Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Watermarking Techniques applied to Data Flooding against Ransomware

Relatore:
Chiar.mo Prof.
Giallorenzo Saverio

Presentata da:
Sami Pietro

IV Sessione
Anno Accademico 2023-2024

Contents

Chapter 1

Introduction

Nowadays, it is impossible to think that a company, a hospital, a government, or a private citizen can live without storing data in a computer and sharing it through the internet. That is why cyber-attacks are becoming more and more dangerous and frequent. Several examples can be found during the COVID-19 pandemic [11], where cyber-criminals, exploiting fear and confusion, deployed ransomware in Brno University Hospital in the Czech Republic, forcing the hospital to shut down its IT systems and cancel surgeries.
In this thesis, we introduce a new approach, to refine a contrast technique of Data Flooding against Ransomware (DFaR), which exploits the full potential of dynamic honeypots to detect and mitigate ransomware attacks by flooding a specific file location with decoy files. Our contribution is to improve this approach by adding digital watermarking techniques to the flooding algorithm. Digital watermarking is a set of techniques that allows hiding a message inside a file. Today, it is widely used in several fields, such as copyright protection and content authentication. In our case, digital watermarking will be added to the Random flooding process, which floods the path with randomly generated decoy files, allowing them to be recognized during the restoration phase.

Chapter 2

Background

This section provides a general background to understand the development of this thesis, i.e., the main themes upon which this thesis revolves.

2.1 Ransomware

What is Ransomware? This name refers to a family of malware that has a common objective: *to make someone pay a ransom to regain control of some resources.*
In 1989, the first documented ransomware appeared, known as the AIDS Trojan. It was a piece of malware that hid the files on the hard drive and encrypted only their names, a false message would appear informing the user of the expiration date of certain software. In the following decades, ransomware attacks proliferated, coming up with algorithms that can bring corporations, hospitals, and even governments to their knees.
Although there are a very large number of ransomware types, we can group them into three main categories: locker, crypto, and scareware [1].

- **Locker**: By encrypting specific and important files, these ransomware types can deny basic computer functionality (e.g., computer screen and/or keyboard).
- **Crypto**: The objective of this group of ransomware is the user's sensitive files. The idea behind this strategy is quite simple: encrypt user data using irreversible encryption techniques (e.g., AES and RSA). The only solution left to the user is to pay the ransom and get the decryption key. For this family of software, there are three types of encryption schemes:
  - Symmetric: Uses a symmetric encryption scheme, and the key is embedded in the ransomware. For this reason, this type of crypto ransomware

is vulnerable to reverse engineering.

  - Asymmetric: Uses an asymmetric encryption scheme, where the private key is not embedded in the ransomware. However, this solution is slower compared to the symmetric one.
  - Hybrid: This approach uses both symmetric and asymmetric schemes. First, the ransomware creates a symmetric key to encrypt the victim's files. When it finishes the attack, a public-private key pair is generated by a server. The public key is then used to encrypt the symmetric key.

- **Scareware**: This form of ransomware aims to trick the user into believing that they are required to download a software. Instead of encrypting files, this malware exploits the user's fear.

In recent years, there has been significant progress in the fight against ransomware. The current state-of-the-art revolves around various detection techniques, such as: Honeypots, Network traffic analysis, and Machine learning-based approaches. Prevention techniques mostly rely on: Access control, Data and key backup, Hardware-based solutions.

However, a major problem nowadays, is that ransomware can be easily created and used (see Ransomware as a Service [6]). This phenomenon has led to an increasing variety of behaviors in such malware, making it more difficult to find a solution that can be effective in all cases. Traditional static analysis techniques are not enough to detect ransomware and, since it is not enough to catalogue and study their behavioral patterns, the need to create a robust method that can be efficient in prevention, detection, and, most importantly, mitigation, is more necessary than ever.
In the next section, we introduce a possible solution to this problem: **Data Flooding Against Ransomware** [2].

2.2 Data Flooding Against Ransomware

Data Flooding Against Ransomware (DFaR) is a technique that offers robust protection against ransomware, providing detection, mitigation, and restoration services. The core idea of this technique is a **dynamic honeypot approach**, which consists of creating decoy files (honeypots) to detect and mitigate ransomware activities. A honeypot is usually a dummy element (e.g., file) deployed as an easy-to-access computer resource.
This solution could simultaneously detect activities, and also slow down the ongoing attack of ransomware.
Why dynamic? As it is explained by the authors of the approach [2]:

> instead of using static files and incurring in the related trapsurface limitations, our intuition is to adopt a dynamic approach, where detection works by monitoring the activity of processes and by generating "floods" of honeypot files. If the process under inspection modifies the honeypot files - refined instantiations can analyse the patterns of data transformation to minimise false positives - we have strong evidence that it is some malware trying to lock the files of the user.

The strength of this approach is that it gives more importance to sensitive locations, where it can achieve two key benefits:

- Resource Contention: The ransomware is slowed down by flood that generates numerous disk operations.
- Moving Target Defense: By significantly increasing the attack surface, the ransomware will waste time, rising the possibility of preserving the user's data.

Below, we explain the main phases of data flooding against ransomware: detection, mitigation, and restoration. The last two phases are currently implemented in a suite of programs written in Java, under the umbrella of the **Ranflood project**.

2.2.1 Detection

The Detection phase offers two possibilities, distinguished by how target locations for decoy files are handled.

- Static mode: The user defines some target locations, where the detector performs "mini-floods" consisting of the generation of sets of random files in the target location. These files will be monitored, reporting any suspicious activity.

- Dynamic mode: The detector is triggered by a process that "patrols" the system.

2.3 Ranflood

Ranflood is an open-source software written in Java that offers an implementation of the mitigation phase following the DFaR principles. This tool follows a *client-daemon* architecture: a deamon is always running in the background, while the client is controlled by the user.

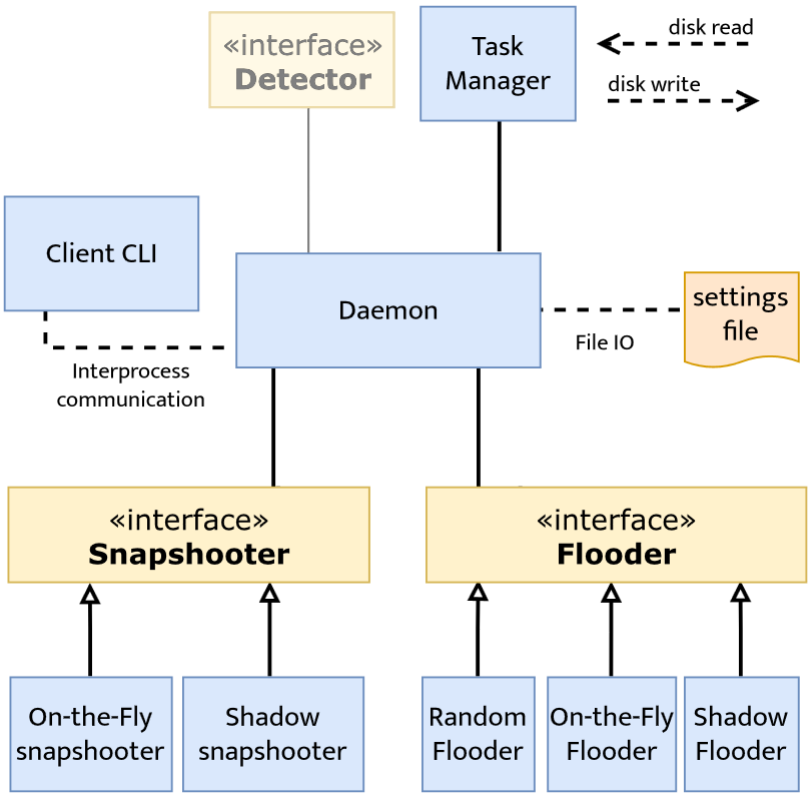

Figure 2.1: Model of Ranflood's Architecture (from [2], Figure 3).

2.3.1 Mitigation

To mitigate the ongoing action of the ransomware, Ranflood generates a multitude of decoy files, making it harder for the ransomware to distinguish them from the originals. Ranflood can performs three data flooding strategies: **Random, On-The-Fly, and Shadow**.

Random flooding

Random flooding is the most straightforward strategy that Ranflood can perform. Basically, Ranflood generates a large number of files, filled with random content but with headers corresponding to the generated file's extension. This approach requires minimal setup and is effective if it meets three conditions:

1. It generates files, using the most common file extensions (e.g., pdf, jpeg, etc.) that are likely targeted by the ransomware.
2. The generated files are difficult to discriminate from real files. Their synthetic nature is covered through some techniques, such as avoid to reuse the same sequences over and over or having file headers not matching the standard format of their related extension.
3. In a short timeframe, it produces a large number of files.

Algorithm 1 Random Data Flooding

```
Require: path, minSize, maxSize
 1: FILE_EXT ← [".doc", ".pdf", ".xls", ".jpg", ".mp4", ...]
 2: while keepFlooding do
 3:     f_size ← randomInt(minSize, maxSize)
 4:     cnt ← newByteArray(f_size)
 5:     ext ← rndSelect(FILE_EXT)
 6:     append(cnt, getHeader(ext))
 7:     seed ← random64Seed()
 8:     for i ← 0 to capacity(cnt)/64 do
 9:         seed ← seed ⊕ (seed ≪ 13)
10:         seed ← seed ⊕ (seed ≫ 7)
11:         seed ← seed ⊕ (seed ≪ 17)
12:         append(cnt, seed)
13:     end for
14:     if capacity(cnt) > 0 then
15:         r ← newByteArray(capacity(cnt))
16:         r ← fillWithRandomBytes(r)
17:         append(cnt, r)
18:     end if
19:     writeFile(rndFilePath(path, ext), cnt)
20: end while
```

All the requirements are met by the *Algorithm 1*, which achieves fast randomness thanks to the Xor-shift routine at lines 7-13, and attaches realistic headers based on file extensions.

On-The-Fly

The On-The-Fly strategy is a copy-based solution that, instead of flooding with random files, generates copies of the actual files found at flooding locations. This method results into a simple way to increase the probability of preserving user data: if more copies of the same file are created, some may escape the ransomware. The Ranflood's team has developed a "snapshotting phase" which is in charge of saving a list of the valid files, avoiding replication of already encrypted ones. The snapshooting procedure saves a digest of a file, that will be later used to validate it during the flooding phase.
It is important to note that the existence of a list entails the risk that this will be encrypted in turn, losing the possibility to apply the method, unless some offsite methods are used to preserve and retrieve the list.

Shadow

The Shadow strategy differs from the On-The-Fly on in how snapshotting works. In this case, snapshots save the entire content of files, acting as a backup of user data. Even in this case:

> since the Shadow snapshooting phase follows the traditional process of backup systems, it also suffers the same, known tradeoffs of local, on-site, and remote backup storage/retrieval.

2.3.2 Restoration

The objective of this phase is to restore the system of the user as it was before the attack. Based on the type of flooding, we can distinguish two cases:

- For the Random Flooding, we want to get rid of all the generated files.
- For the copy-based flooding strategies (On-The-Fly, Shadow), we also want to restore the user's files using, if necessary, the decoy files.

To achieve this goal, we introduce a new tool called, **FileChecker**. For the Random Flooding, it deletes all the files that are not in the snapshot's list. For the copy-based flooding strategies, it restores the user's files using the decoy files, checking the integrity of these.

2.4 Watermarking

By the definition from Mohanty et al. [10]:

> *"Watermarking is the process that embeds data called a watermark, tag, or label into a multimedia object such that the watermark can be detected or extracted later to make an assertion about the object."*

The origin of information concealment techniques dates back to the 13th century AD in Italy. Initially, watermarking emerged as a method for paper manufacturers' identification. In fact, the term probably comes from the resemblance of these identifier marks to damp spots.
Digital watermarking began to spread in the late 1980s, as major companies started to grow interest in robust watermarking systems for the protection of copyright in multimedia products (e.g., copy protection of video on DVD disks). Today, watermarking has a wide range of applications, for example [4]:

- Broadcast Monitoring: Identifying where and when broadcasted content is transmitted
- Owner Identification: Type of embedding used to identify the ownership of a product
- Transaction Tracking: Keeping track of products obtained in a legal manner but distributed illegally
- Content Authentication: Digital signatures that can be checked to verify if a product has been tampered with
- Copy Control: Preventing the illegal creation of copies of legally distributed products

From now on, we refer to a specific multimedia object as a *Work*. The original and not altered Work, such as a image is referred as *Cover Work*, which hides the watermark. We explain the watermark in the images' domain, due to its simplicity and to the presence of a large number of proposals in this field.

2.4.1 Watermarking System

It's important to define principal **properties** ([4], Chapter 2.3) that a Watermarking System has. In particular, effectiveness, fidelity, and payload are linked to the embedding process; while blind or informed detection, false positive behavior, and robustness deal with the detection process. Security and the use of keys are also important.

Properties

- **Embedding Effectiveness**: The probability that the Embedder will successfully embed a watermark in a randomly selected Work.
- **Fidelity**: The perceptual similarity between the unwatermarked and watermarked Works.
- **Data Payload**: The amount of information that can be carried in a watermark.
- **Blind or Informed Detection**:
  - Detectors that require access to the original, unwatermarked Work are referred to as *informed detectors*, and systems using informed detection are often called private watermarking systems.
  - Conversely, detectors that do not require any information related to the original are referred to as *blind detectors*, and systems using blind detection are called public watermarking systems.
- **False Positive Rate**: The frequency with which watermarks are falsely detected in unwatermarked content.
- **Robustness**: The ability of the watermark to survive normal processing of content.
- **Security**: The ability of the watermark to resist hostile attacks. Some examples are: unauthorized removal, unauthorized embedding and unauthorized detection.
- **Cost**: The computational cost of the Embedder and Detector. The two principal issues of concern are:
  - The speed with which embedding and detection must be performed.
  - The number of Embedders and Detectors that must be deployed.

Architecture

Generally, we define the main components of a watermarking scheme as follows:

- **The Watermark**: Which is the message to be embedded.
- **A Key**: Used to embed and detect the watermark, it will be likely to be secret.
- **A Cover Work**: The original Work that will hide the watermark.
- A **Watermark Embedder**, which includes:
  - **The Encoder**: an insertion algorithm, which embeds the watermark
- A **Watermark Detector**, which includes:
  - The **Extractor** and the **Comparator**: respectively, algorithms for the extraction and verification of the embedded watermark.

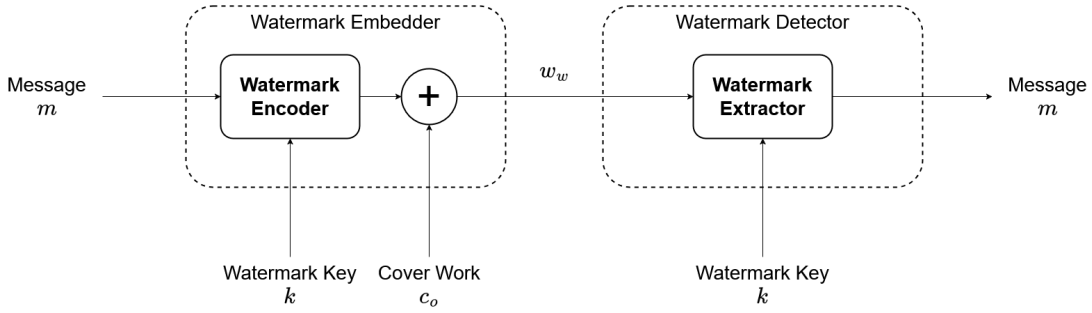

Figure 2.2: Simple Watermarking System with a blind detector.

The **embedding process** is performed by the Watermark Embedder. A message m is encoded to match the Cover Work Domain, using a Watermark Key k, the Watermark Encoder determines how the message will be adapted into the Cover Work. Typically, the key generates a pattern w_r of the same size as the Cover Work. For instance, in the case of images, this process could involve generating a matrix of pixels randomly based on the watermark key. In the final step of the embedding process, the Watermark is added to the Cover Work, resulting in the Watermarked Work w_w.

The **detection process** is performed by the Watermark Detector. Since it knows the Watermark Key, the process can use it to determine whether or not a watermark is present and extract the message.

2.4.2 Geometric Models of Watermarking

It is useful to see Watermarking System from a different perspective, in particular, we can interpret a watermarking system geometrically ([4], chapter 3.4). Imagine a high-dimensional space in which each point corresponds to one Work: we refer to this space as **Media Space**.
Examples: For a 256x256 grayscale image, there are 65,536 dimensions (one for each pixel), for a 5-second mono audio clip, sampled at 44,100Hz, there are 220,500 dimensions (one for each sample).

Analysing a Watermarking System in this way helps us to understand how the recognition of the watermark works. The detection process relies on calculating the *correlation value* between the Watermarked Work and secret pattern generated by the key (for example using *linear correlation* [16]). To better understand this concept, we describe the embedding and detection process in the Media Space.

Assume that we want to embed a message m in a 2D image i with a key k, in Media space, there exists a region of all unwatermarked Works called **Distribution of unwatermarked Works**, that includes the original starting image i. After the embedding process, the watermarked work is in a different region of the Media Space. That said, it is important to ensure that the watermarked image is similar to the original one, and must be located in a **Region of Acceptable Fidelity**, which is a area in Media Space where all Works are perceptually similar to the original one.

The **Detection Region** represents the set of all Works in the Media Space, which with a given message m and key k, will be detected as watermarked.

> The detection region is often defined by a threshold τ_{lc} on a measure of the similarity between the detector's input and a pattern that encodes m. We refer to this measure of similarity as a *detection measure*. For linear correlation, cw_r/N is equal to the product of their Euclidean lengths and the cosine of the angle between them, divided by N. Because the Euclidean length of w_r is constant, this measure is equivalent to finding the orthogonal projection of the N-vector c onto the N-vector w_r. The set of all points for which this value is greater than τ_{lc} is just the set of all points on one side of a plane perpendicular to w_r.

It is important to note that there is not only linear correlation for the correlation calculation. In this thesis, we also use the *normalized correlation* and the *correlation coefficient*. Each of these metrics has its own advantages and disadvantages, and it is important to choose the right one based on the application.

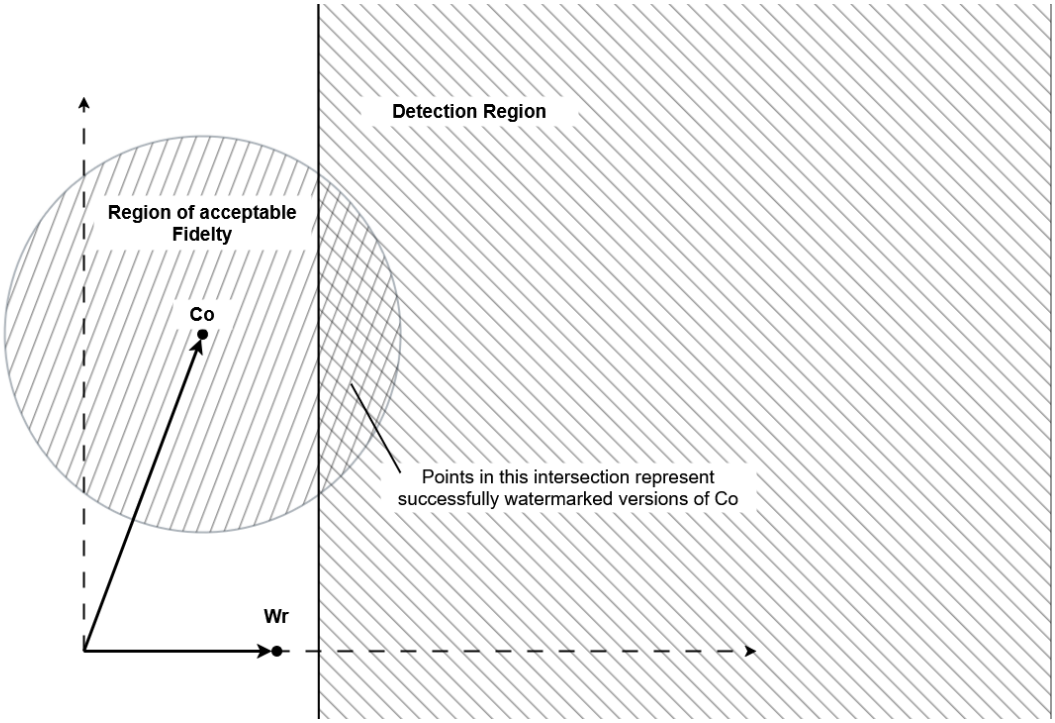

Figure 2.3: The region of acceptable fidelity and the detection regiom in media space ([4], figure 3.13).

2.4.3 Transform Domain

We can perform embedding and detection in different domains rather than the spatial domain. For example, we can transform an image from the spatial domain (the pixel values) to the frequency domain, using the Discrete Cosine Transform (DCT) [15]. There are many reasons to use frequency domain watermarking [8], [5], we can easily achieve robustness against some attacks, such as compression, and we can make the watermark less visible to the human eye.

The process follows this equation [8] [14]:

$$F(u,v) = C^{(N)} f(x,y) (C^{(N)})^T \tag{2.1}$$

when $M = N$, where M is the number of rows, and N is the number of columns of

the matrix.

$$C^{(N)} = \begin{bmatrix} \sqrt{\frac{1}{N}} & [& 1 & 1 & \dots & 1] \\ \sqrt{\frac{2}{N}} & [& \cos\left(\frac{\pi}{2N}\right) & \cos\left(\frac{3\pi}{2N}\right) & \dots & \cos\left(\frac{(2N-1)\pi}{2N}\right)] \\ \sqrt{\frac{2}{N}} & [& \cos\left(\frac{\pi}{2N}\right) & \cos\left(\frac{6\pi}{2N}\right) & \dots & \cos\left(\frac{(2N-1)\pi}{2N}\right)] \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ \sqrt{\frac{2}{N}} & [& \cos\left(\frac{(N-1)\pi}{2N}\right) & \cos\left(\frac{(N-1)3\pi}{2N}\right) & \dots & \cos\left(\frac{(N-1)(2N-1)\pi}{2N}\right)] \end{bmatrix}$$

1. $f(x, y)$ is a digital image matrix of $M \times N$ and the spatial sampled value, that is the pixel value of the point (x, y), which is the coordinates of the input 8×8 pixels.

2. $F(x, y)$ is the frequency domain sampled value, and the coordinates of the output 8×8 transform result.

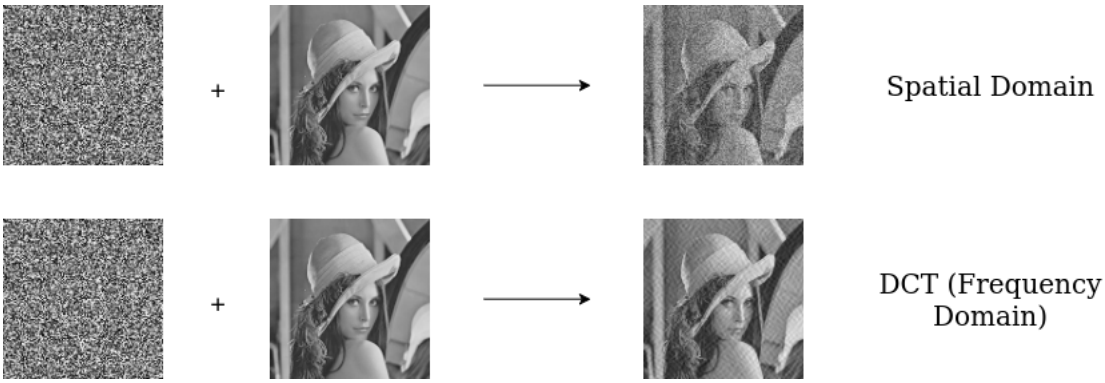

Figure 2.4: Simple embedding examples of a real image, the first one is in the spatial domain, the second one is in the frequency domain, as we can see the watermark is less visible in the second image.

Chapter 3

Our Contribution

After taking a look at the main themes to understand this thesis, we now introduce our contribution to Ranflood. An important improvement mentioned by the authors of this software, is the possibility of using digital fingerprinting to mark the flooding files [2]. By doing so, we can reconstruct the list of files that were generated during the flooding, and avoid saving any information on the fingerprinting process because this hidden "message" is embedded in the files themselves. Without the need to save the list of files, we can also avoid the risk of it being encrypted by the ransomware.

Using watermark techniques, we can create a system that meets our needs. By embedding a watermark in the flooding files, we can:

- Recognize random decoy files during the restoration phase. If a Watermark Detector can detect the watermark, then the file was generated by Ranflood.
- Avoid saving any information on the fingerprinting process.

With a good watermarking system, all these features can be achieved without giving any advantage to the attacker (the ransomware), because the watermark is hidden in the files themselves and only the user is in possession of the key.

3.1 Improving Random Flooding

For this thesis, we focus on the **Random Flooding** strategy. Since Ranflood creates random content during the flooding, it is not necessary to study watermarking methods for specific file types, since all the generated files share a uniform structure and do not need to match any specific format.

Overall this system must also be fast, since Ranflood cannot waste too much time on the watermarking process, otherwise it would decrease Ranflood efficiency

in contrasting ransomare. For this flooding strategy, we integrate a watermarking system into Ranflood that can generate a unique key for each flooding, used to embed a watermark in the randomly generated content of each file.
The system should also detect if a file is a decoy, by identifying the watermark and remove each watermarked file.
The first part is related to the embedding process and is performed during the **mitigation phase**, while the last one is related to the detection process and is carried out during the **restoration phase**.

3.2 Proposed Watermarking System

First of all, we must define the properties of the watermarking system that we want to create:

- **Embedding Effectiveness**: All the files we produce during flooding must be recognized, otherwise it is not worth using the watermark system.
- **Fidelity**: The similarity between the original un-watermarked file and the watermarked one does not matter. Since we are using random files, absolutely useless to the user, we do not need to work on the similarity between the two.
- **Data Payload**: For this type of flooding, we are not concerned about the amount of information we have to embed, we just need to know if a file is generated by Ranflood or not. It is sufficient to know if a file is a decoy or not, without the need of extracting the watermark.
- **Blind Detector / Informed Detector**: We use a blind detector, we do not want to store all the generated files. It would not make sense to store a list of decoy files, since we are trying to avoid saving the user's file list.
- **Robustness**: The attacker could tamper with the Ranflood files. We need to make sure that the watermark is still detectable after some modifications. It is important to underline that, if the ransomware encrypts the files, the watermark will not be detectable but, it is not a problem, because the malware has wasted time encrypting useless files.
- **Security**: Among all attacks that an attacker could perform, we must focus on the *unauthorized detection*. If the attacker can detect which files are generated by Ranflood, it could avoid encrypting, making useless the Random Flooding strategy. We want to make sure that the most efficient solution for the attacker is encrypting all files, because detecting the watermark would be too expensive.

Having clear properties to respect, we can now define the architecture of our watermarking system.

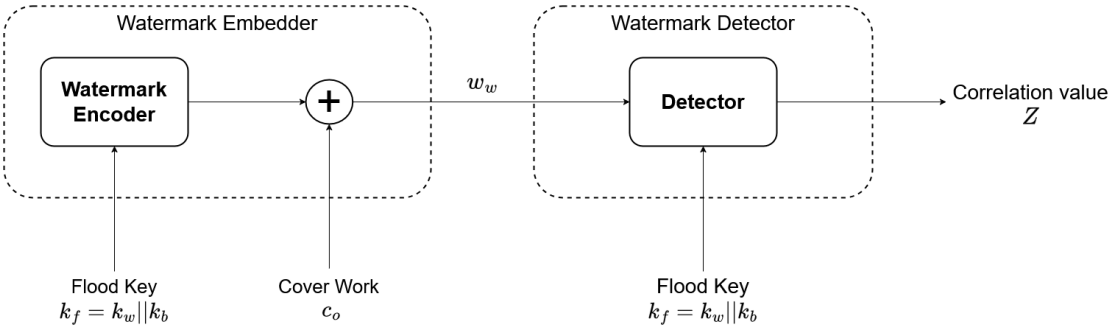

Figure 3.1: Proposed Watermarking System.

3.2.1 Embedding

The embedding process is performed during the flooding. First, the Embedder generates a key k_f called **Flood Key**, that will be used to embed the watermark in the files. This key is the concatenation of the **Watermark Key** k_w, the **Block Key** k_o.

$$k_f = k_w \parallel k_b \tag{3.1}$$

The Flood Key is given to the user, and not saved in the system, in order to avoid the risk of being found by the attacker.
The Watermark Key k_w determines the random pattern that is added to the Cover Work, in particular, it is the seed of a pseudo random number generator, that creates a matrix of 8-bit integer values (that could be interpreted as pixels) with a size of $N \times N$, called the Watermark w. The Block Key k_o will be described later.

After creating the key, the Random Flooder generates random content as we have seen in *Algorithm 1.* In this case, we adapt to work as a fixed-size matrix $R \times M$, where R is a random integer that changes for each file and M is another parameter called *Default Embedding Surface.* We can interpret this matrix as the Cover Work c_o.
At this point, the Watermark Encoder could add the watermark in the Cover Work according to certain parameters: **Watermark intensity, Embedding Type, Use Random Blocks, Watermark Size, and Default Embedding Surface**.

Watermark intensity A scalar value α that determines the intensity of the Watermark w embedded in a Cover Work c_o that produce a Watermarked Work

w_w.

$$\begin{aligned} w_w &= c_o + \alpha \cdot w \\ 0 &< \alpha < 1 \end{aligned} \tag{3.2}$$

Embedding Type The watermark could be added in the spatial domain or in the **frequency domain**, using the **DCT**. For the DCT-Embedding, the Work and the watermark are converted using the DCT, and the watermark is added to the Work in the frequency domain, then the IDCT (inverse DCT) is applied to the result.

Algorithm 2 Embedding Process

```
function EMBED(coverWork, watermark, e_params)
    if e_params.useDCT then
        c_o ← DCT(coverWork)
        w ← DCT(watermark)
        c_o ← c_o + e_params.wtIntesity · w
        wtWork ← IDCT(c_o)
    else
        wtWork ← coverWork + e_params.wtIntesity · watermark
    end if
    return wtWork
end function
```

In *Algorithm 2*, *e_params* (embedding parameters), a dictionary containing the embedding settings, we store: the Watermark Size, the Watermark Intensity, the Default Embedding Surface, the Embedding Type, and the Use of Random Blocks Embedding.

Use Random Blocks Embedding The watermark could be spread in the Cover Work only in some sub-blocks [5] [12], making more difficult for the attacker to detect it. This operation follows the concept that the watermark is added only in certain blocks and only who knows how the watermark is spread can detect it. To achieve this goal, the watermark and the Cover Work are divided in sub-blocks of size 8×8, the Block Key k_o is used to select random blocks (via a pseudo random number generator) in the Cover Work.

Algorithm 3 Embedding Process with Random Blocks

```
1: function EMBEDBLOCKS(coverWork, watermark, e_params, K_b)
2:     if not e_params.useBlocks then
3:         return embed(coverWork, watermark, e_params)
4:     end if
5:     coBlocks ← divideInBlocks(coverWork)
6:     wtBlocks ← divideInBlocks(watermark)
7:     random.seed(K_b)
8:     for i ∈ wtBlocks do
9:         n ← random.randInt(0, len(CoverWorkBlocks))
10:        coBlocks[n] ← embed(coBlocks[n], wtBlocks[i], e_params)
11:    end for
12:    return coBlocks
13: end function
```

In *Algorithm 3*, the *random.seed* function initializes a pseudo-random number generator, that will be used to select the blocks in the Cover Work where the watermark will be added.

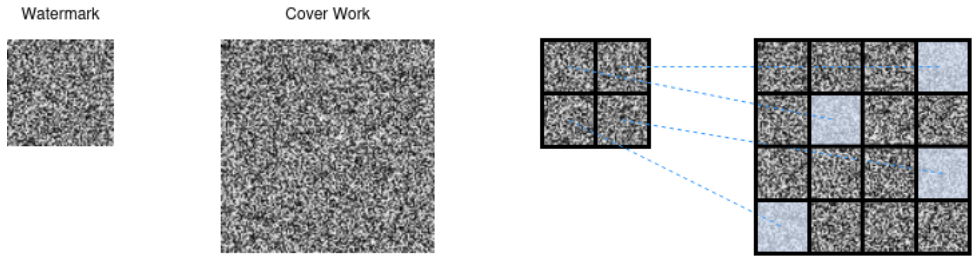

Figure 3.2: Random Blocks Embedding.

Watermark Size and Default Embedding Surface The **Watermark Size** N determines the shape of the matrix generated at the beginning of the embedding process. The **Default Embedding Surface** M is the size of the matrix where the watermark is added, this measure has to be greater or equal than the watermark size.
In case of non-Random Blocks Emebedding, the watermark is added only in a sub-matrix of $N \times N$ of the Cover Work. For Random Block Embedding the watermark could be spread in all the Cover Work through the blocks.

In the last step of the embedding, the matrix is converted into a string of byte, the following steps follow the instruction in the *Algorithm 1*, a signature is added, and the files is saved into the disk with a random extension and with its specific header attached.

3.2.2 Detection

The detection process if performed during the restoration phase, the Flood Key (provided by the user) is used to determine the watermark, and the sequence of blocks in case of blocks embedding. The process follows these steps:

1. The file is read from the disk and converted into a matrix of bytes.
2. The signature is removed, the remaining content is converted in a matrix of M columns (Default Embedding Surface) and R rows (different from each file).
3. The matrix and the watermark are compared through a correlation measure. If the correlation value is greater than a **threshold**, the file contains the watermark, so it was generated by Ranflood. Otherwise, the file does not contain the watermark, and could be a user file (also an encrypted one). The value of the threshold can be calculated before the flooding, after Ranflood generates the Flood Key, it could perform a mini-flooding and calculate the average correlation of these files.

In the cleaning process, all files that contain the watermark (i.e., all the files with a correlation value greater than the threshold) are removed from the disk.

Correlation Measure

As we mentioned before, there are different correlation measures that can be used to detect the watermark.

Linear Correlation This is the simplest correlation measure, which consists in calculating the average product of two vectors, in our case, two matrices A and B with sizes $M \times N$.

$$\mathcal{Z}_{lc}(\mathbf{A}, \mathbf{B}) = \frac{1}{MN} \sum_i \sum_j \mathbf{A}[i,j] \cdot \mathbf{B}[i,j] \tag{3.3}$$

In media space, Comparing $\mathcal{Z}_{lc}$ against a threshold leads to detection region with planar boundary. But, as enlighted by Cox et al. in this book [4]

> One of the problems with linear correlation is that the detection values are highly dependent on the magnitudes of vectors extracted from Works.

Normalized Correlation The problems of linear correlation can be solved by normalizing the matrices before computing the correlation.

$$\begin{aligned} \tilde{A} &= \frac{A}{|A|} \\ \tilde{B} &= \frac{A}{|A|} \\ \mathcal{Z}_{nc}(A, B) &= \sum_i \sum_j \tilde{A}[i][j] \cdot \tilde{B}[i][j] \end{aligned} \tag{3.4}$$

Algorithm 4 Random Data Flooding with Watermarking

```
Require: path, floodKey, e_params
 1: FILE_EXT ← [".doc", ".pdf", ".xls", ".jpg", ".mp4", ...]
 2: M ← e_params.defaultEmbeddingSurface
 3: N ← e_params.watermarkSize
 4: wt ← generateWatermark(N, floodKey[0-63])
 5: while (keepFlooding) do
 6:     R ←randomInt(N, M)
 7:     cnt ← newByteArray(M × R)
 8:     ext ← rndSelect(FILE_EXT)
 9:     seed ← random64Seed()
10:     for i ← 0 to capacity(cnt)/64 do
11:         seed ← seed ⊕ (seed ≪ 13)
12:         seed ← seed ⊕ (seed ≫ 7)
13:         seed ← seed ⊕ (seed ≪ 17)
14:         append(cnt, seed)
15:     end for
16:     if capacity(cnt) > 0 then
17:         r ← newByteArray(capacity(cnt))
18:         r ← fillWithRandomBytes(r)
19:         append(cnt, r)
20:     end if
21:     cnt ← embedBlocks(cnt, wt, e_params, floodKey[64-128])
22:     writeFile(rndFilePath(path, ext), cnt)
23: end while
```

Algorithm 4 combines the watermark embedding process just described with the Random Flooding strategy seen in the *Algorithm 1*.
The function *generateWatermark*, takes the size of the matrix, and the first 64 bits of the Flood Key, and return a $N \times N$ matrix of 8-bit integer values.

Algorithm 5 Restoration Process for Random Flooding with Watermarking

```
Require: path, floodKey, e_params, threshold, corrMeasure
 1: M ← e_params.defaultEmbeddingSurface
 2: wt ← generateWatermark(N, floodKey[0-63])
 3: for file in path do
 4:     f ←readFile(file)
 5:     f ←removeHeader(f)
 6:     cnt ← newByteArray(getSize(f))
 7:     cnt ← f
 8:     Z ← correlation(cnt, wt, floodKey, e_params, corrMeasure)
 9:     if Z > threshold then
10:        removeFile(file)
11:     end if
12: end for
```

In *Algorithm 5*, the *correlation* function is used to calculate the correlation between the watermark and the file according to *e_params* and follows the steps described in the Detection process. The *corrMeasure* could be: *Linear Correlation or Normalized Correlation*.

3.2.3 Security Analysis

Now, we analyze the security of the watermarking system we have created. In a possible attack scenario, the ransomware may not only encrypt files, but also try to detect the watermark in order to skip the encryption of the Ranflood files, making the Random Flooding strategy counterproductive. We concern about the **unauthorized detection**, which is the ability of the attacker to detect the watermark in the files without the Flood Key.

We analyse some of the simplest strategies that the attacker could use to detect the watermark:

Brute Force Attack The attacker could try to detect the watermark by trying all possible Flood Keys. If we use a 64-bit key for the Watermark Key, and a 64-bit key for the Block Key, the attacker would have to try 2^{128} keys.

Using a watermarked file as watermark The attacker could use a known file, generated by Ranflood, to detect the watermark. If our watermarking system is not robust, an attacker could use an already watermarked file as a reference, that may have the correlation value sufficiently different from a generic user file, making the detection easier. For example, if the Embedder produces watermarked files with the same size, each file will have some similarities with the others, due to the fact that the watermark is added in the same way. To avoid this attack, we can use the **Random Blocks Embedding** technique. By spreading the watermark in some random blocks, each file will have differences on how the watermark was added. Since each file has different size, the watermark will be likely added in different blocks, making each file more indistinguishable from the others.

Rebuild the Watermark Even with the use of Random Blocks Embedding, the attacker could try to rebuild the watermark.

1. The attacker takes two watermarked files, f_1 and f_2, and divide them in blocks.
2. For each block of f_1, the attacker calculates the correlation with each block of f_2.
3. The blocks where it found the highest correlation values are likely to contain the watermark.
4. The attacker can then try to rebuild the watermark by taking the blocks with the highest correlation values.

5. For each other file, the attacker can try to detect the watermark by comparing each block found in file f_1 with the blocks of the other files.

To avoid this attack, we can:

- Use a strong PRNG, like Fortuna [9], to select blocks and the watermark.
- Use a large Default Embedding Surface, to make the attacker take more time to find the blocks.
- Ensure the size of all block is small, to make the correlation values less significant.

Chapter 4

Evaluation

4.1 Implementation

In this chapter, we describe how the Watermarking System was tested. This implementation was essential to evaluate the effectiveness of the designed system. The implementation was written in Python, both because it is a language easy to use and understand, and because it offers many easy-to-access libraries, for example, Open CV for DCT or Matplotlib for the graphic visualization of results.

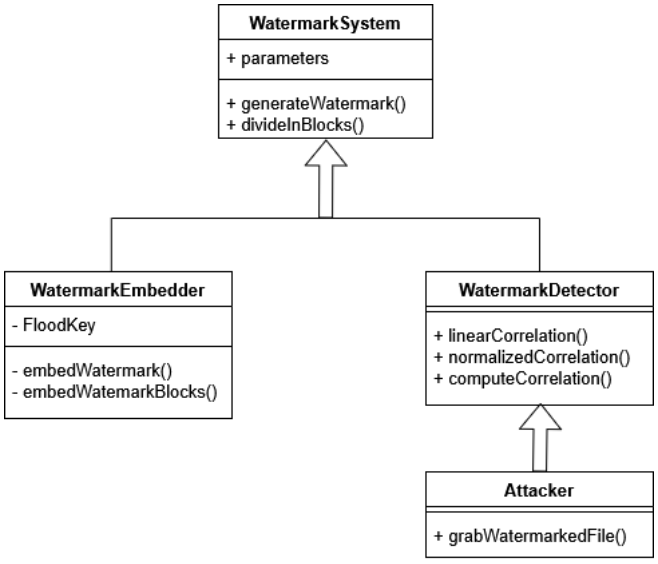

Figure 4.1: UML Diagram

We created an environment that simulates a flooding operation. For this purpose, we designed several classes: an Embedder that embed the watermark, a

Detector that detects the watermark with the key, an Attacker that tries to detect the watermark without the key, and a Simulator that simulates the Ranfood's instance.

4.2 Testing

The embedding and detection process follows the same scheme we have seen in Chapter 3. In this specific implementation, first an Embedder, a Detector and an Attacker are created. Each of these instances is initialized with the same parameters (Watermark Intensity, Embedding Type, Use Random Blocks, Watermark Size and Default Emebedding Surface). At this point, a directory is filled with 100 decoy files watermarked by the Embedder. Then, the correlations (both Linear Correlation and Normalized Correlation) with different type of files are calculated:

- **Correlation with Watermark and watermarked files** $\mathcal{Z}_{w,w}$: The correlation with the watermark is calculated for each file.
- **Correlation with watermark and random content** $\mathcal{Z}_{w,r}$: Done to see what values we expect when the correlation is calculated with random content, like encrypted files.
- **Correlation with watermark and real files** $\mathcal{Z}_{w,f}$: Which represents the correlation we expect when the watermark is calculated with user files.

Real files are taken from CIFAR-100 dataset [7], where we take only 100 randomly selected images. Even if we only use the images domain, this correlation could be calculated with any type of file. For this purpose, we have created a script that calculates: the correlation with images from CIFAR-100, the correlation with real files with common file extension (pdf, xls, mp4 ...), and the correlation with watermarked files (Figure 4.2).

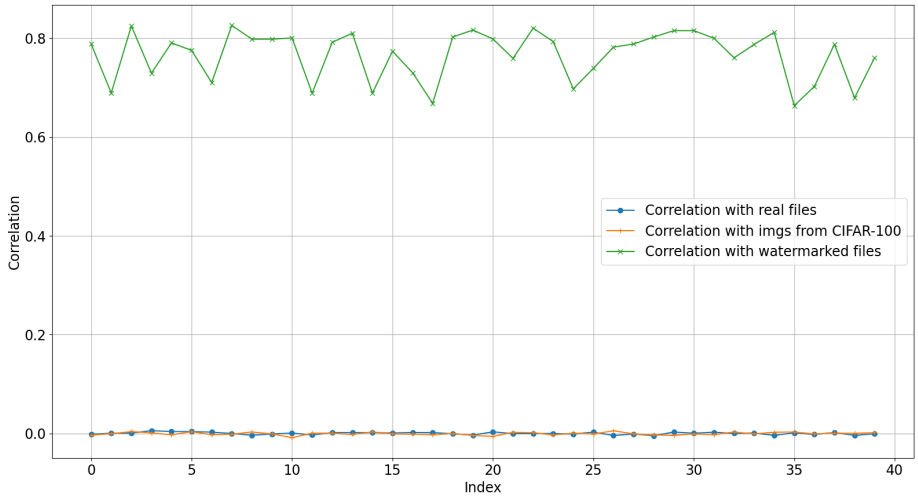

Figure 4.2: Correlation with CIFAR-100 and Real Files.

To study the behaviour of a hypothetical attacker, we calculate:

- **Correlation with a watermarked file and another one** $\mathcal{Z}_{a,w}$: This correlation is calculated between two watermarked images (3.2.3). This should simulate the correlation that an attacker would obtain with the decoy files.
- **Correlation with a watermarked file and a real file** $\mathcal{Z}_{a,f}$: This correlation represents the correlation that an attacker would obtain with a user file and the watermarked file, which is used to detect the watermark.

This process simulates an attack where, without prior knowledge of which images are watermarked, the attacker identifies the file that exhibits the highest correlation value with a batch of files (e.g., 20) and then uses it in the later stages of the attack. At the end of the test, all correlations are plotted and saved in a json file.

4.3 Results

In this section, we show the results of the tests carried out. The tests were carried out with the following embedding parameters: embedding type, use random blocks, watermark intensity, watermark size, default embedding surface, and blocks size.

The results are shown in a line chart, where the x-axis represents the file index and the y-axis represents the correlation value (Normalized Correlation). On the x-axis, the i-th file is the same for both the blue and red correlation lines. Similarly, the i-th file corresponds to both the purple and green lines. This means that the correlations are computed on the same files, but under different conditions or methods, allowing for a direct image-by-image comparison of the results.

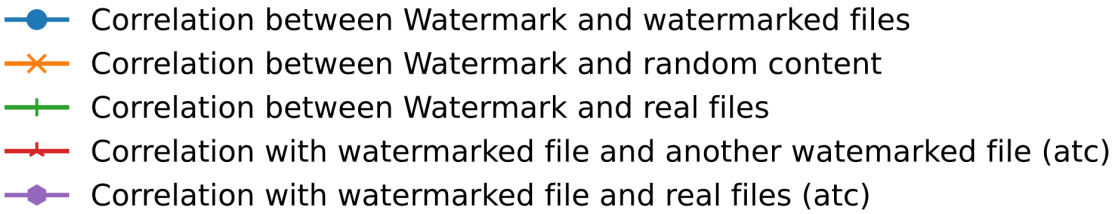

Figure 4.3: Legend.

With the simplest configuration, using the spatial domain embedding (STD), a watermark intensity of 0.9, and no random blocks, the results are as follows:

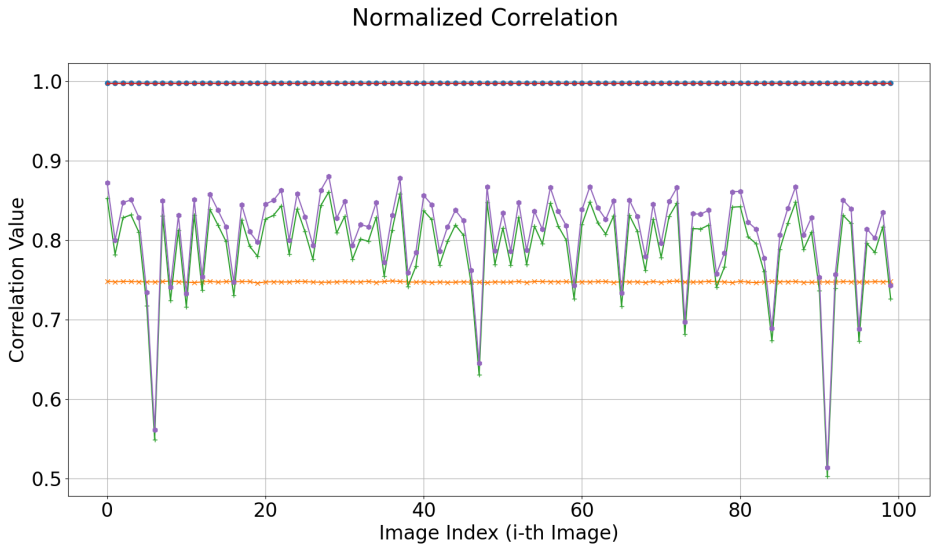

Figure 4.4: watermark intensity: 0.9, watermark size: 512, embedding type: STD.

As can be seen from the Figure 4.4, this configuration is not very effective. On the one hand, we can easily recognize which files are watermarked and which are not, the average correlation with these files is 0.99768 with a variance of $3.05540 \cdot 10^{-11}$. On the other hand, the correlation calculated by the attacker with the watermarked files is very high too (the average is 0.99735).

Trying to reduce the **Watermark intensity** to $0,1$, seems to be not effective, the average correlation calculated using the secret watermark is 0.79174. Not only the problem of high value of correlation of the attacker persists with an average value of 0.78682, but the normalized correlation does not allow us to distinguish which is the watermark and which is not (Figure 4.5).

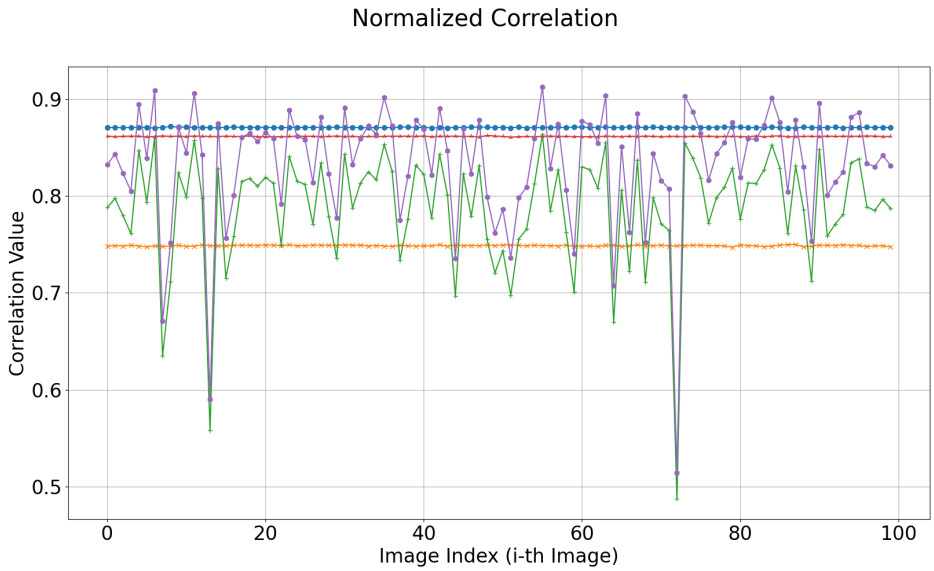

Figure 4.5: watermark intensity: 0.1, watermark size: 512, embedding type: STD.

Moving to **DCT** embedding, we can see that the results are better. The average correlation between watermarked files and the watermark is 0.11031, while the average correlation calculated by the attacker is 0.01205. This is a big improvement compared to the previous configuration, but the attacker is still able to detect the watermark because the correlation is still higher than the correlation with real files (Figure x4.6).

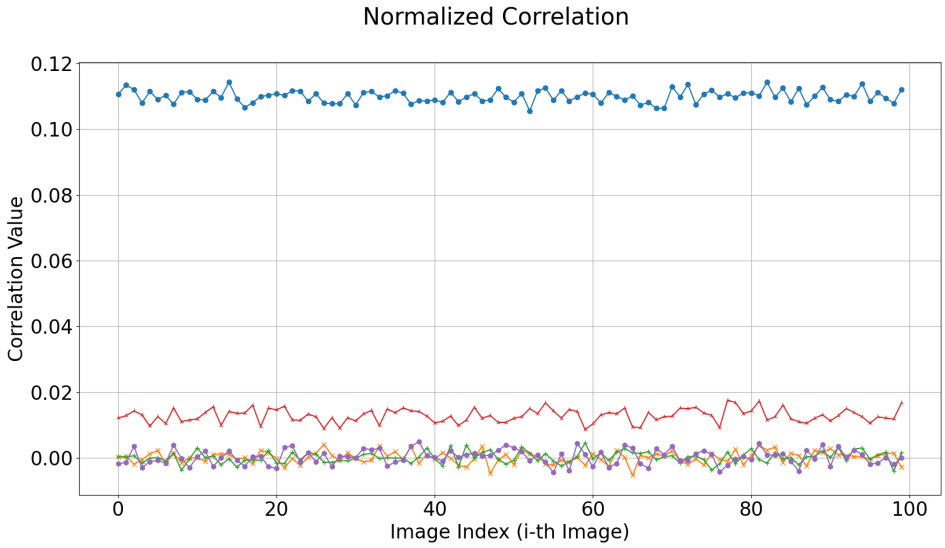

Figure 4.6: watermark intensity: 0.1, watermark size: 512, embedding type: DCT.

A notable improvement is obtained if we use **Random Blocks Embedding**. Using this technique allows us to use Watermark Size and Default Embedding Surface values different from each other (Figure 4.7).

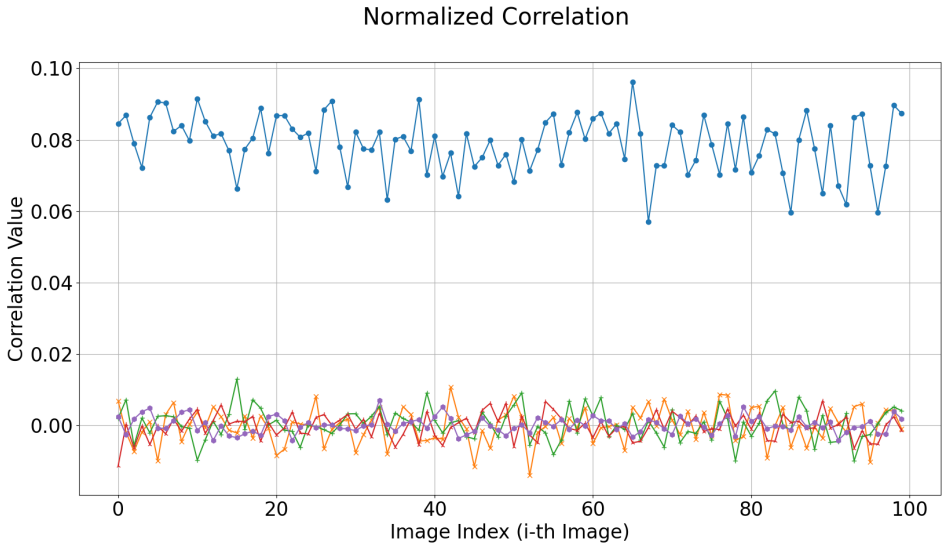

Figure 4.7: watermark intensity: 0.1, watermark size: 512, embedding type: DCT, random blocks: True, default embedding surface: 1024.

4.4 Finding the best configuration

Although we know that some parameters lead to improvements, it is not easy to find the best configuration, considering all parameter combinations. For this reason, we model our problem as an **objective function** that as to be minimized. In particular, we want to minimize the difference between *Correlation with a watermarked file and another one* and *Correlation with a watermarked file and a real file*, so that the attacker is not able to distinguish between the two. Meanwhile, we want to maximize the difference between *Correlation with Watermark and watermarked files* and **Correlation with watermark and random content and real files**, so the watermark can be easily detected.
The variables that we can change, with their boundaries, are:

- **Embedding Type**: STD, DCT
- **Use Random Blocks**: True, False
- **Watermark Intensity**: 0.1, 0.9
- **Watermark Size**: 128, 256, 512
- **Default Embedding Surface**: 512, 1024, 2048
- **Blocks Size**: 8, 16, 32

To implement this model, we have used the Python library *Scipy* [13] its module, *Scipy.optmize*, provides several functions for minimizing (or maximizing) objective functions. In particular, we have used the **dual annealing** algorithm [3]. Dual Annealing is a stochastic optimization method that can handle mixed continuous and discrete variables (with bounds).

The objective function to be minimized, is defined as follows:

$$f_o = |\mathcal{Z}_{a,w} - \mathcal{Z}_{a,f}| - |\mathcal{Z}_{w,w} - \mathcal{Z}_{w,r}| - |\mathcal{Z}_{w,w} - \mathcal{Z}_{w,f}| \tag{4.1}$$

Furthermore, we have some constraints that we have implemented in the objective function:

- Negative values are set to 0.
- During the test, we have seen that dual annealing had the tendency to increase the watermark intensity, obtaing high values of $\mathcal{Z}_{w,w}$ but also $\mathcal{Z}_{a,w}$. To avoid this, when: $|\mathcal{Z}_{a,w} - \mathcal{Z}_{a,f}| > \omega$, where ω is a threshold, the objective function returns an infinite value.

The best configuration found is:

- **Embedding Type**: DCT
- **Use Random Blocks**: True
- **Watermark Intensity**: 0.3
- **Watermark Size**: 256
- **Default Embedding Surface**: 1024
- **Blocks Size**: 8

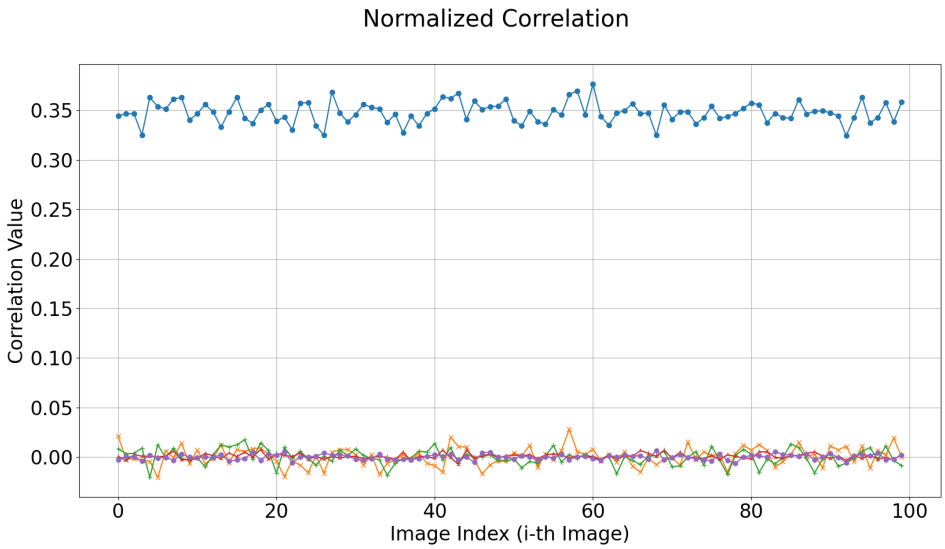

Figure 4.8: Best Configuration.

Using normalized correlation as a metric, permits to compare every correlation value in a equitable manner, because the correlation values are normalized between 0 and 1. It's important to note that the configuration found through a heuristic approach, it is not necessarily the best absolute configuration,

4.4.1 Selection of parameters

Even if we found an approximation of the best configuration, the idea is that, the selection of the parameters will depend on the specific use case. A user who wants better performance in terms of time, could choose to use a small Watermark. The core idea is that the system is flexible and can be adapted while ransomware attacks evolve (2.1).

4.5 Performance

The performance of the system is evaluated in terms of the time required to embed and detect the watermark of 100 files. These tests were carried out on a Debian 12 (bookworm) machine equipped with an Intel i3-8100 (3.600GHz), and 16GB of RAM.
To simulate the random flooding, we start the embedding process without adding the watermark to the files, a Python implementation of the Ranflood random content generator was used to be as faithful as possible to the original implementation, the Default Embedding Surface was 1024. The time required to perform a flooding without the watermark is 35.71382 seconds.

A simple watermark process (embedding type: STD, no Random Blocks embedding, and a watermark size of 1024), took the Embedder 36.6068 seconds, while the Detector took 0.90186 seconds to calculate the normalized correlation for all the decoy files. The Attacker took 2.92393 seconds.

Moving to the frequency domain embedding, the Embedder took 37.55784 seconds, the Detector took 1.38709 seconds, and the Attacker took 4.27894 seconds.

Using the best configuration found, the Random Blocks embedding parameters allow to significantly increase the time required by the Attacker, due to the fact that, at each file, the Attacker has to calculate the correlation for each block of the file. The Embedder took 86.39539 seconds, the Detector took 53.39403 seconds, and the Attacker took 220.07028 seconds.

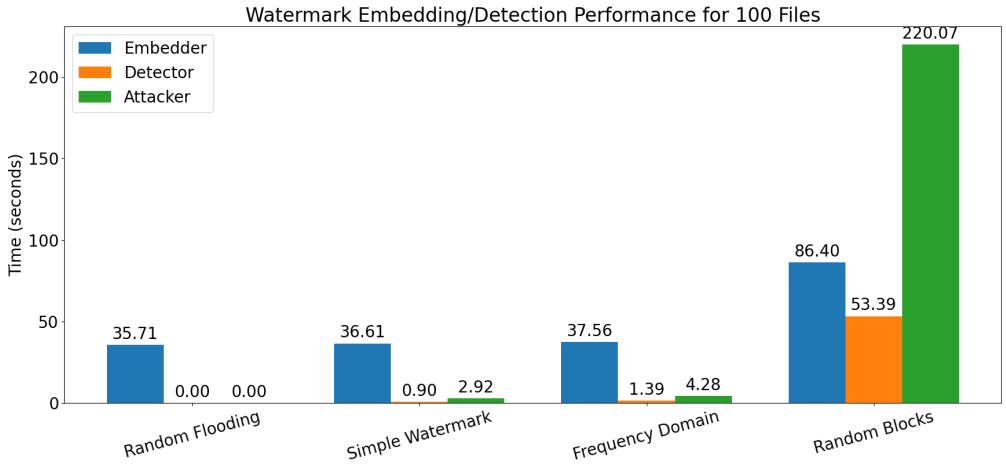

Figure 4.9: Times required for Embedder, Detector and Attacker.

Chapter 5

Conclusion

5.1 Summary

In chapter 2 of this thesis, we have analysed a major problem in today's world, ransomware. We presented an innovative solution, Ranflood, that put into practice a new approach called Data Flooding against Ransomware. Like any emerging technology, there are lots of components that need to be further developed. That is why we give an introduction of Digital Watermarking as a technique useful for improving some Ranflood strategies, like the Random Flooding, where it helps in recognizing decoy files generated by Ranflood.
In chapter 3, we described the watermarking system that we developed. The system is designed to be used in conjunction with Ranflood. In particular, it embeds a watermark in the decoy files during the mitigation, and then it can be used to recognize the decoy files during the restoration phase. We studied the behavior of a hypothetical attacker (the ransomware). In chapter 4, we presented the results of the experiments we carried out to evaluate our watermarking system, and showed how the results are affected by the parameters. We also showed the best configuration of parameters, found using the dual annealing optimization algorithm.
During the experiments, we also stored and analyzed the time takes by Embedder, the Detector and the Attacker to perform their tasks.

5.2 Future Work

Java Implementation in Ranflood The system was tested in a standalone environment using Python. The next step is to integrate the watermarking system into the suite of software that makes up Ranflood. It should not be too difficult thanks to the fact that the watermarking system has been developed around the

Random Flooding algorithm, and thanks to the followed object-oriented design, the integration consists of creating the same classes described in the watermarking system implemented in Python, but in Java. These will provide methods to interact with the already existing Ranflood classes.

Making random files more realistic During the experiments, we noticed that the correlation calculation reveals different behavior between real files and random files. For example, for the best correlation found in section 4.4, $\mathcal{Z}_{w,r}$ and $\mathcal{Z}_{a,w}$ have values respectively of -3.64212 and 4.53522, with variance of 3286.39770 and 357.99387. Meanwhile, $\mathcal{Z}_{w,f}$ and $\mathcal{Z}_{a,f}$ have values respectively of -0.0088 and -0.00228, with variance of 0.01817 and 0.00155. This behaviour is shown even if the watermark is not present. The attacker can exploit this difference to recognize decoy files, so it's important to make the random files more realistic, whether or not the watermark is applied. The random file generation process must be refined so that its statistical properties (e.g., entropy, mean, variance) closely mirror those of real files.

Exploring other watermarking techniques and attack strategies The watermarking system that we had shown in this thesis is not the only one that can be used to improve Ranflood. There are many other techniques that can be implemented [12]. The idea is to offer to the user different techniques and use them based on use cases.
The same applies to the attacker, we have only tested a simple strategy because we have no examples in literature for this kind of issue.

Improve copy-based solutions The study of watermark applied to Ranflood opens the door to improvement that can be made for the Shadow and the On-the-fly strategies. In particular, with the help of steganography, we can hide additional flooding information in the generated files, such as the path of the original copy.

Deepen the study of frequency domain embedding The results show that the combination of DCT and random blocks embedding is the most effective in our case. The combination of normalized correlation and DCT embedding allows us to obtain better results. A reasonable explanation could be that DCT produces a matrix with values greater than 255, or even negative, so the amount of information available for embedding increases, enhancing the watermark's resilience and making it easier to distinguish watermarked files from random content during detection. It would be interesting to deepen the study of the frequency domain embedding, to understand if there are other techniques that can be used to improve the watermarking system.

Bibliography

[1] Craig Beaman, Ashley Barkworth, Toluwalope David Akande, Saqib Hakak, and Muhammad Khurram Khan. Ransomware: Recent advances, analysis, challenges and future research directions. *Computers & security*, 111:102490, 2021.

[2] Davide Berardi, Saverio Giallorenzo, Andrea Melis, Simone Melloni, Loris Onori, and Marco Prandini. Data flooding against ransomware: Concepts and implementations. *Computers & Security*, 131:103295, 2023.

[3] The SciPy community. dual annealing. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html#id5.

[4] Ingemar Cox, Matthew Miller, Jeffrey Bloom, Jessica Fridrich, and Ton Kalker. *Digital Watermarking and Steganography*. Morgan Kaufmann, 2007.

[5] Ingemar J Cox, Joe Kilian, F Thomson Leighton, and Talal Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE transactions on image processing*, 6(12):1673–1687, 1997.

[6] Noël Keijzer. The new generation of ransomware: an in depth study of ransomware-as-a-service. Master's thesis, University of Twente, 2020.

[7] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[8] Haiming Li and Xiaoyun Guo. Embedding and extracting digital watermark based on dct algorithm. *Journal of Computer and Communications*, 6(11):287–298, 2018.

[9] Robert McEvoy, James Curran, Paul Cotter, and Colin Murphy. Fortuna: cryptographically secure pseudo-random number generation in software and hardware. In *2006 IET Irish Signals and Systems Conference*, pages 457–462. IET, 2006.

[10] Saraju P Mohanty. Digital watermarking: A tutorial review. *URL: http://www. csee. usf. edu/˜ smohanty/research/Reports/WMSurvey1999Mohanty. pdf*, 1999.

[11] Heba Saleous, Muhusina Ismail, Saleh H AlDaajeh, Nisha Madathil, Saed Alrabaee, Kim-Kwang Raymond Choo, and Nabeel Al-Qirim. Covid-19 pandemic and the cyberthreat landscape: Research challenges and opportunities. *Digital communications and networks*, 9(1):211–222, 2023.

[12] Arooshi Verma. Image-watermarking-using-dct. https://github.com/arooshiverma/Image-Watermarking-using-DCT.

[13] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[14] Open Source Computer Vision. Dct. https://docs.opencv.org/4.x/d2/de8/group__core__array.html#ga85aad4d668c01fbd64825f589e3696d4.

[15] Wikipedia. Discrete cosine transform. https://en.wikipedia.org/wiki/Discrete_cosine_transform, 2025.

[16] Wikipedia. Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient, 2025.