MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Disk sector content analysis and visualization

Bachelor's Thesis

JAKUB MALOŠTÍK

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Disk sector content analysis and visualization

Bachelor's Thesis

JAKUB MALOŠTÍK

Advisor: Ing. Milan Brož, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2022

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jakub Maloštík

Advisor: Ing. Milan Brož, Ph.D.

Acknowledgements

These are the acknowledgements for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

keyword1, keyword2, ...

Contents

List of Figures

Introduction

Disks (e.g., hard drives, SSDs, Flash drives) are divided into atomic parts named sectors, which are represented as blocks in the software layer. Sectors store a fixed amount of data, usually 512 bytes or 4KiB, but other sector sizes can be used. Sectors may contain partition tables, file system information, files or be empty.

Sectors can contain specific byte patterns which can be analyzed and used to identify the type of content stored in the sector. When a byte pattern is not present, sector content can be analyzed for entropy to estimate whether it is encrypted. A good way to get an idea about which parts of the disk are encrypted and where filesystem data is stored is to visualize the data. This visualization will allow humans to distinguish between different data encryption methods such as filesystem-level and full-disk encryption and even uncover faulty encryption. Visualizing can also be very useful as an illustration while teaching.

The utility introduced in this bachelor's thesis analyzes the sectors of a user-specified size of a provided disk image and visualizes the result using the Pillow Python library. The utility is also easily extensible by other output methods.

The text of this thesis is structured into five chapters. Chapter number one explains the foundations of the thesis and examines prior work. Chapter number two lists some byte patterns of sectors and discusses algorithms for their detection. Chapter number three discusses algorithms used to calculate entropy and possible issues with their accuracy. Chapter number four discusses ways of visualization and their advantages and disadvantages. The last chapter concludes with an evaluation of the resulting utility.

The resulting utility is available on GitHub[1] under the MIT License.

1. `https://github.com/malon43/entropy-visualization`

1 Sector content visualization

This review focuses on works on the topics of block pattern and encryption detection and ways of visualization

1.1 Analysis

Each disk is divided into tens, even hundreds of millions of sectors. Each disk sector stores some data. Sectors of empty new drives would be mostly initialized with a pattern of zeroes, except for partitioning tables and file system metadata.

Most recent drives use 4KiB sized sectors, also known as Advanced Format, but still provide backward compatibility with older systems which expect 512B sector size with 512B sector size emulation. [1]

This section describes parts of analyzing the disk sectors.

1.1.1 Block patterns

Sector byte pattern is a specific configuration of bytes, which would indicate what this sector is used for. For example, a repeated pattern of byte `x00` often signalizes that this sector has not been used yet, or that the blocks have been freed by the TRIM command. The TRIM command is used by the software to inform the drive which sectors no longer contain user data in order to increase performance.[2] Or, bytes `x55xAA` at the end of the sector would signalize a block containing master boot record (MBR). However, while in many cases, analysis for positions of bytes is not as time-intensive as analysis of randomness or single-byte patterns, multiple problems show up:

- Testing for many positions and byte configuration will add up.
- Files with magic bytes may be contained in the first sector where the file is stored, but there is no easy way of telling whether the file simply ends, continues on the next sector, or is placed in a completely different sector.
- As the magic bytes at the beginning of the file are only visible when the file is unencrypted, and correctly encrypted sectors

should appear random, the sector containing the magic bytes will mostly get picked up by the randomness analysis and marked as *not random*.

Most works focusing on detecting patterns of bytes on sectors[3, 4] do it through the lens of forensic analysis and use the filesystem metadata in combination with magic bytes of files to allow the user to find information faster. These can provide beneficial information when identifying common patterns of entire sectors or repeating portions of bytes in a single sector.[3]

1.1.2 Randomness

In order to properly classify all disk sectors, one cannot rely exclusively on byte patterns since files can span multiple sectors and can even be encrypted. In this case, it is possible to check the predictability of byte values or even of single bits.

In order to precisely differentiate random data, the provided samples would need to be in the order of gigabytes, which is far from the provided 512 or 4096 Bytes. However, we can at least get an estimate using the techniques described in this subsection.

Based on these estimates and the fact that correctly encrypted data should appear as random, sectors with high randomness can be marked as potentially encrypted.

Entropy

Shannon's entropy calculates the amount of information in bits provided by each byte value in the sector.[5] For example, the entropy of 8 bits means that every byte value is contained the same number of times (i.e., exactly $\frac{s}{256}$ times). Whereas the entropy value of 0 means that only a single byte value is contained and is repeated through the whole sector.

Shannon's entropy of a sector $H(S)$ can be calculated using:

$$H(S) = -\sum_{i=0}^{255} (P(x_i) \log_2(P(x_i)))$$

Where $P(x_i)$ represents the probability of byte value from the sector being i (i.e., the number of times value i appears in the sector divided by the number of all bytes in the sector). Which can be then used to calculate normalized entropy $\mu(S)$ for the sector:

$$\mu(S) = \frac{H}{H_{max}} = -\frac{1}{8}\sum_{i=0}^{255}(P(x_i)\log_2(P(x_i)))$$

Where H is the entropy of the sector and H_{max} is the maximum possible entropy a sector can achieve, which for sector sizes of non-zero multiples of 256 is always 8. Normalized Shannon's entropy ranges from 0, the least random (a single repeated byte value), to 1, the most random (every byte value is contained in the sector an equal amount of times).

Correctly encrypted data should appear as random. From the definition of entropy, random data cannot be described easily, and each bit of truly random data should require one bit to be described. Based on this, one can estimate whether the sector contains encrypted data from the normalized entropy.

However, the goal of the compression algorithms is to try to encode the provided data into a smaller number of bits. This naturally increases the entropy of the data as each bit encodes as much information as the compression algorithm was able to compress into it. That means that most sectors containing compressed file formats like videos, jpeg images, or zip files will be almost indistinguishable from encrypted sectors by entropy as there is no simple line where all sectors with a higher entropy are encrypted, and all with lower entropy are not.

After this point, each time entropy is mentioned, it refers to the normalized version.

Another problem that arises when using Shannon's entropy is that the order of the values is completely disregarded. For example, simple counting up (x00 x01 ... xFE xFF) repeatedly, which is often part of files, results in the entropy of 1, despite this clearly not being random.

Most works I found that attempted to use entropy calculation to classify small data samples used Shannon's entropy despite its drawbacks mentioned above. However, each work aimed to use the calculated entropy differently. Some used[3] or tried to use[4] it to classify blocks for use in file carving and not encryption detection.

Other works used[6] or tried to use[7] entropy calculation as input or part of the input for machine learning trained to classify network packets. Work[6] also suggested using Tsallis entropy for calculation. However, the work did not attempt to calculate Tsallis entropy and instead decided to focus on Shannon's entropy.

Another work worthy of consideration[8] compared multiple entropy estimation algorithms. The work concluded by recommending the Miller-Madow method for uniform byte value distributions to estimate entropy. Entropy estimation will be helpful when considering the efficiency and speed of the entropy calculation.

Chi-squared test

The chi-squared test or χ^2 test is used to determine whether or not the data fit our expectations.[9] For example, consider flipping five fair coins and counting flipped heads. The probability distribution of the results (assuming that the coins cannot land on their side) would look like this:

number of heads	0	1	2	3	4	5
probability	$\frac{1}{32}$	$\frac{5}{32}$	$\frac{10}{32}$	$\frac{10}{32}$	$\frac{5}{32}$	$\frac{1}{32}$

First, we select a significance level α (e.g. $\alpha = 0.05$). Significance level signifies where the cutoff is when our hypothesis is rejected.

Then, after repeating the experiment of flipping five coins 160 times and adding up the results, we get the following table:

number of heads	0	1	2	3	4	5
number of flips (X)	2	8	34	64	44	8
expected number of flips (E)	5	25	50	50	25	5

Given counts of variables X_i, expected counts of variables $E_i = 160 * \text{probability}$, and the number of columns n, chi-square test statistic can be calculated using:

$$\chi^2 = \sum_{i=0}^{n-1} \left(\frac{(X_i - E_i)^2}{E_i} \right)$$

After getting the value approximation, the corresponding value from the cumulative chi-squared distribution for $n-1$ degrees of freedom represents how likely the measured data is from the distribution of our null hypothesis. We can reject the null hypothesis if the p-value (i.e., likelihood of getting results as extreme as the results obtained) is smaller than half of our chosen α value. On the other hand, receiving exact or very close numbers from the distribution with truly random events is equally unlikely. Therefore we can also reject the hypothesis when the p-value is larger than $1-\alpha$. [10]

So, for the coin example, the chi-square statistic is calculated:

$$\chi^2 = \frac{(2-5)^2}{5} + \frac{(8-25)^2}{25} + \cdots + \frac{(8-5)^2}{5} = 38.64$$

After calculating the image of 38.64 under the chi-square cumulative distribution function for 5 degrees of freedom, we can see that $F_5(38.64) = 0.9999997$, which means that the p-value $1 - 0.9999997$ is smaller than our $\frac{\alpha}{2} = 0.0025$, and we can therefore reject our null hypothesis, meaning that the fact that the measured coined flips were made using fair coins is less than 5%. And indeed, the obtained counts are from tests using two fair and three rigged coins with the probability of getting heads of $\frac{2}{3}$. After calculating the chi-square statistic for the hypothesis with rigged coins, we get $\chi^2 = 4.141$ and $F_5(4.141) = 0.4707$ and since $1 - 0.4707 > \frac{\alpha}{2}$ and $0.4707 > \frac{alpha}{2}$, this hypothesis cannot be rejected.

As the chi-square test is only an approximation and gets more precise with more data, expected values should be at least 5, and it is preferable that they are much higher. [10]

For detection of random numbers, it is possible, for example, to create a column for each possible number, a column for ranges of numbers, or create a column for each remainder after division by a preselected number. Since the distribution of truly random numbers should be uniform, the expected value (E_i) should be the same for all columns. When the null hypothesis of uniformity of the numbers with sufficiently small α gets rejected, we can assume that the numbers are not random enough.

With the chi-squared test, the problem of counting up, unlike with Shannon's entropy, does not arise. When each number is represented the same number of times, the chi-squared statistic for uniform distribution is equal to zero. The image of 0 in the χ^2 cumulative distribution for any number of degrees of freedom is 0, giving the p-value of 1, which is then rejected and marked as *perfect random*.

1.2 Visualization

After classifying all disk sectors based on byte patterns and entropy, it all comes down to visualizing the gathered data. While it would be certainly possible to draw a histogram of all sectors' entropy values or a pie graph based on detected patterns, this would not be as illustrative as the chosen approach, and much of the information about sector position in the disk would be lost. That is why the resulting utility visualizes the data using a bitmap, where each pixel represents a single sector on a disk.

Many works which were visualizing data used the most straightforward technique of *sweeping*. [11, 12, 13] This means that the first pixel is placed in the top-left corner, and each following pixel is placed to the right of the previous one except for when the position exceeds the fixed width of the image. In that case, the pixel is placed on the left-most position on the following line. This technique can be very illustrative in cases when the disk contains long sequences of equally classified sectors. However, when the disk would contain a shorter sequence, this would produce only a horizontal line with a single-pixel width, which could be hard to see and easily overlooked.

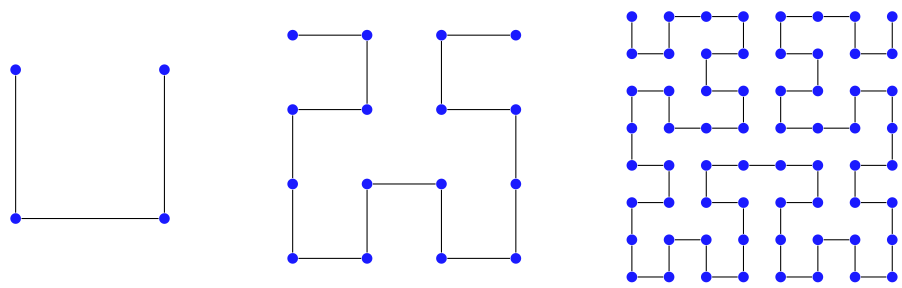


Figure 1.1: The first three Hilbert curve iterations

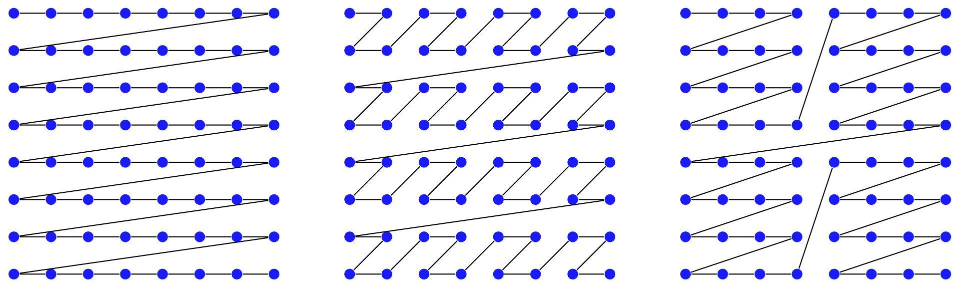


Figure 1.2: Sweeping, 2x2 block sweeping, and 4x4 block sweeping

That is why the work [14] used the more complex Hilbert space-filling curve. The Hilbert curve passes through every pixel in a square exactly once in such a recursive pattern for which consecutive pixels always share one side.[15] Moreover, placing pixels in these specific ways ensures that the shorter sequences are expanded into multiple lines and aggregated into clusters which makes them more easily visible. The curve covers a square with the side length of 2^i pixels (i.e., a total of 4^i pixels) where i is the number of iterations.

The order in which the Hilbert curve goes through pixels in a 2D plane for each of the first three iterations can be seen in the figure 1.1.

However, when using the Hilbert curve, another problem arises. There is no intuitive way to tell where the visualized sectors are located in the source image.

A middle ground between simple sweeping and the Hilbert curve would be the technique of block-sweeping I came up with. Block-sweeping uses the sweeping method to fill up a square NxN pixels in size, then continues to another NxN pixel block and places these pixel blocks in the same way simple sweeping would place individual pixels. This means that sweeping is just a version of block-sweeping, with pixel blocks of 1x1 pixels. By employing this technique, most shorter sequences are still more pronounced by getting expanded into multiple lines, and the position of pixels in the image more closely resembles the sector position in the source image than in the Hilbert curve. However, same as with sweeping, consecutive sectors are not always guaranteed to be right next to each other.

Figure 1.2 depicts the order in which the sweeping, 2x2 block sweeping, and 4x4 block sweeping methods visit pixels.

2 Used tools

This chapter describes the third-party python libraries the utility uses for both sector classification and visualization.

2.1 Pillow

Pillow[16] is an image manipulation library for Python, which is a fork of the discontinued library PIL[17]. This library is used to visualize the analysis results. Since the results should be visualized as an image, where each pixel represents a single disk sector, statistical visualization libraries like Matplotlib[18], seaborn[19], or Gnuplot[20] were not good choices as they were not created with this exact type of visualization in mind. While they provide the means to create such visualizations, they are not as straightforward as the means provided by Pillow. While Pillow offers many more features beyond the very basics needed, it still keeps the interface for drawing one pixel at a time very simple.

2.1.1 Image

The module `PIL.Image` provides an essential toolkit for manipulating images. Given the image mode (e.g., RGB or RGBA) and image size, function `Image.new` creates an instance of the `PIL.Image.Image` class. The `Image` class stores the state of the resulting image and can be modified using its methods.

The method `Image.putpixel` modifies the state of the Image object and changes the color of the pixel on the given coordinates to the given color. `Image.save` tries to store the image on the provided path, and the method `Image.close` releases allocated memory. [21]

2.1.2 ImageDraw

The module `PIL.ImageDraw` is used to modify the `Image` class from `PIL.Image` in more powerful ways than just changing single pixels. Function `ImageDraw.Draw` creates a special context object for the given `Image` object, which can be used for further in-place modifications.

The class of the context object ImageDraw.ImageDraw provides a wide range of shape drawing methods.

The method ImageDraw.rectangle, allows for drawing a rectangle for provided coordinates and colors. It is also possible to specify the width and color of the rectangle outline. The method ImageDraw.text allows for writing a provided string in a font on the image. Both of these methods can be used for drawing the image legend. [22]

2.1.3 ImageFont

The module ImageFont is used for working with fonts with the Pillow library. It provides the means to load installed fonts by name or from path using the function ImageFont.truetype or to load a fallback font in case no other font is found with the function ImageFont.load_default. ImageFont.truetype returns an instance of ImageFont.FreeTypeFont and ImageFont.load_default returns ImageFont.ImageFont. While neither of these classes is a subclass of the other, thanks to Python's duck typing can still be used interchangeably. Both of these classes implement the method ImageFont.getsize for calculating the dimensions in pixels of the box occupied by provided text written in this font. [23]

2.2 Scipy

SciPy[24] is a Python library for scientific computing. This library is used to calculate the inverse cumulative distribution function for the chi-squared distribution. SciPy provides a couple of different modules, but only one was used in the implementation.

2.2.1 stats

The module stats can be used to calculate values of many discrete[25] or continuous[26] statistical distributions, including the chi-squared distribution. The module provides callable object stats.chi2.ppf, which calculates the percentile point function (inverse cumulative distribution function) for given probability and degrees of freedom.[27] The resulting value is then used as the threshold or limit for when the results of the chi-square statistic are significant and should therefore be marked as *perfect random* or *not random*.

3 Implementation

This chapter focuses on the individual parts of the implementation.

The utility reads the provided disk image in blocks of the provided block size, analyzes the read data, and passes the results alongside information about the position of the read block to the visualization class.

3.1 Analysis

The utility provides several analysis methods. These methods are each implemented as a class. Each analysis class implements a `calc` method, which analyzes the provided buffer and returns the results. The results take the form of a tuple and contain: detected randomness from the range $\langle 0, 1 \rangle$, result flag, and a possible result argument (used to pass the single-byte patterns if found).

3.1.1 Entropy

The provided buffer's Shannon entropy can be calculated using the formula mentioned in the subsubsection 1.1.2. However, since $P(x_i) = \frac{c_i}{s}$ where c_i is the number of times the value i was in the buffer and s is the unchanging sector size, the formula for normalized entropy can be simplified to

$$-\frac{1}{8s}\sum_{i=0}^{255}(c_i * \log_2(c_i)) + \log_2(s)$$

The counts c_i are calculated using `Counter` from the `collections` module[28]. The fact that the `Counter` object only iterates over nonzero counts does not matter. By the entropy definition, the impossible results should not affect it.

However, Shannon's entropy is not a statistical test, and its results are not used to distinguish between random and not random. Therefore there are no levels of significance, and the analysis result is just a number from the range 0-1. While this analysis is not helpful to recognize random from non-random data at a glance, it can illustrate whether sectors with similar entropy (and therefore with similar file contents) are stored next to each other. Image generated with Shannon's entropy can be found in the figure FIGURE.

3.1.2 Chi-square

The implementation includes multiple options for analysis by chi-square test. Each option takes n consecutive bits and treats them as a separate number. If the sector size is not a multiple of n, the remaining bits are ignored.

The chi-square statistic can be calculated using the formula mentioned in the subsubsection 1.1.2. As the numbers should be from the uniform distribution and the expected count for each value should be the same, the formula can be modified to

$$\frac{1}{E}\sum_{i=0}^{2^n-1}((X_i - E)^2)$$

where E is the expected count of each number and can be calculated using

$$E = \frac{\lfloor \frac{8s}{n} \rfloor}{2^n}$$

where s is the sector size in bytes.

Each option, while easily generalizable, is implemented separately due to possible speed improvements. (i.e., there is no need for reading a byte bit per bit and then merging the bits into an eight-bit number again; however, for $n = 3$, there is no such easy workaround) `chi2-8` simply counts each byte value, `chi2-4` separates the byte values into halves using the python bitwise `&` and `>>` operators, `chi2-3` goes through each byte bit-by-bit, and `chi2-1` uses the `int.bit_count method` to get the number of bits set to 1 in the byte value.

Each method first calculates the expected count of each value. If this value is less than 5, a warning about possible issues with precision[10] is printed. This, however, occurs only for the combination of chi2-8 analysis and the sector size of 512 bytes (or lower, but 512 is the smallest commonly used sector size). Then the method precalculates limits for the chi-square statistic, for which the sector is marked as *suspiciously random*, *random*, or *not random*. Then for each sector, its statistic is compared to these limits, and a matching output flag is selected. All methods also check for single-byte patterns. The most common ones (i.e., `0x00` and `0xFF`) are always checked. However, all other single-byte patterns are checked only by `chi2-8` and `chi2-4`. These two implementations were already counting the number of bytes, and this check could be done with a minimal additional performance penalty. Finding the other single-byte patterns is not essential as they are not likely to be contained unless they are part of an unencrypted stored file, which will almost always be marked as *not random* by the analysis. The only exception is with `chi2-1` for the seventy byte patterns, which contain an equal number of zeroes and ones, in which case the sector will be marked as *perfect random*.

3.2 Output and visualization

Each output method is implemented in the form of a class. Each output class should implement the following three methods.

- Method `output` for taking the analysis output and storing it.
- Method `error` to be run in case of an error to display it.
- Method `exit`, which is run before the program terminates to close any open data streams and save stored data into files.

Each output class can have an attribute `default_parameters`, which is a mapping of a parameter name to `Parameter` dataclass, which stores the type of the parameter, default value, help string, and can also store string description of the default value and list of available options.

3.2.1 Text output

The utility, while being primarily intended for visualization, also implements two methods for text output.

The first one is `sample-output`, which just formats the analysis output of each sector on a seperate line as follows: <sector number> (<sector offset>) - <randomness>, <result flag> (pattern of: <single-byte pattern if present>)

The second method, `csv`, outputs the analysis result as a CSV file. The script `from_csv.py` can later use this CSV file to generate a visualization using the other output methods without analyzing the disk image again.

3.2.2 Image output

The result of all Image output methods is a bitmap, where each pixel represents the result of an analysis of one sector. The image is created using the Pillow library.

Using the command line arguments, the user can change the image's color palette, background color, and legend text color. When specifying the background color but not the text color, the text color is determined automatically using the contrast ratio and the relative luminance in accordance with W3 guidelines.[29]

Since the image needs to fit all of the sectors and the number of sectors can vary between different disk images, it is essential to have varying proportions of the output images.

The utility also provides means for generating the images using multiple color palettes. One of the two implemented is the one used to generate all images in this thesis - the `sample` palette. The other implemented color palette is the one used in Milan Broz's blog[13] - the `asalor` palette. By default, the utility also shows the color legend for all image output methods.

3.2.3 Sweeping and block sweeping

The user can set both the image width and sweeping block size. When both are provided, it is only checked that width is a multiple of sweeping block size. When only width is provided, the smallest divisor of width larger or equal to 32 but smaller than square root of width is set as the sweeping block size. If no such number exists largest divisor of width smaller than 32 is used. The utility tries to form a rectangle closest to a square if the width is not defined. In this case, the sweeping block size is set to 32 if not specified.

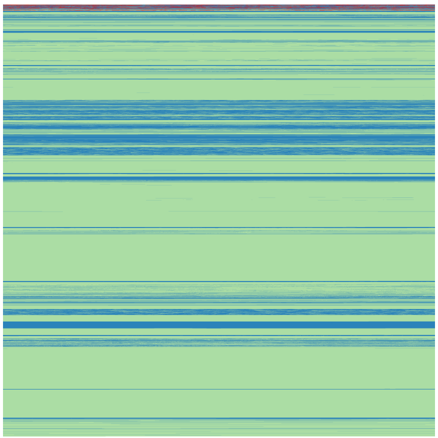
Sweeping is a special case of block sweeping with the sweeping block size set to 1.

3.2.4 Hilbert curve

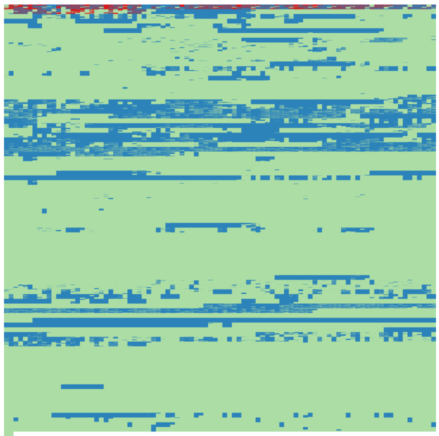
For the Hilbert curve, the image size can only be determined automatically. The smallest single curve which would fit all the sectors needs to have precisely $i = \lceil \log_4(\text{number of sectors}) \rceil$ iterations. However, it is possible to put multiple Hilbert curves under each other and chain them together. Therefore the implementation first checks whether up to three smaller curves (i.e., curves with $i - 1$ iterations) could fit all the sectors. If that is the case, width is set to 2^{i-1} and height to the needed number of curves in half-curve-width increments. Otherwise, the calculated number of iterations is used. And therefore, the height and width are the same and equal to 2^i.

The algorithm used for transforming sector sequence numbers into x and y coordinates from the Hilbert curve has been adapted from the C implementation in [30]. The C implementation from [30] is a special case of the algorithm from [31] adapted to use two dimensions and distance from the start rather than a transposed array of bits of the distance.

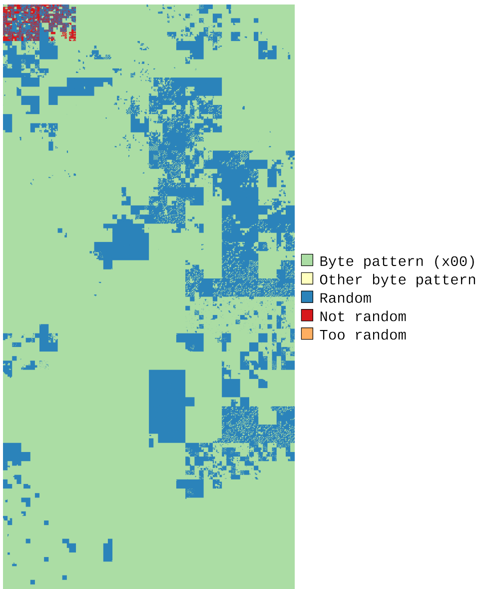
In order to chain multiple Hilbert curves under each other and still keep the locality-preserving properties, the curves portrayed in the figure 1.1 need to be mirrored along their top-left to bottom-right diagonals. This can be simply achieved by swapping the x and y coordinates.



(a) Sweeping



(b) Sweeping blocks



(c) Hilbert curve with legend

Figure 3.1: A 16GiB disk image visualized using all three image visualization methods

4 Results

4.1 Used color palette

All utility-generated images in this chapter were generated using the `sample-palette`. The `sample-palette` represents by blue color the sectors that the analysis has marked as containing only zeroes. Pixels of green color represent sectors that only contain a single-byte pattern, which is not of byte `x00`. The shade of green represents the repeated byte value (i.e., the brightest green `xFF` and dimmest green `x01`). Red represents sectors marked as *perfect random*.

And finally, by magenta are marked all the remaining sectors. The shade of magenta depends on the detected randomness by the analysis. For analysis using Shannon's entropy, the resulting shades of magenta range from black for the lowest but larger than 0 entropy to the brightest magenta (i.e., `#FF00FF` in hex color representation) for the highest entropy.

On the other hand, the chi2 analysis variants only produce two shades of magenta. The lighter shade represents sectors marked as random the darker shade marks all sectors that do not contain a single-byte pattern and are marked as not random. Furthermore, red pixels are only contained in the images analyzed through the chi2 analysis variants as Shannon's entropy analysis cannot mark sectors as *perfect random*.

Legend is not included in every image for space-saving reasons but can be found in the figure 4.1.

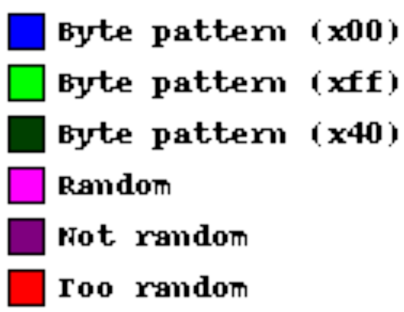


Figure 4.1: Legend for images generated using `sample-palette`

4.2 TRIM

In order to show how TRIM affects the data stored on the disk and how the utility visualizes it crpytsetup 2.4.3 was used. First, a plain dm-crypt mapping with allowed discards to a 2GiB file was created. Then through the mapping, a file system was created into which an unpacked Linux kernel 5.16.9 was copied. Then, all the files in the largest directory, `drivers`, were deleted. After deleting the files, a copy of the image was taken. Generated images of analyses of these copies can be found in figures 4.2a and 4.2c. These copies were then compared to the resulting image after calling `fstrim` on the mounted mapper, which discarded all unused sectors. Images generated from images with discarded sectors can be found in figures 4.2b and 4.2d.

At first glance, one can see the difference. The sectors marked as blue are the ones that were discarded, and it can be clearly seen how many sectors the data actually takes up and where precisely the encrypted data is stored.

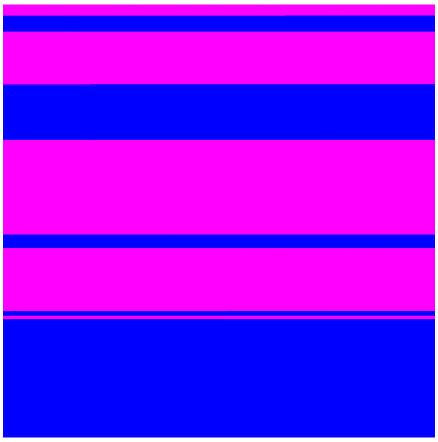
This need not necessarily be a huge security risk; however, in cases when it is vital not to show such information, it is good to be wary of such drawbacks of allowing trim commands.

Note that the light magenta is not solid, and one can, after a closer inspection, occasionally see sectors marked by red or a dark shade of magenta, which are the result of sectors randomly having the p-value of its chi-square statistic smaller than the default significance level of 0.0001. The number of these outlier pixels can be decreased by decreasing the significance level with the cost of decreasing the number of truly non-random sectors correctly identified as non-random. A closeup of these pixels can be seen in the figure 4.3.

4.3 Flawed encryption

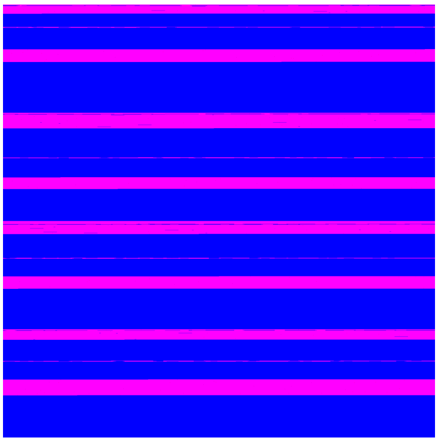
For illustration, a fresh Canonical Ubuntu 20.04[32] installation with the ext4 filesystem was created without encryption. Then a file of the size of one gigabyte containing only a repeating word 'test' was written. After analyzing the 10 GiB disk image using the analysis method of `chi2-4` and visualizing it using the Hilbert curve visualization method, the image in the figure 4.4a was produced. On the generated image,

(a) Encrypted ext4 without TRIM



(b) Encrypted ext4 with TRIM

(c) Encrypted xfs without TRIM



(d) Encrypted xfs with TRIM

Figure 4.2: Comparison of encrypted filesystems with and without TRIM generated by `chi2-4` analysis and sweeping

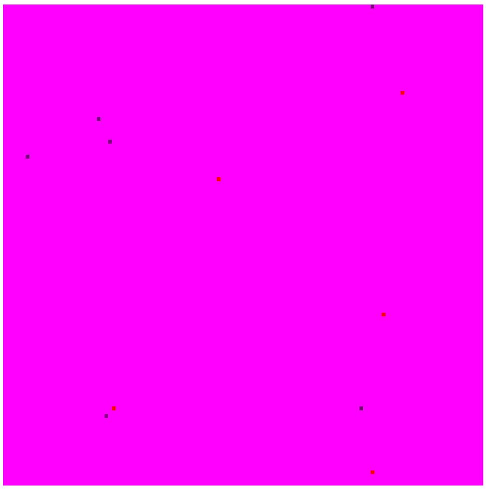


Figure 4.3: Zoomed in figure 4.2a

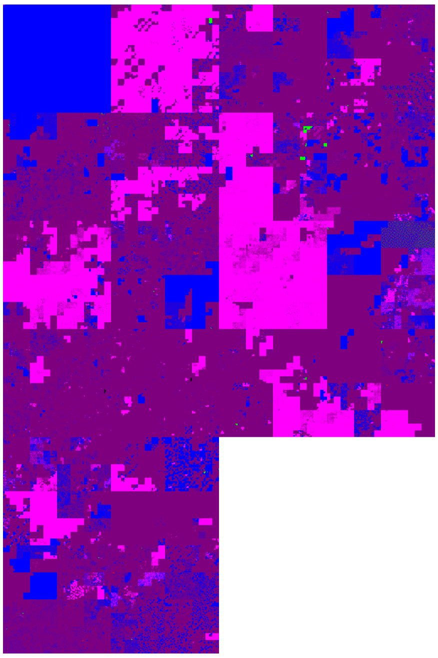
one can distinctly see many areas of sectors of zeroes and also areas of sectors of unencrypted data.

Then, since disk encryption utilities, for obvious reasons, do not provide flawed encryption methods, OpenSSL 1.1.1n[33] was used to encrypt the entire disk image with 128-bit AES in ECB mode. Using the same analysis and visualization methods on the encrypted disk image as on the unencrypted variant, the utility generated the image in the figure 4.4b. On the resulting image are evidently visible areas of detected unencrypted sectors. These are mostly the result of encrypting sectors of zeroes. However, many areas that previously contained more complex data remained marked as unencrypted. Therefore not only would trimmed sectors be affected, but this effect can also occur in sectors storing actual data.

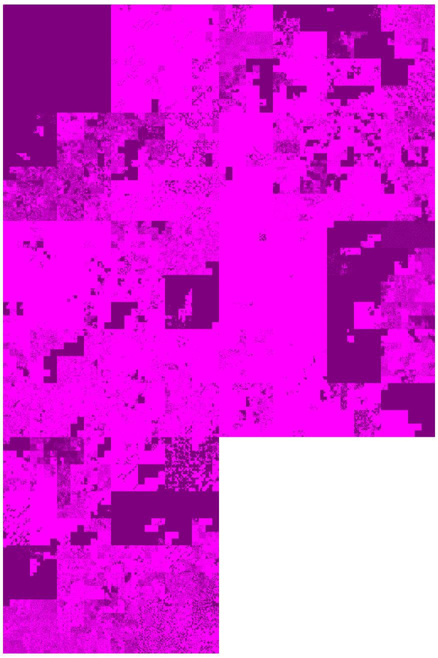
This is caused by the ECB mode's property of encrypting every same block of bytes to the same encrypted block of bytes. Since the sectors contain multiple sectors of the same blocks, the `chi2-4` analysis method picks up on it and marks the sector as not random.

4.4 Visualizing encrypted and unencrypted parts of a disk

For each of the following examples, a different disk image was used.



(a) Unencrypted



(b) Encrypted with 128-ubit AES in ECB mode

Figure 4.4: 10GiB disk images visualized using `chi2-4` analysis method and hilbert curve

4.4.1 BitLocker

For this example, a clean Windows 10 installation was created on a 50GiB virtual drive through Oracle VM VirtualBox. First, a raw copy of the disk image was taken using VBoxManage. Then, using the BitLocker user interface, full disk encryption was set up, after which another raw copy was taken. Then the images in the figure FIGURE were produced using the utility.

In the drive, there are three partitions. The first is the system partition containing hardware-specific files needed to load windows [34], the second is the data partition visible from windows under the letter C, and the last is the recovery tools partition [35].

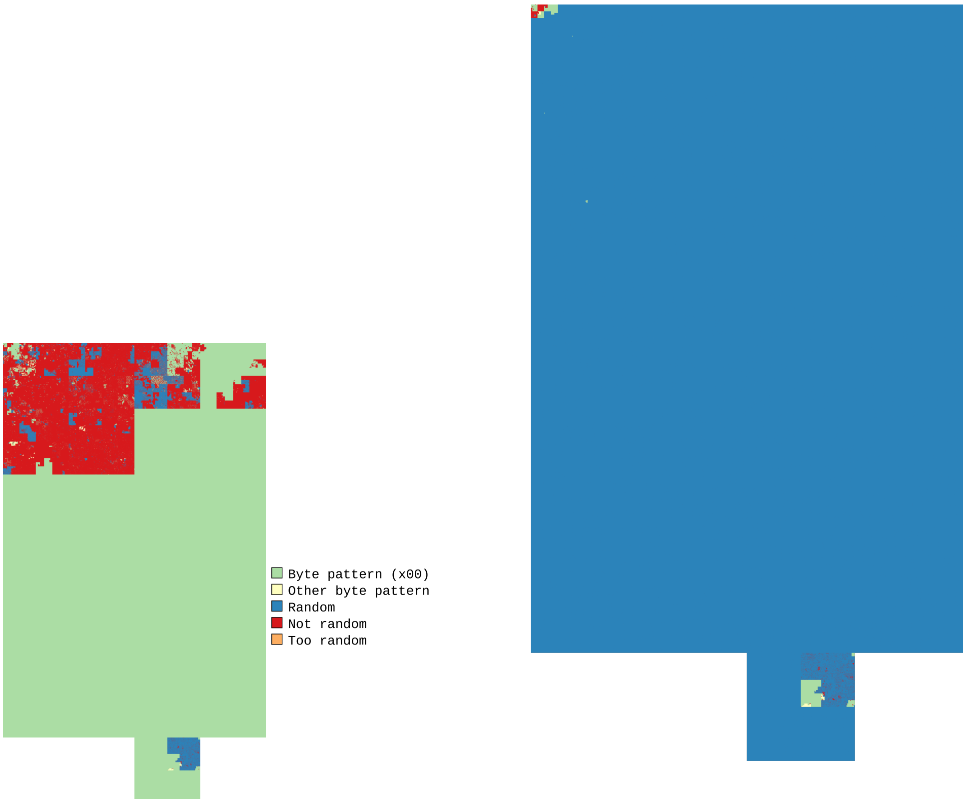
In the figure FIGURE, one can see that all sectors seem to be mostly unencrypted and, in the figure FIGURE, only the first and last partitions remain unencrypted. After closer inspection, there are also visible chunks of unencrypted sectors in the second partition, and these are sectors containing FVE metadata blocks and their padding [36]. However, Oracle VM VirtualBox does not allow trim by default. Therefore unused zeroed out sectors are not visible, and it is not easily discernible whether the sector is occupied by a file.

4.4.2 LUKS

For this example, the image from the study materials from the year 2019 of the pv204 course[1] with whole system encryption with LUKS was used. The figure FIGURE resulted from converting the image in `.vdi` to raw format and using the utility. In the figure, one can see that only the first part of the image contains chunks of unencrypted sectors. Most of the unencrypted part is the boot partition. However, after focusing only on the first two megabytes of the encrypted partition, unencrypted sectors, in the beginning, can still be clearly seen. This is the LUKS header, and it contains crucial information on decrypting the rest of the partition [37].

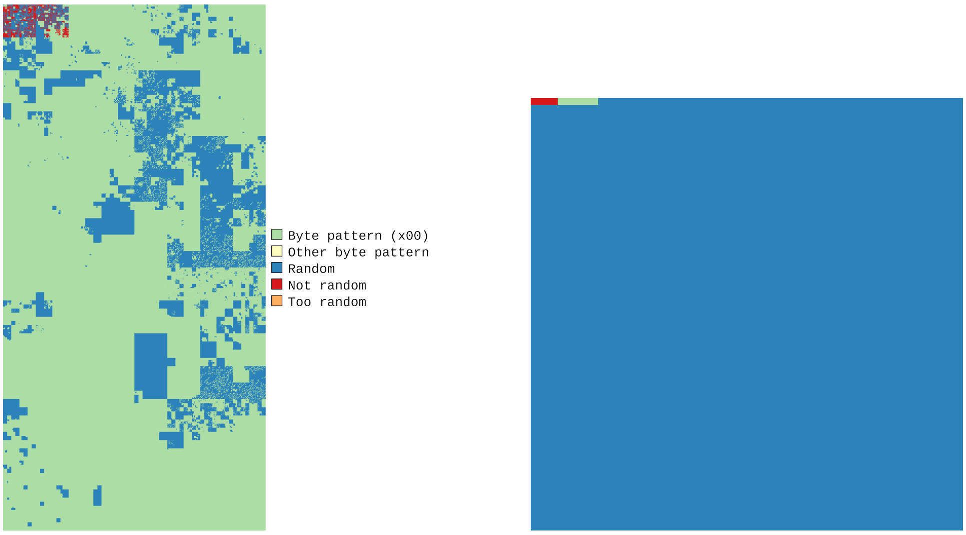
4.4.3 Veracrypt

1. `https://is.muni.cz/auth/el/fi/jaro2019/PV204/um/seminars/08_diskencryption/pv204_fde.zip`



(a) Unencrypted **(b)** Encrypted using BitLocker

Figure 4.5: Windows 10 disk image before and after encryption with BitLocker



(a) Generated image from the entire disk

(b) First two megabytes of the encrypted partition

Figure 4.6: Whole system encryption with LUKS

5 Conclusion

Bibliography

1. *Transition to advanced format 4K sector hard drives* [online] [visited on 2021-12-12]. Available from: https://www.seagate.com/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/.
2. MCMILLEN, Wes. *White Paper: Western Digital Trim Command - General Benefits for Hard Disk Drives* [online]. 2021-12 [visited on 2022-03-05]. Tech. rep. Available from: https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-purple-hdd/whitepaper-generic-benefit-for-hard-disk-drive.pdf.
3. FOSTER, Kristina. *Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus*. 2012. Available also from: https://apps.dtic.mil/sti/citations/ADA570831. MA thesis. Naval Postgraduate School Montery CA.
4. GARFINKEL, Simson; MCCARRIN, Michael. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigation*. 2015, vol. 14, S95–S105. Available from DOI: 10.1016/j.diin.2015.05.001.
5. SHANNON, C. E. A mathematical theory of communication. *The Bell System Technical Journal*. 1948, vol. 27, no. 3, pp. 379–423. Available from DOI: 10.1002/j.1538-7305.1948.tb01338.x.
6. WANG, Yipeng; ZHANG, Zhibin; GUO, Li; LI, Shuhao. Using Entropy to Classify Traffic More Deeply. In: *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*. 2011, pp. 45–52. Available from DOI: 10.1109/NAS.2011.18.
7. BEZAWADA, Bruhadeshwar; BACHANI, Maalvika; PETERSON, Jordan; SHIRAZI, Hossein; RAY, Indrakshi; RAY, Indrajit. Behavioral Fingerprinting of IoT Devices. In: *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*. Toronto, Canada: Association for Computing Machinery, 2018, pp. 41–50. ASHES '18. ISBN 9781450359962. Available from DOI: 10.1145/3266444.3266452.

8. FIALLO, Ernesto; LEGÓN, C.M. Comparison of estimates of entropy in small sample sizes. 2019. Available from DOI: `10.13140/RG.2.2.19371.28960`.
9. PEARSON, Karl. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*. 1900, vol. 50, no. 302, pp. 157–175. Available from DOI: `10.1080/14786440009463897`.
10. KNUTH, Donald Ervin. *The Art of Computer Programming Vol.2: Seminumerical Algorithms*. 2nd ed. Addison-Wesley, 1981. ISBN 0-201-03822-6.
11. HARGREAVES, Christopher. Visualisation of allocated and unallocated data blocks in digital forensics. In: 2013. Available also from: `https://www.researchgate.net/publication/282538793_Hash-based_carving_Searching_media_for_complete_files_and_file_fragments_with_sector_hashing_and_hashdb`.
12. CHARALAMPIDIS, Ioannis. *Visualising Filesystems* [online]. 2018-11-08 [visited on 2021-12-14]. Available from: `https://www.linkedin.com/pulse/visualising-filesystems-ioannis-charalampidis`.
13. BROŽ, Milan. *TRIM & dm-crypt ... problems?* [Online]. 2011-08-14 [visited on 2021-12-14]. Available from: `https://asalor.blogspot.com/2011/08/trim-dm-crypt-problems.html`.
14. CORTESI, Aldo. *Visualizing binaries with space-filling curves* [online]. 2011-12-23 [visited on 2021-12-14]. Available from: `https://corte.si/posts/visualisation/binvis/`.
15. HILBERT, David. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. 1891. Available also from: `http://www.digizeitschriften.de/en/dms/img/?PID=GDZPPN002253127&physid=phys476#navi`.
16. KEMENADE, Hugo van; MURRAY, Andrew; WIREDFOOL; CLARK, Alex; ET AL. python-pillow/Pillow: 9.0.1. 2022. Available from DOI: `10.5281/zenodo.5953590`.

17. *Python Imaging Library* [online] [visited on 2022-02-28]. Available from: `https://web.archive.org/web/20150316203935/http://effbot.org/zone/pil-index.htm`.

18. HUNTER, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*. 2007, vol. 9, no. 3, pp. 90–95. Available from DOI: `10.1109/MCSE.2007.55`.

19. WASKOM, Michael L. seaborn: statistical data visualization. *Journal of Open Source Software*. 2021, vol. 6, no. 60, p. 3021. Available from DOI: `10.21105/joss.03021`.

20. Gnuplot [online]. [N.d.] [visited on 2022-02-11]. Available from: `http://www.gnuplot.info/`.

21. *Image Module* [online]. [N.d.] [visited on 2022-03-14]. Available from: `https://pillow.readthedocs.io/en/stable/reference/Image.html`.

22. *ImageDraw Module* [online]. [N.d.] [visited on 2022-03-14]. Available from: `https://pillow.readthedocs.io/en/stable/reference/ImageDraw.html`.

23. *ImageFont Module* [online]. [N.d.] [visited on 2022-03-14]. Available from: `https://pillow.readthedocs.io/en/stable/reference/ImageFont.html`.

24. VIRTANEN, Pauli; GOMMERS, Ralf; OLIPHANT, Travis E.; HABERLAND, Matt; ET AL. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. 2020, vol. 17, pp. 261–272. Available from DOI: `10.1038/s41592-019-0686-2`.

25. *Discrete Statistical Distributions* [online]. [N.d.] [visited on 2022-03-24]. Available from: `https://docs.scipy.org/doc/scipy/tutorial/stats/discrete.html`.

26. *Continuous Statistical Distributions* [online]. [N.d.] [visited on 2022-03-24]. Available from: `https://docs.scipy.org/doc/scipy/tutorial/stats/continuous.html`.

27. *scipy.stats.chi2* [online]. [N.d.] [visited on 2022-03-24]. Available from: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2.html`.

28. *Collections — Container datatypes* [online]. [N.d.] [visited on 2022-03-27]. Available from: https://docs.python.org/3/library/collections.html.

29. *Web Content Accessibility Guidelines (WCAG) 2.0* [online] [visited on 2022-03-28]. Available from: https://www.w3.org/TR/2008/REC-WCAG20-20081211/.

30. Hilbert curve. In: [online]. [N.d.] [visited on 2022-04-12]. Available from: https://en.wikipedia.org/wiki/Hilbert_curve.

31. SKILLING, John. Programming the Hilbert curve. *AIP Conference Proceedings*. 2004, vol. 707, no. 1, pp. 381–387. Available from DOI: 10.1063/1.1751381.

32. *Ubuntu* [online] [visited on 2022-04-11]. Available from: https://ubuntu.com/.

33. *OpenSSL – Cryptography and SSL/TLS Toolkit* [online] [visited on 2022-04-11]. Available from: https://www.openssl.org/.

34. *Hard drives and partitions* [online]. 2022-03-16 [visited on 2022-04-18]. Available from: https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/hard-drives-and-partitions?view=windows-10.

35. *BIOS/MBR-based hard drive partitions* [online]. 2021-11-30 [visited on 2022-04-18]. Available from: https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/configure-biosmbr-based-hard-drive-partitions?view=windows-10.

36. METZ, Joachim. *BitLocker Drive Encryption (BDE) format specification* [online] [visited on 2022-04-18]. Available from: https://github.com/libyal/libbde/blob/main/documentation/BitLocker%5C%20Drive%5C%20Encryption%5C%20(BDE)%5C%20format.asciidoc.

37. *Frequently Asked Questions Cryptsetup/LUKS* [online] [visited on 2022-04-18]. Available from: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/FrequentlyAskedQuestions.