made it almost hopeless to wait for some bugs found by the author during the development process to be fixed, which led to some workarounds to bypass this “bottleneck”.

Since Jython is not a real Python environment it has a limitation of not being able to run native components written in C, thus Scikit-Learn dependency of the code developed in Chapter 5 was not an option. However the author found a way to bypass this limitation with minimal impact to logic and prediction precision (see Chapter 6.4).

6.2 Existing Encryption Detection Module Analysis

There is an existing encryption detection module in Autopsy. Its behaviour and source code were carefully analysed with different types of data including corner cases identified in previous chapters. The results clearly showed that this module is very basic and does not provide flexibility and robustness needed for this task. Some of the weaknesses found during examination include:

- Compressed files with no headers were all identified as encrypted due to high Shannon value (false positives).
- All encrypted files smaller than 1MB were omitted from results due to unreasonable minimum file size requirement (false negatives).
- Encrypted files that had their original file extensions preserved simulating "ransomwared" data (false negatives).

After going through the source code it became clear that internal Autopsy module relies on the MIME type acquired from file extension and it only parses files that are set to have MIME type of "application/octet-stream".

```
/*
 * Qualify the MIME type.
 */
String mimeType = fileTypeDetector.getMIMEType(file);
if (mimeType.equals("application/octet-stream") && isFileEncryptionSuspected(file)) {
    return flagFile(file, BlackboardArtifact.ARTIFACT_TYPE.TSK_ENCRYPTION_SUSPECTED,
            String.format(Bundle.EncryptionDetectionFileIngestModule_artifactComment_suspected(), calculatedEntropy));
} else if (isFilePasswordProtected(file)) {
    return flagFile(
        file,
        BlackboardArtifact.ARTIFACT_TYPE.TSK_ENCRYPTION_DETECTED,
        Bundle.EncryptionDetectionFileIngestModule_artifactComment_password());
}
```

Figure 24. Autopsy Encryption Detection module decision logic.

The encryption detection function then simply calculates Shannon entropy value for the whole data source and if the result is larger than the minimum set (default is 7.5) decides that it is encrypted. This in no way can be treated as trustful as the previous chapters clearly showed.

6.3 Ingest Module Functional Specification

Autopsy ingest module is a type of a plug-in that analyzes the data in the data source, regardless of its origin. This can be a single logical file or folder, a local storage device, or a disc image (i.e. byte-for-byte copy of a hard drive or other storage media). These modules perform the entire analysis and data source parsing routines. As soon as a new data source is added the user is presented with a dialog to enable and configure ingest modules. When configuration is done selected ingest modules immediately start parsing and providing the findings in real time.

My idea was to develop a simple ingest module with the aim on proving the previously identified method in real life situations. The list of required features included:

- Identification of high entropy files.
- Making assumption whether the file is encrypted.
- Making assumption whether the file is highly compressed.
- Displaying Shannon entropy, chi square value, Monte Carlo Pi, and other randomness stats for suspicious files.

It was decided not to develop any specific GUI elements for this realisation of plug-in, since Python modules are not very suitable for user elements development (however it is definitely a target for second version). Instead the standard blackboard controls from Autopsy were used for displaying of identified files and their statistics.

6.4 Implementation Details

The Autopsy module is developed and distributed as part of Entro.py library (see Chapter 4.2.1), since it utilizes most of its core features.

Since it was not a possibility to utilize Scikit-Learn library and any pre-trained models from Chapter 5, due to Jython limitations, it was decided to extract prediction rules from pre-trained Decision Tree. Its prediction precision was a bit less than of the Random Forest, yet acceptably close (0.91655 and 0.92716 respectively). The resulting code is a series of nested IF-statements, representing the decision making logic (see Figure 24).

```
def is_encrypted(max_chi, average_chi, block_count):
    if max_chi <= 520.0:
        if average_chi <= 255.75341796875:
            return 1
        else:   # if average_chi > 255.75341796875
            if block_count <= 647.0:
                if average_chi <= 262.5716857910156:
                    return 1
                else:   # if average_chi > 262.5716857910156
                    return 0
            else:   # if block_count > 647.0
                if average_chi <= 255.80758666992188:
                    return 1
                else:   # if average_chi > 255.80758666992188
                    return 0
    else:   # if max_chi > 520.0
        if average_chi <= 255.04031372070312:
            if max_chi <= 600.0:
                return 1
            else:   # if max_chi > 600.0
                return 0
        else:   # if average_chi > 255.04031372070312
            if max_chi <= 568.0:
                if average_chi <= 255.22213745117188:
                    return 1
                else:   # if average_chi > 255.22213745117188
                    return 0
            else:   # if max_chi > 568.0
                return 0
```

Figure 25. Decision Tree extracted as Python code.

It is possible to extract Random Forest in the same manner, but since it did not make a significant difference in precision it was decided to continue with Decision Tree.

Module's main logic flow is as follows: whole file's entropy is computed and if it exceeds 7.9 the file is checked for encryption and moved to the "Potentially Encrypted" results, if positive, or to "Potentially highly compressed" otherwise (see Figure 25).

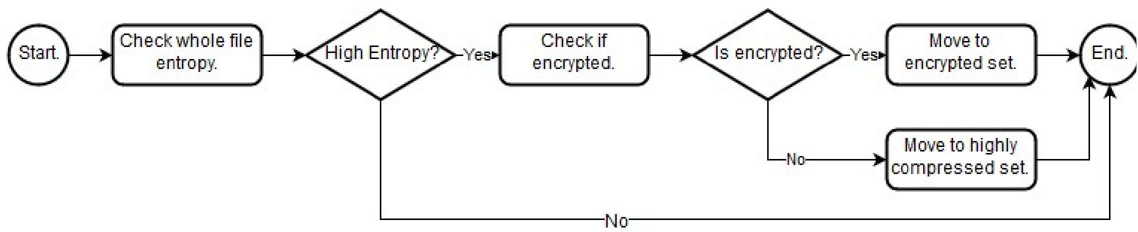


Figure 26. Autopsy module logic flowchart.

Autopsy analyses different types of data so for this module anything that is not a file or is unused or unallocated block set is filtered out, to avoid confusion. However it is planned for future versions to have functionality that is able to analyze arbitrary binary blobs.

6.5 Results

This section covers two severely large tests conducted to identify limitations of current solution.

6.5.1 Synthetic Test

As a first test for the created ingest module it was decided to conduct a special synthetic test in an attempt to identify any edge cases that may occur. To achieve this goal a very diverse mix of real user data with highly compressed header-less and encrypted files was comprised.

A set of real business data consisting of 1502 documents was selected. File set is mostly comprised of MS Word documents, MS Excel spreadsheets, PDF documents, packed archives (ZIP and RAR), and digitally signed packages (BDOC and ASICE). To simulate ransomware attack some of the files were encrypted with CAST5 algorithm (405 items), but their filenames and extensions were left intact. To make the test more advanced another subset of files (100 items) was compressed with xz tool with "raw" and "maximum compression" options set, i.e. the files were turned into highly compressed binary streams with no identifying information, such as headers, footers, or signatures. Another subset consisting of 110 items was compressed using libzip (again raw compressed stream and maximum compression were used).

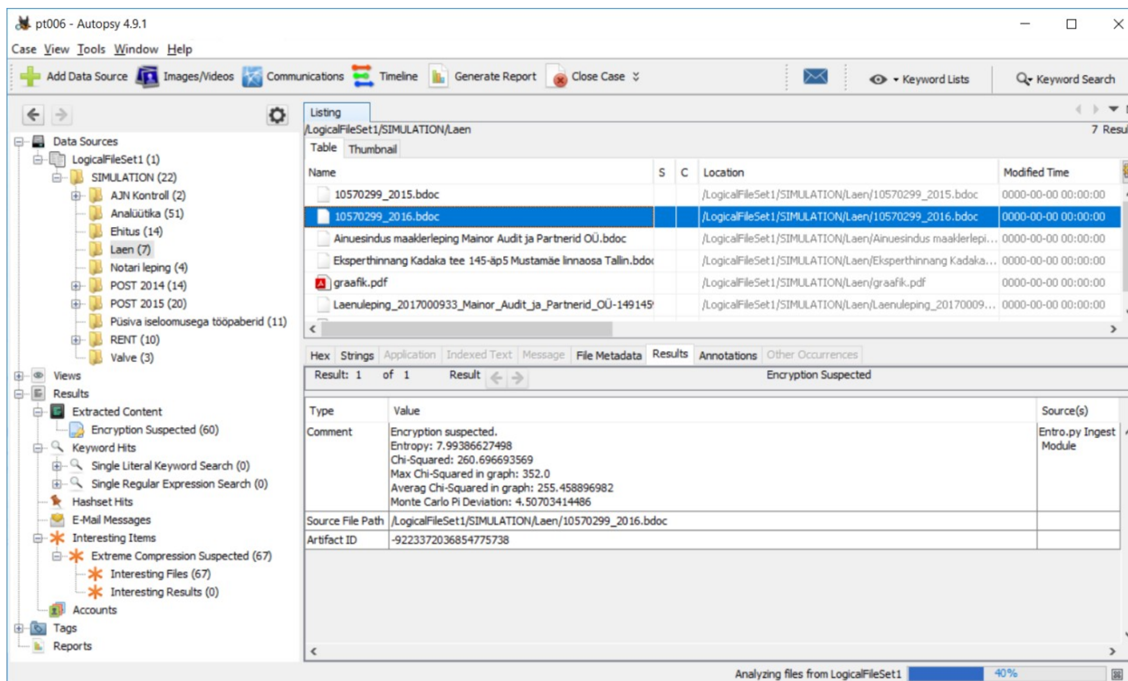


Figure 27. Entro.py ingest module results in Autopsy blackboard.

A full scale simulation was then run on the whole file set. As a result every single encrypted file was identified as such. Vast majority of compressed files were successfully distinguished from encrypted ones. Note that hardest conditions were used for this test since no signature data was stored. Compressed items that were not qualified as compression automatically ended up as encryption suspects (false positives).

Test Parameter Name	Result
Number of encrypted files identified	405 (100%)
Number of LZMA compressed files identified	58 (58%)
Number of GZIP compressed files identified	116 (97%)
Encryption detection false positive count	46
Encryption detection false negative count	0

Table 6. Encryption and compression detection results.

Thorough analysis of false positive results for Entro.py module showed that 73% of them were relatively small files (less than 1MB), which was expected from experiments in Chapter 4.

Some of the user data, namely PDF documents and ASICE/BDOC containers, that was neither compressed, nor encrypted also ended up in high compression suspects result set, but the author does not consider it a false positive, as those file formats utilize

different compression algorithms. BDOC's are essentially a ZIP-containers taking advantage of DEFLATE algorithm, whereas PDF employs the whole series of different compressions for different data types (LZW, DEFLATE, JPEG, RLE, etc.).

Plug-in module works very reliably and showed some impressive results which gave some additional proof to experiment results in previous chapters.

6.5.2 Realistic Test

For the more lifelike test special software replicating the behaviour of typical ransomware was created. The developed application "attacks" a user system and performs the following actions:

- Identifies user files with DOC, DOCX, XLS, XLSX, JPEG, and TXT extensions.
- Original files are then encrypted with Gpg4Win (CAST5 algorithm) and file headers are cut down. The encrypted versions are put instead of the originals with the filenames left intact.
- Statistics on what exact files were encrypted is collected for the verification purposes.

A test Windows 10 Home user system with a set of real user documents and images was then infected with this test virus. Disk image of the system state was taken for preparation in Autopsy. Since system files were out of interest in this situation ingest module was appointed to only parse "C:\Users" directory.

	Total Infected Count	**Identified as Encrypted**	**Comments**
Images	12539	12452	87 clipart and really small images (less than 2KB) were missed due to small size. They were identified as extreme compression suspects.
MS Word Document	75	71	3 temporary MS Word files were missed due to small size. They were

			identified as extreme compression suspects. 1 Empty (0 byte) document was not even encrypted.
MS Excel Spreadsheets	24	24	
Text Files	881	776	A total of 105 files were not identified. These files (in encrypted state) were all less than 1KB in size. None of them represented real user data, but some system or software files instead (logs, CMakeLists, README's, etc.).

Table 7. Realistic test results.

The results of the realistic test seemed not very satisfying at first with around 200 files missed, but after thorough analysis it turned out that the missed items did not represent valuable user data, but system/software helper files and/or temporary data. With that in mind the test can be accounted as successful; however the issue with small files identification must be addressed in the future. Another action point for upcoming improvements is the algorithm used: for the sake of quick validation Decision Tree model was used, but as seen in Chapter 5, this was not the top performing algorithm.

6.6 Conclusion

In this chapter a robust digital forensics tool was developed utilizing the findings and knowledge of previous experiments. The machine learning model trained in Chapter 5 was successfully converted to a source code structure to be used as a stand-alone programming module. The development process was followed by two comprehensive tests targeted at corner case situations detection and lifelike environment behaviour simulation. Both tests showed that the tool's precision is very high and it gracefully handles most edge cases. At this point a full scale field test is needed and there's a plan to work in that regard in cooperation with Sleuth Kit Autopsy community and Estonian Forensic Science Institute.

7 Summary and Conclusions

In this chapter a brief summary of conducted work as well as the results and future proposals are given.

7.1 Conclusion and Future Work

In the very beginning of this work it was intended to come up with a robust and flexible solution for effective distinguishing of encrypted data. During the course of the work hardest conditions for the algorithm were identified: extreme levels of compression tend to have very similar patterns as encryption. Deep subject exploration as well as significant experimental mass helped in identification of a method capable of achieving the desired functionality by means of combining randomness tests result data and machine learning techniques. As a result of these efforts a theoretical model and a set of practically usable tools were developed. Concluding validation tests showed big potential in the proposed solution: method works well in very hard conditions with an acceptable success rate and has a lot of points to improve on in the future.

The practical outcome of this work is a long awaited comprehensive tool for work with entropy targeted at open source digital forensics community. It is still in its prototype stage; however there are currently no analogues with the same functionality available. Still there are some limitations and improvement points to work on:

- Low precision on extremely compressed small data.
- Non-binary encryption (e.g. ASCII PGP) is not accounted.
- Overall performance is pretty poor and for now can only be applied to offline detection.

With that said a range of improvements and future activities was specified:

- During the experiments in Chapter 4 a severely big amount of unstructured entropy data was collected. This can be used in an attempt to train an unsupervised ML model such as CNN. This potentially could improve the work with small data samples.

- ML classifiers can be replaced with regressors to output the chances of some binary blob being encrypted instead of outputting a binary decision.

- Probably it is a good idea to try and combine the results of different randomness tests as proposed in [2], but calculate histograms instead of a single value. This could also potentially increase the precision of prediction.

- The Autopsy plug-in should be migrated to Java, to benefit from rich GUI capabilities provided by Autopsy Java API.

- The proposed method needs a better field testing. For that there is a plan to provide the module to the open source society and some governmental organizations.

This project was started out as a basic entropy research, but ended up as a thorough exploration of the subject with some interesting practical discoveries. The author sees a great potential in the ML approach proposed, but most probably a far larger sample set needs to be comprised to improve the success rate.

References

[1] C. Shannon, “A Mathematical Theory of Communication”, 1948.

[2] S. Cha, H. Kim, “Detecting encrypted traffic: a machine learning approach”, Department of Software, Sungkyunkwan University, Republic of Korea, 2017.

[3] T. Brosch, M. Morgenstern, “Runtime Packers: The Hidden Problem,” Proc. Black Hat USA, Black Hat, 2006 [Online] Available: www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf [Accessed: 07.12.2018].

[4] R. Lyda, J. Hamrock, “Using entropy analysis to find encrypted and packed malware”, IEEE Security & Privacy, Volume: 5, Issue: 2, March-April 2007, pp. 40-45, 2007.

[5] C. Heffner, “Differentiate Encryption From Compression Using Math” [Online] Available: http://www.devttys0.com/2013/06/differentiate-encryption-from-compression-using-math/ [Accessed: 07.01.2018].

[6] C. Heffner, “Encryption vs Compression, Part 2” [Online] Available: http://www.devttys0.com/2013/06/encryption-vs-compression-part-2/ [Accessed: 07.01.2018].

[7] L. Collin, “A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA” [Online] Available: https://tukaani.org/lzma/benchmarks.html [Accessed: 07.01.2018].

[8] C. M. Bishop, “Pattern Recognition and Machine Learning”, Springer, 2006.

[9] I. Bobriakov, “Top 15 Python Libraries for Data Science in 2017” [Online] Available: https://medium.com/activewizards-machine-learning-company/top-15-python-libraries-for-data-science-in-in-2017-ab61b4f9b4a7 [Accessed: 07.01.2018].

[10] W. McKinney, “Python for Data Analysis, 2nd Edition, Chapter 4. NumPy Basics: Arrays and Vectorized Computation”, O'Reilly Media, 2017.

[11] D. Huffman, “A Method for the Construction of Minimum-Redundancy Codes”, 1952.

[12] P. Elias, “Universal codeword sets and representations of the integers.", IEEE Transactions on Information Theory, vol. 21, no. 2, pp. 194-203, 1975.

[13] A. Haecky, C. McAnlis, “Understanding Compression”, O’Reily Media, 2016.

[14] D. Salomon, G. Motta, “Handbook of Data Compression, 5th Edition”, 2010.

[15] G. Stix, “Profile: David A. Huffman”, Scientific American issue September 1991, pp. 54-58, 1991.

[16] National Institute of Standard and Technology, “Advanced Encryption Standard (AES) Specification”, 2001.

[17] P. Rogaway, “Nonce-Based Symmetric Encryption”, 2003.

[18] P. Shor, “Lempel-Ziv notes” [Online] Available: http://www-math.mit.edu/~djk/18.310/Lecture-Notes/LZ-worst-case.pdf [Accessed: 07.01.2018].

[19] R. Gordon, “Understanding 7z Compression File Format” [Online] Available: http://www.romvault.com/Understanding7z.pdf [Accessed: 07.01.2018].

[20] ForensicsWiki, “RAR version 4.11 - Technical information” [Online] Available: http://www.forensicswiki.org/w/images/5/5b/RARFileStructure.txt [Accessed: 07.01.2018].

[21] PKWARE Inc., “APPNOTE.TXT – .ZIP File Format Specification, version 6.3.4” [Online] Available: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT [Accessed: 07.01.2018].

[22] Packetizer, Inc., “AES File Format” [Online] Available: https://www.aescrypt.com/aes_file_format.html [Accessed: 07.01.2018].

[23] T. Klausmann, “Gzip, Bzip2 and Lzma compared” [Online] Available: https://web.archive.org/web/20130106193958/http://blog.i-no.de/archives/2008/05/08/index.html [Accessed: 07.01.2018].

[24] P. Dorfinger, G. Panholzer, W. John, “Entropy Estimation for Real-time Encrypted Traffic Identification”, In Proceedings of the 3rd International Conference on Traffic Monitoring and Analysis, 2011.

[25] A. M. White, S. Krishnan, M. Bailey, F. Monrose, and P. Porras, “Clear and Present Data: Opaque Traffic and its Security Implications for the Future”, In Proceedinds of the Network and Distributed Systems Security Symposium, The Internet Society, 2013.

[26] H. Zhang, C. Papadopoulos, D. Massey, “Detecting encrypted botnet traffic”, In Conference on Computer Communications Workshops, 2013.

[27] S. Kenny, “Random Number Generators: An Evaluation and Comparison of Random.org and Some Commonly Used Generators”, 2005.

[28] C. Petit, F. Standaert, O. Pereira, T. G. Malkin, M. Yung, “A Block Cipher based PRNG Secure Against Side-Channel Key Recovery”, UCL Crypto Group, Universite catholique de Louvain. Dept. of Computer Science, Columbia University., Google Inc., 2009.

[29] T. Mulder, J. Peters, “Entropy Rate of Stochastic Processes”, pp 6-7, 2015.

Appendix 1 – Entro.py Library Source Code

```
import math
"""
entrolib.py: Function library for different data entropy tests.
"""
__author__    = "Pavel Chikul"
__copyright__ = "Copyright 2018, REGLabs"


def compute_shannon(data):
    """
    Calculate Shannon entropy value for a given byte array.

    Keyword arguments:
    data -- data bytes
    """
    entropy = 0
    for x in range(256):
        it = float(data.count(x))/len(data)
        if it > 0:
            entropy += - it * math.log(it, 2)

    return entropy


def compute_entropy_graph(data, step, shannon=True, chi=True):
    """
    Calculates entropy graph values with different with different 
algorithms and step.

    Keyword arguments:
    data -- data bytes
    step -- step in bytes
    shannon -- indicates whether Shannon entropy should be calculated
    chi -- indicates whether Chi-Squared should be calculated
    """
    shannons = []
    chis = []
    current_position = 0

    while current_position < len(data):
```

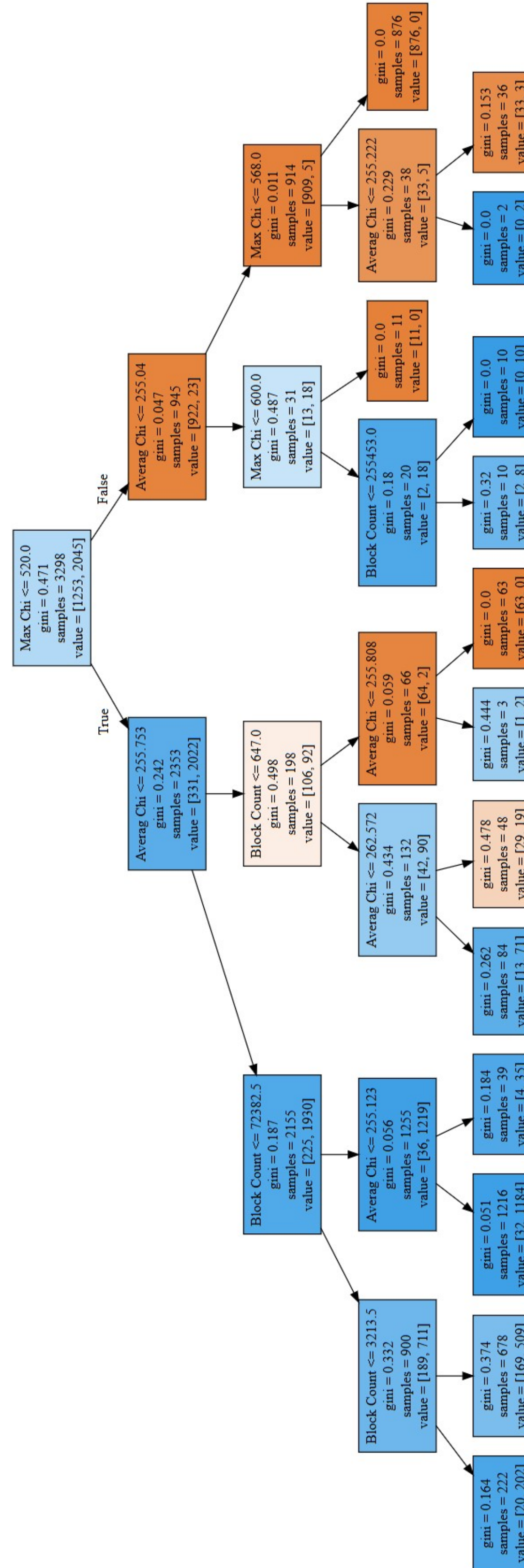
```
        if shannon:

shannons.append(compute_shannon(data[current_position:current_position +
step]))

        if chi:

chis.append(compute_chi_squared(data[current_position:current_position +
step]))

        current_position += step # Note: We skip the last chunk if it's
less than step.

    return (shannons, chis)


def compute_monte_carlo_pi(data):
    """
    Calculate Monte Carlo Pi approximation for a given byte array.

    Keyword arguments:
    data -- data bytes
    """
    set_length = int(len(data) / 2) # All of the set values are inside
square.
    r_square = 128 * 128
    circle_surface = 0

    for i in range(set_length):
        if ((data[i*2] - 128) ** 2 + (data[i*2+1] - 128) ** 2) <=
r_square:
            circle_surface += 1

    return 4 * circle_surface / set_length


def get_pi_deviation(pi_value):
    """
    Returns an absolute percentage of difference between the provided Pi
value and canonic.

    Keyword arguments:
    pi_value -- Pi value to be tested for difference
    """
    return abs(100 - (pi_value * 100 / math.pi))


def compute_chi_squared(data):
    """
```

```
    Calculate Chi-Squared value for a given byte array.

    Keyword arguments:
    data -- data bytes
    """
    expected = len(data) / 256
    observed = [0] * 256
    for b in data:
        observed[b] += 1

    chi_squared = 0
    for o in observed:
        chi_squared += (o - expected) ** 2 / expected

    return chi_squared
```

Appendix 2 – Software and command line commands used to generate test data

Algorithm	Software	OS	Command Line
AES	AESCrypt 3.1	Windows	aescrypt.exe -e -p <password> -o <out_file> <in_file>
BLOWFISH	mcrypt	Linux	mcrypt <in_file> -a blowfish -k <password>
CAST5	Gpg4Win 3.1	Windows	gpg -r <key> -o <out_file> -e <in_file>
DEFLATE	zip	Linux	zip -9 -j <out_file> <in_file>
DES	mcrypt	Linux	mcrypt <in_file> -a des -k <password>
ENIGMA	mcrypt	Linux	mcrypt <in_file> -a enigma -k <password>
LZMA	7Zip 18.05	Windows	7z.exe a -mx9 -t7z <out_file> <in_file>
LZSS	WinRAR 5.60	Windows	Rar.exe a -m5 -s <out_file> <in_file>

Appendix 3 – Full Decision Tree structure



Appendix 4 – Machine Learning Tests Source Code (cropped)

```
#!/usr/bin/python3
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier, export_graphviz
import numpy as np
import pydot

import warnings
with warnings.catch_warnings():
    warnings.filterwarnings("ignore",category=DeprecationWarning)
    from sklearn.ensemble import RandomForestClassifier,
AdaBoostClassifier

from sklearn.tree import _tree


if __name__ == '__main__':
    # Load data and drop unnecessary features.
    data = pd.read_csv("test06_unified.csv", sep=';')
    data.drop(["File Name", "Peak IDs", "Peak Positions", "Data Size"],
axis=1, inplace=True)

    # Separate features from target class.
    y = data["Encrypted"]
    x = data.drop("Encrypted", axis=1)

    # Split items 70/30 for training/validation.
    x_train, x_valid, y_train, y_valid = train_test_split(x, y,
test_size=0.3, random_state=17) # random seed 17 guarantees the
split/shuffle will be the same every time

    # Default decision tree.
    first_tree = DecisionTreeClassifier(random_state=17)
    print(f"Decision Tree        {np.mean(cross_val_score(first_tree,
x_train, y_train, cv=5))}") # k-fold k = 5

    # Default KNN.
    first_knn = KNeighborsClassifier()
    print(f"KNN                  {np.mean(cross_val_score(first_knn,
x_train, y_train, cv=5))}")
```

```
    # Decision tree tweaking.
    tree_params = { "max_depth": np.arange(1, 11) } # Look for best
max_depth param in range 1-10.
    tree_grid = GridSearchCV(first_tree, tree_params, cv=5, n_jobs=-1)
    tree_grid.fit(x_train, y_train) # Train all variations and find best
score/estimator.
    print(f"Grid Decision Tree   {tree_grid.best_score_}  Params:
{tree_grid.best_params_}")

    # KNN tweaking.
    knn_params = { "n_neighbors": np.arange(1, 100) } # Look for best
neighbor count in range 1-100.
    knn_grid = GridSearchCV(first_knn, knn_params, cv=5, n_jobs=-1)
    knn_grid.fit(x_train, y_train) # Train all variations and find best
score/estimator.
    print(f"Grid KNN             {knn_grid.best_score_}  Params:
{knn_grid.best_params_}")

    # Default random forest.
    first_forest = RandomForestClassifier(random_state=17)
    print(f"Random Forest        {np.mean(cross_val_score(first_forest,
x_train, y_train, cv=5))}")

    # Tweaked random forest.
    forest_params = { "max_depth": np.arange(1, 11), "n_estimators":
np.arange(10, 21)}
    forest_grid = GridSearchCV(first_forest, forest_params, cv=5,
n_jobs=-1)
    forest_grid.fit(x_train, y_train)
    print(f"Grid Forest          {forest_grid.best_score_}  Params:
{forest_grid.best_params_}")

    # Default Ada Boost.
    first_ada = AdaBoostClassifier(random_state=17)
    print(f"Ada Boost            {np.mean(cross_val_score(first_ada,
x_train, y_train, cv=5))}")
    #first_ada.fit(x_train, y_train)

    # Tweaked Ada Boost.
    ada_params = { "n_estimators": np.arange(10, 70) }
    ada_grid = GridSearchCV(first_ada, ada_params, cv=5, n_jobs=-1)
    ada_grid.fit(x_train, y_train)
    print(f"Grid Ada             {ada_grid.best_score_}  Params:
{ada_grid.best_params_}")

    # Logistic Regression.
    first_lr = LogisticRegression()
    print(f"Logistic Regression  {np.mean(cross_val_score(first_lr,
x_train, y_train, cv=5))}")
```

```
    first_lr.fit(x_train, y_train)


    print()
    print("VALIDATION")
    print(f"Tree    {accuracy_score(y_valid,
tree_grid.predict(x_valid))}")
    print(f"KNN     {accuracy_score(y_valid, knn_grid.predict(x_valid))}")
    print(f"Forest {accuracy_score(y_valid,
forest_grid.predict(x_valid))}")
    print(f"Ada     {accuracy_score(y_valid, ada_grid.predict(x_valid))}")
    print(f"LReg    {accuracy_score(y_valid, first_lr.predict(x_valid))}")

    # Export tree as a .dot file for later visualization.
    export_graphviz(tree_grid.best_estimator_, out_file="tree_graph.dot",
feature_names=x.columns, filled=True, max_depth=3)

    # Print out tree as code snippet.
    tree_to_code(tree_grid.best_estimator_, list(x.columns))
```

Appendix 5 – Autopsy Module Source Code (cropped)

```
def is_encrypted(max_chi, average_chi, block_count):
    if max_chi <= 520.0:
        if average_chi <= 255.75341796875:
            if block_count <= 72382.5:
                if block_count <= 3213.5:
                    return 1
                else:   # if block_count > 3213.5
                    return 1
            else:   # if block_count > 72382.5
                if average_chi <= 255.1233673095703:
                    return 1
                else:   # if average_chi > 255.1233673095703
                    return 1
        else:   # if average_chi > 255.75341796875
            if block_count <= 647.0:
                if average_chi <= 262.5716857910156:
                    return 1
                else:   # if average_chi > 262.5716857910156
                    return 0
            else:   # if block_count > 647.0
                if average_chi <= 255.80758666992188:
                    return 1
                else:   # if average_chi > 255.80758666992188
                    return 0
    else:   # if max_chi > 520.0
        if average_chi <= 255.04031372070312:
            if max_chi <= 600.0:
                if block_count <= 255453.0:
                    return 1
                else:   # if block_count > 255453.0
                    return 1
            else:   # if max_chi > 600.0
                return 0
        else:   # if average_chi > 255.04031372070312
            if max_chi <= 568.0:
                if average_chi <= 255.22213745117188:
                    return 1
                else:   # if average_chi > 255.22213745117188
                    return 0
            else:   # if max_chi > 568.0
                return 0

class EntropyIngestModuleFactory(IngestModuleFactoryAdapter):
    moduleName = "Entro.py Ingest Module"

    def getModuleDisplayName(self):
        return self.moduleName

    def getModuleDescription(self):
        return "Ingest module that provides information on file entropy and gives
the assumption if the file is encrypted."
```

...

```
    def process(self, file):
        # Skip non-files
        if ((file.getType() == TskData.TSK_DB_FILES_TYPE_ENUM.UNALLOC_BLOCKS) or
            (file.getType() == TskData.TSK_DB_FILES_TYPE_ENUM.UNUSED_BLOCKS) or
            (file.isFile() == False)):
            return IngestModule.ProcessResult.OK

        temp_file = os.path.join(Case.getCurrentCase().getTempDirectory(),
str(file.getId()) + ".tmp")
        ContentUtils.writeToFile(file, File(temp_file))
        data = []
        with open(temp_file, "rb") as binary_file:
            data = binary_file.read()

        # This is apparently needed for Python 2.7.
        data = [ord(e) for e in data]

        shannon = compute_shannon(data)

        if shannon > 7.85:
            _, chis = compute_entropy_graph(data, 32, shannon=False)
            max_chi = max(chis)
            avg_chi = float(sum(chis)) / len(chis)
            encrypted = is_encrypted(max_chi, avg_chi, len(chis))

            self.log(Level.INFO, file.getName() + " Max Chi: " + str(max_chi) +
", Avg Chi: " + str(avg_chi) + ", Blocks: " + str(len(chis)) + ", Encrypted: " +
str(encrypted))

            chi_square = compute_chi_squared(data)
            pi_deviation = get_pi_deviation(compute_monte_carlo_pi(data))

            if encrypted:
                self.potentially_encrypted_files_found += 1
                artifact =
file.newArtifact(BlackboardArtifact.ARTIFACT_TYPE.TSK_ENCRYPTION_SUSPECTED)
                attribute =
BlackboardAttribute(BlackboardAttribute.ATTRIBUTE_TYPE.TSK_COMMENT,
                    EntropyIngestModuleFactory.moduleName, "Encryption suspected.
\nEntropy: " + str(shannon) +
                    "\nChi-Squared: " + str(chi_square) +
                    "\nMax Chi-Squared in graph: " + str(max_chi) +
                    "\nAverag Chi-Squared in graph: " + str(avg_chi) +
                    "\nMonte Carlo Pi Deviation: " + str(pi_deviation))
                artifact.addAttribute(attribute)

                # Fire an event to notify the UI and others that there is a new
artifact
                IngestServices.getInstance().fireModuleDataEvent(
                    ModuleDataEvent(EntropyIngestModuleFactory.moduleName,
                        BlackboardArtifact.ARTIFACT_TYPE.TSK_ENCRYPTION_SUSPECTED
, None))
```

```
            else:
                self.potentially_compressed_files_found += 1
                artifact =
file.newArtifact(BlackboardArtifact.ARTIFACT_TYPE.TSK_INTERESTING_FILE_HIT)
                attribute =
BlackboardAttribute(BlackboardAttribute.ATTRIBUTE_TYPE.TSK_SET_NAME,
                    EntropyIngestModuleFactory.moduleName, "Extreme Compression
Suspected")
                artifact.addAttribute(attribute)
                attribute =
BlackboardAttribute(BlackboardAttribute.ATTRIBUTE_TYPE.TSK_COMMENT,
                    EntropyIngestModuleFactory.moduleName, "Extreme compression
suspected. \nEntropy: " + str(shannon) +
                    "\nChi-Squared: " + str(chi_square) +
                    "\nMax Chi-Squared in graph: " + str(max_chi) +
                    "\nAverag Chi-Squared in graph: " + str(avg_chi) +
                    "\nMonte Carlo Pi Deviation: " + str(pi_deviation))
                artifact.addAttribute(attribute)

                # Fire an event to notify the UI and others that there is a new
artifact
                IngestServices.getInstance().fireModuleDataEvent(
                    ModuleDataEvent(EntropyIngestModuleFactory.moduleName,
                        BlackboardArtifact.ARTIFACT_TYPE.TSK_INTERESTING_FILE_HIT
, None))


            # Fire an event to notify the UI and others that there is a new
artifact
            IngestServices.getInstance().fireModuleDataEvent(
                ModuleDataEvent(EntropyIngestModuleFactory.moduleName,
                    BlackboardArtifact.ARTIFACT_TYPE.TSK_INTERESTING_FILE_HIT,
None))


        return IngestModule.ProcessResult.OK
```