Reliable and Robust Watermarking for Data Flooding against Ransomware Random Techniques

Saverio Giallorenzo*†, Simone Melloni‡, Pietro Sami*
*Università di Bologna, Bologna, Italy †Olas Team, INRIA, Sophia Antipolis, France ‡ARPAE Emilia-Romagna, Italy

***Abstract*—Data Flooding Against Ransomware (DFaR) techniques combat ransomware through decoy files that can reveal a ransomware's activity and reduce the effectiveness and efficiency of attacks by confounding legitimate user files and competing for IO resource access of the attacked host. While effective, existing DFaR random strategies (which flood a user system with realistic yet random-content decoy files) face challenges during restoration, due to the necessity of pre-attack file lists to discriminate between proper and decoy files (the latter should be removed to restore the system to its pre-attack state). To tackle this issue, we present a watermarking-based approach that embeds imperceptible watermarks in random-content decoy files. Our technique preserves the indistinguishability of decoys from user files to attackers, while providing users with a reliable mechanism to differentiate between authentic and decoy content, obviating the need for pre-attack file lists. We present experimental evaluations that demonstrate that our watermarking technique a) imposes minimal-to-medium computational overhead (depending on user-configurable parameters) compared to existing random-content flooding methods (i.e., it is efficient when contrasting ransomware and restoring a user's system) and b) it provides strong resistance against adversarial inference attacks.**

***Index Terms*—Ransomware defence, Data flooding, Watermarking, Honeypots**

I. Introduction

Ransomware attacks have emerged as one of the most pressing cybersecurity threats of the contemporary digital landscape, representing a critical area of active research within the information security community [1]. The proliferation of sophisticated ransomware variants [2] and the emergence of Ransomware as a Service [3] have demonstrated an alarming capacity to disrupt critical infrastructure, compromise healthcare systems, and inflict substantial economic damage across both public and private sectors [4], [5]. Recent high-profile incidents targeting hospitals, municipal services, and major corporations have underscored the urgent need for innovative defensive strategies that can effectively counter these evolving threats [6]. In this work, we focus on the category of crypto ransomware, which encrypt a victims files, asking for a ransom to obtain the decryption key.

Recent years have witnessed advances in combating ransomware through state-of-the-art detection techniques including honeypots, network traffic analysis, and machine learning-based approaches, while prevention approaches focus on access control, data and key backup systems, and hardware-based solutions [1], [7], [8]. Data Flooding Against Ransomware [9] (DFaR) is a promising complement to these solutions.

Briefly, DFaR introduces a dynamic honeypot approach that provides comprehensive protection against ransomware through detection, mitigation, and restoration mechanisms. The technique operates by creating floods of decoy files that serve as honeypots to detect malicious activity. However, the main action of DFaR solutions regards slowing down ransomware operations through resource contention and moving target defence. The approach emphasises decoy deployment in sensitive locations where it can maximise its defensive impact.

DFaR implementations, such as the open-source Ranflood tool [10], employ various *flooding strategies* to generate decoy files. In particular, there are strategies that generate numerous files *with random content* but appropriate headers corresponding to common file extensions, aiming to make these random-content decoys indistinguishable from legitimate user files. However, random-content decoy generation faces significant challenges during the restoration phase. Indeed, to restore the system, one needs to distinguish between genuine user files and generated decoy ones. To discriminate between the two kinds of files, existing implementations rely on file lists; for instance, the approach implemented in Ranflood [10] is to generate a user-files list. In general, one can populate a user-files list either at scheduled times (e.g., daily, as one would do with regular backups) or before contrasting an attack (i.e., before the DFaR strategy starts flooding the system under attack) or, complementarily, a DFaR tool can save a decoy-files list that records all decoys it generated during flooding. To restore the system, if we have the user-files list, we can delete all files absent therein; vice versa, with the decoy-files list we can directly delete all files in the list. In both cases, losing the list would compromise the restoration of the system. Moreover, depending on when one generated the user-files list, the restoration could delete user files that are pristine (not encrypted by ransomware) but absent from the list (e.g., files created after the generation of the list). In general, relying on these lists introduces complexity (e.g., of saving the list in remote locations not reachable by ransomware) and potential points of failure of the random-content flooding category.

Contribution: In this paper, we present a novel solution that substantially improves the quality of the category of DFaR's random flooding strategies by introducing a watermarking-based technique for discriminating between user files and generated random-content decoys. Essentially, we propose to embed imperceptible watermarks within the randomly generated decoy files, enabling reliable identifica-

tion and removal during system restoration, dispensing the problematic usage of lists, while maintaining the deceptive properties essential for effective ransomware mitigation – so that an attacker would hardly discriminate between user and watermarked decoy files. Instead of having to deal with cumbersome file lists and different versions thereof, the only element that the user needs to preserve (specifically, from attacks) is a small piece of data, easy to keep safe, called the flood key, needed to watermark and identify the decoy files.

Structure of the Paper: We provide necessary background knowledge about DFaR and file watermarking techniques in Section II. Then, we present the theoretical foundations behind our watermarking approach for random-generated decoys in Section III. We immediately apply our formulation, in Section IV, by presenting a prototypical implementation and experimental benchmarks to evaluate the quality of the proposed solution. Specifically, we measure two critical performance dimensions: *overhead*, i.e., the technique's throughput compared against vanilla random decoy generation methods (that would rely on pre-existing file lists for restoration) and *robustness*, i.e., the watermarking strength against adversarial attempts to discriminate between watermarked and non-watermarked files (if the technique were not robust, then ransomware could detect the decoys and skip them; weakening the confusion element of the DFaR approach). In particular, given the parametric nature of our watermarking technique, we employ dual annealing optimisation to identify optimal parameter values that boost the watermarking's security guarantees. The experimental results demonstrate that our watermarking approach can afford *high performance*, since it imposes minimal overhead compared to vanilla random file generation. At the same time, we achieve *high reliability* in discriminating between proper and decoy files for legitimate users (useful for system restoration) and *high robustness* against inference-based discrimination attacks. In practical terms, the experimental evidence indicates that, using our technique, legitimate users can both efficiently defend (during an attack) and restore (after an attack) their systems, while ransomware would encounter significant challenges in distinguishing between authentic and decoy files, thereby enhancing the defensive efficacy of the random flooding category. We conclude by positioning our contribution within the literature in Section V and drawing closing remarks and discussing future work in Section VI.

II. Background

We introduce the ingredients on which we build our contribution: Data Flooding against Ransomware and Watermarking.

A. Data Flooding against Ransomware

DFaR is a recent technique [9] that offers robust protection against ransomware, ranging over detection, mitigation, and restoration phases. 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 static dummy element (e.g., file) deployed as an easy-to-access computer resource. The DFaR approach sees the concept of honeypot under a dynamic lens: it proposes the generation of "floods" of honeypot files at specific locations. In doing so, DFaR achieves *resource contention* with the ransomware, slowing down the attacker by contending with it access to resources (CPU, disk, etc.), and implementing a *moving target defence*, confounding user files with decoys the ransomware wastes time on (increasing the probability of preserving the user's data).

Conceptually, DFaR defines defence against ransomware using data flooding, but it does not specify the logic for generating the flooding files. For instance, in the seminal work that introduced DFaR [9], Berardi et al. present two categories of flooding strategies, one based on the generation of random-content files, and one based on copies of existing user files.

In this work, we concentrate on the random-content category of strategies. From Berardi et al.'s definition, a random-content flooding implementation should meet two conditions when generating decoy files: a) they should use the most common file extensions (e.g., pdf, jpeg, etc.) that are likely targeted by ransomware; b) they should be difficult to discriminate from the files of the user, e.g., contain different byte sequences and match file headers with the related extension.

Regarding restoration, one can recover a system from random-content decoys by removing the files generated during flooding. To implement this phase, Berardi et al. [9] use user-files lists so that one preserves only those files in the intersection between the ones in the list and those present on disk.

B. Watermarking

Quoting Mohanty et al. [11] "*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*".

Traditional use cases for digital watermarking regards protection of copyright in multimedia products, owner identification (the closest case to how we use watermarking in this work), and content authentication copy control [12].

Technically, we refer to multimedia objects as *work*s. We call a *cover work* a work which hides the watermark. To simplify our presentation, in this proposal, we mainly describe watermarking as applied in the image domain. Nonetheless, the presented techniques are trivially generalisable, as we do, to binaries, oblivious to the actual file content.

Before delving into the technical details, we present the components and nomenclature used henceforth.

The components of a watermarking scheme include: the *watermark*, which is the message to be embedded, a *key*, used to embed and detect the watermark (usually kept secret), a *watermark embedder* that implements a watermark insertion algorithm, called an *encoder*, and a *watermark detector*, which includes an *extractor* and a *comparator*, used respectively for the extraction and verification of the embedded watermark.

Nomenclature-wise, the *effectiveness* of a watermarking system is the probability of legitimately detecting the watermarking after embedding, while *fidelity* is the perceptual similarity between the non-watermarked and watermarked versions of a

work. The *data payload* is the amount of information that can be carried in a watermark. We have *informed detection* when the detection requires access to the original, non-watermarked work, while *blind detection* happens when the detector needs no information related to the original. The *false positive rate* is the frequency of watermark detections in non-watermarked content, while *robustness* and *security* are respectively the ability of the watermark to survive normal processing of content and to resist hostile attacks – in our case, the attack regards unauthorised detection. Moreover, the *cost* of watermarking corresponds to the computational cost of embedding and detection.

The watermark embedder performs the *embedding process*, encoding a message m, so that it matches the cover work domain, using a watermark key k. Essentially, the embedder uses the watermark encoder to embed the message within the cover work. Typically, the key generates a pattern w_r of the same size of 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 watermark detector performs the *detection process*, using the watermark key to determine whether a watermark is present and, if it is, extracting the message.

Geometric Models of Watermarking: From a geometric standpoint, we can view a watermarking system as a high-dimensional space, called a *media space*, in which each point corresponds to one work [12, chapter 3.4].

For example, 256-pixel square ($2^8 \times 2^8$) black-and-white images entail a $2^{8\times2}$-dimension (65,536) media space (one for each pixel, either white or black), while for a 5-second mono audio clip, sampled at 44,100Hz, we have a 220,500-dimension space (one for each sample).

Analysing a watermarking system through its media space helps, in particular, to visualise how watermark recognition works. The detection process relies on calculating the *correlation value* between the watermarked work and the secret pattern generated by the key.

Assume that we want to embed a message m in an image i with a key k. In the media space, a region called *distribution of non-watermarked works* includes i. After the embedding process, the cover work is in a different region of the media space. To guarantee fidelity, we should have the cover work located in a *region of acceptable fidelity*, i.e., the portion of space of works perceptually similar to the original one.

The *detection region* represents the set of all works in the media space where a detector can extract the message m, using key k, hence detecting the work as watermarked. The detection region is often defined by a threshold τ on a measure of similarity between the detector's input (a work, possibly watermarked) and a pattern that encodes m. We refer to this measure of similarity as a *detection measure*.

Given the n-dimensional vector c that represents a work, the measure for τ using linear correlation is equal to the product of their Euclidean lengths and the cosine of the angle between them, divided by n, i.e., $(c \cdot w_r)/n$. Because the Euclidean length of w_r is constant, the measure implies finding the orthogonal projection of c onto the vector w_r. The set of all points whose value is greater than τ is on one side of a plane perpendicular to w_r, which is the detection region.

One can use other measures of correlation. In this proposal, besides linear correlation, we use *normalised correlation*. Different measures generate different space partitions, which, practically, present advantages and disadvantages according to the different application contexts. Part of our contribution is evaluating which technique suits our use case the best.

Transform Domain: Of course, one can perform embedding and detection in different domains than the spatial one. For example, we can transform an image from the spatial domain (the pixel values) to the frequency domain, e.g., using the Discrete Cosine Transform (DCT) [13]. Among the reasons to use frequency-domain watermarking, there is achieving greater robustness against several kinds of data manipulations and to produce higher-fidelity cover work [14], [15].

Moving from the spatial to the frequency domain follows the formula [14], [16] $F(u,v) = C \cdot f(x,y) \cdot C^T$, where $F(u,v)$ is the frequency domain transformation of the input work. To simplify the explanation of the DCT transformation formula, we consider the input work to be a 2D square image of size $n \times n$. Then, we have an n^2-dimensional square matrix C multiplied by the input work $f(x,y)$ as an n^2-dimensional square image in the spatial domain – where the pixel values of the points $(x,y) \in n^2$ – and the transpose of C, C^T. C is an n^2-dimensional square transform matrix calculated as in Eq. (1) that contains the basis functions.

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

The power of this matrix formulation lies in its exploitation of the separability property of 2D transforms. Rather than computing a computationally expensive 2D transform directly, the operation $C \cdot f(x,y)$ first applies a 1D transform to each row of the image, and then the subsequent multiplication by C^T applies the 1D transform to each column of the intermediate result, reducing the computational complexity of the operation. Each element $F(u,v)$ in the resulting frequency domain represents how much the spatial signal correlates with a specific 2D sinusoidal pattern characterised by horizontal u and vertical v frequencies.

III. Watermarking for Random Flooding

Thanks to watermarking, we can improve random-based flooding by dispensing with file lists for the restoration phase. More specifically, with a suitable and properly configured watermarking system, if a detector with the (secret) key can detect the watermark, we know the file is a decoy, and we can safely remove it at restoration. On the other hand, the system should practically give no advantages to the attacker (the ransomware), both in terms of performance overhead

(which could degrade the contention effect of flooding) and target confusion (which would allow the ransomware to discriminate between proper and decoy files and diminish the confounding effect of honeypot files). More formally, the detector should correctly discriminate between files with and without watermarking, to ensure *soundness*, i.e., all files that the detector identifies as watermarked are decoys, and *completeness*, i.e., the detector identifies all watermarked files, which guarantee at least the same restoration properties as provided by using file lists – "at least" because file lists can get outdated, marking user files not present in the list as decoys, which cannot happen with watermarking.

A. Watermarked Random-content Flooding

The intervention we propose involves two parts; one corresponding to the watermark embedding process, performed in the mitigation phase (when contrasting a ransomware attack), and the other corresponding to the watermark detection process, carried out in the restoration phase (after an attack, removing the generated decoy files).

To detail the logic of the embedding and detection processes, we must define the properties of the desired watermarking system, considering the traits mentioned in Section II-B.

- *Embedding effectiveness*: The detector shall recognise all the files produced during flooding (completeness).
- *Fidelity*: we generate random-content files, so fidelity is immaterial (the decoys for flooding are cover works).
- *Data payload*: we are not concerned about the amount of information we have to embed since we just need to know if a file is watermarked (we do not extract the watermark).
- *Detector type*: as a consequence of the "fidelity" item, we use a blind detector (we aim to detect cover works).
- *Robustness*: the attacker (ransomware) could tamper with the decoy files, encrypting them – as intended by the confounding DFaR defence approach. Robustness against this type of modifications is immaterial since we can identify encrypted files by ransomware, which we can remove at restoration if considered useless/unrecoverable.
- *Security*: ransomware that would try to evade the flooding defence could attempt an *unauthorised detection* attack on the watermarked decoys (if the attacker can detect the watermark, it can skip the decoys and concentrate on the user files). Hence, we need a watermarking system that makes it expensive to perform unauthorised detection attacks – so the ransomware wastes time on watermark detection or encrypting decoys.

Considering the defined properties, we can now proceed to characterise the desired watermarking system.

B. Flooding and Embedding

The flooder performs the embedding process. Practically, the user generates a secret flood key k_f, used for the watermark file embeddings. More specifically, k_f comprises two subkeys: k_w, called the *watermark key*, used to generate the watermark pattern, and k_b, called the *blocks key*, used when performing watermarking based on multi-block embedding – we integrate a spread-spectrum multi-block watermarking to further increase our proposal's security, as discussed later.

For simplicity, we assume that the user keeps k_f secret and provides it for initialising the configuration of a DFaR tool that implements the random-content watermarking system discussed henceforth, storing the secret in an encrypted enclave. More sophisticated implementations can use asymmetric watermarking to increase security and simplify the process of key management (cf. Section VI).

Technically, k_w determines the random pattern added to the cover work (the decoy files) as the seed of a pseudo random number generator that creates a square matrix of 8-bit integer values, called the watermark. To generate the "original" content for the decoy files, one can use the same approach described by Berardi et al. [9, Algorithm 1], as we do in Algorithm 1, which shows a pseudocode description of the general logic of the watermarking random flooding we present.

The flooder in Algorithm 1 takes as inputs three parameters: the target $path$ of flooding, k_f, and par, a dictionary of parameters that configure the watermarking logic, such as the watermark size. The algorithm starts, at line 1, by generating a watermark pattern wt as an 8-bit integer-valued square matrix of the size indicated in par, using key k_w (part of k_f) as seed.

At line 2, we use a while loop controlled by an external variable (keepFlooding) to simplify the logic of flooding control, where we generate one file per iteration; actual implementations are more refined and use multiplexing techniques to coordinate the parallel generation of decoy files [9], [10].

To generate a decoy file, first, we have to determine its (random) size. In this implementation, we adapt an algorithm that works on 2D matrices with a fixed dimension – the columns, but the row-wise logic is complementary. The reason for fixing one dimension of the matrix is to know when to "wrap" the sequence of bytes that make up a file's content into a matrix form amenable to watermarking (both embedding and detection). Since we fix the number of columns that make up the file layout, determining its size means deciding how many rows it spans. We make this choice at line 3, where we determine the number of rows by taking a random integer between the size of the watermark (the file must be at least as large as the watermark pattern, to effectively apply the latter) and the parameter for maximum decoy file size. The product of the number of rows and the fixed column size determines the size of the file, corresponding to the number of bytes of an empty byte array we populate with random data. To achieve this point, at lines 5–14, we reuse the logic presented by Berardi et al. [9], based on xorshift [17], to quickly fill the file with random data. We finally apply the watermarking on the random-content file at line 15, using the function represented in Algorithm 2 – commented below. Once we obtain the watermarked content, we write it into a file, at line 16 – where function rndFilePath generates a file object that assigns a name and an extension to the decoy file and prefixes the bytes in cnt with the header related to the extension (e.g., it adds a PDF format header if

Algorithm 1 Random Data Flooding with Watermarking

Require: $path$, k_f, par
1: $wt \leftarrow$ generateWatermark($par.wtmarkSize$, $k_f.k_w$)
2: **while** (keepFlooding) **do**
3: $rows \leftarrow$randomInt($par.wtmarkSize$,$par.maxFileSize$)
4: $cnt \leftarrow$ newByteArray($rows$*$par.matrixColumnSize$)
5: $seed \leftarrow$ random64Seed()
6: **for** $i \leftarrow 0$ to capacity(cnt)/64 **do**
7: $seed \leftarrow seed \oplus (seed \ll 13)$
8: $seed \leftarrow seed \oplus (seed \gg 7)$
9: $seed \leftarrow seed \oplus (seed \ll 17)$
10: append(cnt, $seed$)
11: **if** capacity(cnt) > 0 **then**
12: $r \leftarrow$ newByteArray(capacity(cnt))
13: $r \leftarrow$ fillWithRandomBytes(r)
14: append(cnt, r)
15: $cnt \leftarrow$ watermark(cnt, wt, par, $k_f.k_b$)
16: writeFile(rndFilePath($path$), cnt)

Algorithm 2 Watermarking Process

Require: cnt, wt, par, k_b
1: **if** not $par.useBlocks$ **then**
2: **return** embed(cnt, wt, par)
3: $cntBlks \leftarrow$ divideInBlocks(cnt)
4: $wtBlks \leftarrow$ divideInBlocks(wt)
5: $irng \leftarrow$ rng.seed(k_b)
6: **for** $i \in wtBlks$ **do**
7: $j \leftarrow irng$.randInt(0, len($cntBlks$))
8: $cntBlks[j] \leftarrow$ embed($cntBlks[j]$, $wtBlks[i]$, par)
9: **return** $cntBlks$

Algorithm 3 Embedding Process

Require: cnt, wt, par
1: **if** $par.useDCT$ **then**
2: $c \leftarrow$DCT(cnt) + $par.wtIntesity$·DCT(wt)
3: **return** IDCT(c)
4: **return** $cnt + par.wtIntesity \cdot wt$

the filename extension is ".pdf").

To simplify our explanation, we divide the watermarking process, presented in Algorithm 2, from the embedding process, illustrated in Algorithm 3. Specifically, we implement two embedding processes in Algorithm 3 and two watermarking modalities in Algorithm 2. Starting from Algorithm 2, a watermarking modality entails the usage of a pseudorandom spreading sequence, generated from the block key k_b, to select which file blocks carry watermark information, similarly to spread spectrum methods [15]. Intuitively, using embedding based on spread-spectrum sequencing, we add a level of security to the proposed watermarking technique, since the attacker would also need to know which blocks it should analyse (the watermarked ones) and which to ignore (the others). The other modality classically applies the watermarking on the file following conventional sequential block processing. Selecting this last modality – by setting the $useBlocks$ parameter to false – directly calls the embed function, presented in Algorithm 3. Using block-based spreading watermarking, we first divide both the file content and the watermark pattern into blocks[1]. Since the file content is at least as large as the watermark (as per the definition of Algorithm 1), we have at most as many blocks as those of the watermark to process. Hence, we apply the embedding ranging over all the blocks of the watermark. At line 5 of Algorithm 2, we initialise a pseudorandom number generator using the block key k_b to determine on which block of the file content we apply that watermark block. We find this logic at lines 6–8, where, given the i-th watermark block, we determine the value of an index j, corresponding to the next integer provided by the pseudorandom number generator, and use j to target a block of the content that we watermark with the i-th watermark block (using the function in Algorithm 3).

[1]One could parametrise the size of the blocks but, as discussed in Section III-D, keeping the block size small increases the watermarking security. Thus, in this instance, we fix the block size to 8×8.

The last piece of the flooder's logic regards the actual embedding process, illustrated in Algorithm 3, which either performs a classical spatial-domain embedding, obtained through additive superposition, where it scales the watermark matrix wt by the intensity parameter $par.wtIntensity \in [0, 1]$ and adds the result directly to the original content matrix cnt. The intensity parameter controls the watermark strength through a scaling factor that determines the trade-off between imperceptibility and robustness. Alternatively, if the configuration parameter $useDCT$ is set to true, we embed the watermarking using the frequency domain, through the Discrete Cosine Transform (DCT). The algorithm first transforms both the original content matrix cnt and the watermark matrix wt into the frequency domain using DCT, then performs the additive embedding by scaling the watermark's DCT coefficients with $par.wtIntensity$ and adding them to the content's frequency coefficients. After combining the transformed signals, we calculate its inverse DCT (IDCT) to convert the watermarked frequency coefficients back to the spatial domain, producing the final watermarked content. Note that, when the *useBlocks* option is set to false, if the wt and cnt matrices differ in size – i.e., the watermark pattern could be smaller than the file content – the watermarking operations zero-pad wt to match cnt's dimensions.

C. Restoration and Detection

The restoration phase entails the detection process, i.e., the identification and removal of the watermarked random-content decoys. Also in this case, the user has to provide the flood key k_f to determine the watermark – and the block sequence, in case of spread-spectrum embedding – used to compare the content of the analysed files. Algorithm 4 illustrates the logic of the whole restoration process. We specify a $path$ that contains the files under analysis, the flood key k_f, and the parameters used for the flooding. Since the detection logic hinges on

Algorithm 4 Restoration (Detection) Process

Require: $path$, f_k, par, $threshold$, $corrType$

1: $wt \leftarrow$ generateWatermark($par.wtmarkSize$, $f_k.f_w$)
2: **for** $file$ in $path$ **do**
3: $f \leftarrow$readFile($file$)
4: $f \leftarrow$removeHeader(f)
5: $cnt \leftarrow$ newByteArray(getSize(f))
6: $cnt \leftarrow$ f
7: $corrVal \leftarrow$ calcCorr($cnt, wt, k_f.k_b, par, corrType$)
8: **if** $corrVal > threshold$ **then**
9: removeFile(file)

calculating the correlation between the watermark generated from the watermark key (k_w) and the content of the file, we provide to the algorithm a $threshold$ that sets the boundary for meaningful correlation (i.e., when a file's content correlation indicates it is a decoy watermarked by our system). The algorithm supports different correlation algorithms, specified using the $corrType$ parameter. Similarly to Algorithm 1, the first instruction of Algorithm 4 generates the watermark wt, used, in this case, to calculate the correlation with each file's content. Then, for each $file$ found within the provided $path$, we read its content (line 3), remove the header (at line 4; added using rndFilePath at line 15 of Algorithm 1), and generate a new byte array to store the content of the file (lines 5 and 6). The calcCorr function computes the correlation value between the file's content cnt and the watermark wt; if the correlation value ($corrVal$) is above the set $threshold$ (line 8) we have detected a removable (line 9) decoy file.

For brevity, we omit to present the pseudocode of the calcCorr function, since its logic is close to the one of Algorithm 2, with the major difference that calcCorr calculates a correlation value instead of calling the embed function, at lines 2 and 8. Essentially, if the $useBlocks$ parameter is false, we calculate the correlation between cnt and wt using the correlation measure indicated by the $corrType$ variable – in this work, we only implement linear and normalised correlation, but one can easily include alternatives by extending the calcCorr function. This piece of logic is similar to the one at lines 1–2 of Algorithm 2. Correspondingly, if $useBlocks$ is true, we have a procedure similar to the one found at lines 3–9 of Algorithm 2. Namely, we divide in blocks both cnt and wt, we use the block key k_b to determine the block sequencing and proceed to calculate the correlation between each wt's and cnt's corresponding blocks. The other meaningful modification of the code at lines 3–9 of Algorithm 2 is that, at line 8, we accumulate the correlation values into a variable, so that we return (at line 9) the average of all block-wise correlations.

For reference, we summarise the traits of the two types of correlation measures we consider. Linear correlation consists in calculating the average product of two vectors, in our case, the two matrices A and B of size $m \times n$. Formally, we define the linear correlation $\mathcal{Z}_{lc}(A, B)$ of matrices A and B as

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

In media space, comparing $\mathcal{Z}_{lc}$ against a threshold leads to a detection region with planar boundary. As highlighted by Cox et al. [12], a problem with linear correlation is that the detection values are highly dependent on the magnitudes of the compared vectors. Indeed, since the correlation measure depends on the absolute values rather than the relationship patterns between the elements, it can produce misleading results when comparing substantially different intensities and, e.g., the measure might report a low threshold for works that are structurally similar but have different overall magnitudes.

One can tackle the problems of linear correlation by normalising the matrices. Formally, we define the normalised correlation $\mathcal{Z}_{nc}(A, B)$ of matrices A and B (as presented above) by introducing a normalisation operator $\tilde{\bullet}$ such that $\tilde{A} = {}^{A}/_{|A|}$, where we apply $|A|$, norm of A, to all elements of the matrix

$$\mathcal{Z}_{nc}(A, B) = \sum_i \sum_j \tilde{A}[i, j] \cdot \tilde{B}[i, j]$$

In practice, normalised correlation mitigates the problems of linear correlation by removing the dependency on absolute magnitudes and focusing purely on the structural patterns and relationships between elements. Instead of directly multiplying raw values together (which inflates results for high-magnitude data and deflates them at the low-end of the spectrum), normalised correlation discounts the magnitudes of the input values by scaling them within a standard range (e.g., [-1,1]), i.e., the correlation measures how similarly the values vary relative to their own scales rather than their raw products.

D. Threat Model and Security Analysis

Now that we defined our watermarking approach, we can specify the threat model we consider and analyse its security.

We assume that an attacker cannot access the flood key k_f – and, thus, the subkeys k_w and k_b. In a possible attack scenario, ransomware would encrypt files, but it could also try to detect the watermark, which would allow it to skip the encryption of random-content decoy files, partially undermining the effectiveness of the flooding strategy. Technically, we define this type of attack an *unauthorised detection*, i.e., an attacker that can detect the watermarked files without the watermark key – and, in case of block-wise watermarking, the block key.

An attacker could attempt unauthorised detection through:

- *Brute force*: the attacker could try to detect the watermark by trying all possible flood keys. This path hardly seems feasible, since we assume the usage of 64-bit keys for both the watermark and block key, thus, the attacker would have to guess both keys in a 2^{128} space.
- *Cover work as watermark*: the attacker could use a known decoy (cover work) to detect the watermark. If our watermarking system is not robust, an attacker could use a watermarked file as a reference, which

could provide a correlation value sufficiently different from generic user files to make detection feasible. For example, if the embedder produced all decoy files of the same size, each decoy would have similarities with the others, due to embedding the watermark in the same way. Also for this reason, our embedding process generates files of different sizes. Moreover, using the spread-spectrum random block embedding technique, we further complicate the usage of cover work as watermark, since the watermarked blocks confound with the non-watermarked ones in patterns specific to each file (namely, their size, cf. line 7 of Algorithm 2) and the attacker would struggle to consistently compare cover works.

- *Rebuild the watermark*: an attacker could try to rebuild the watermark. Let us consider the harder case of decoys watermarked using the spread-spectrum block embedding. To implement this attack, the attacker should take two watermarked files (hence, it would need to find two files that are actually decoys) f_1 and f_2 and divide them into blocks. Then, for each block of f_1, the attacker would calculate the correlation with each block of f_2. The blocks with the highest correlation values are likely to contain the watermark. Hence, the attacker can try to rebuild the watermark by taking the blocks with the highest correlation values. For each other file, the attacker can try to detect the watermark by comparing each block found in e.g., f_1, with the blocks of that file.
  Mitigating this kind of attack mainly entails the adjustment of the embedding parameters to harden watermarking security. Specifically, one can use strong pseudo-random number generation algorithms [18] for generating the watermark and the block sequencing. Moreover, generating larger files (in particular, than the watermark size) increases the search surface for the attacker's analysis, which one can further complicate by adopting small block sizes (e.g., we set it to 8×8), to decrease the strength of correlation values.

IV. Implementation and Evaluation

Given the definitions in Section III, we implement a prototype version of the flooding/watermarking and restoration/detection algorithms. Besides serving as a reference implementation of our proposal, we use it to evaluate the effectiveness of the designed system. The implementation is written in Python, mainly for speed of development and availability of libraries like OpenCV for DCT. Production-grade implementations would use compiled binaries and native executables, as the DFaR Ranflood tool does [10]. Moreover, we implement a prototypical opponent attempting to perform unauthorised detection attacks (cf. Section III-D).

A. Evaluation: Watermarking Security and Overhead

We use our prototype implementation to test our proposal. We instantiate the embedder, detector, and attacker components with the same parameters – such as usage of random blocks, watermark size, and matrix format column size. While providing the same parameters to the embedder and the detector is expected, we underline that supplying them to the attacker implies considering an *advanced opponent*, able to exfiltrate and use the configuration parameters to drastically increase the efficiency of its attack – e.g., to perform an effective correlation attack, the attacker must know how to format the files in the correct matrix layout, i.e., the matrix format column size. An attacker ignorant of these parameters would face a harder task, since its search space would be far larger, resulting in low attack efficiency. As per threat model (cf. Section III-D), the attacker does not have access to the flood key (k_f and the watermark k_w and block k_b subkeys).

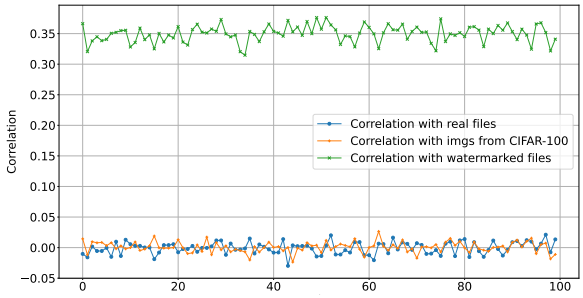
In the following, we start by benchmarking and exploring the configuration space of our proposal, in particular, through the usage of optimisation heuristics (dual annealing) to find an optimal set of configuration parameters that enjoys both *soundness* and *completeness* (cf. the first paragraph of Section III) and high security against unauthorised detection attacks.

Testing Protocol: To reliability test our proposal, we first need to specify a reference dataset of "user files". For reproducibility, we use 100 images randomly selected from the CIFAR-100 dataset. To verify that these images serve as valid proxies for real user files in our watermarking evaluation, we compare their correlation and that of 100 randomly selected files from a real user's work profile (containing documents, pictures, and videos) against watermarked decoy files generated by our flooder system. Hence, we generate a watermark pattern wt, generate with it 100 watermarked decoy files and measure the pair-wise linear correlation (the most "lenient" of the two measures we consider in this work, i.e., the one more likely to give false positives) between wt, the generated decoys, the user files, and the selected images.

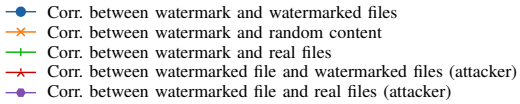
The idea is to verify that we can safely use the reference images in place of the user files because they have similar (low) correlation values with wt w.r.t. the decoy files. As expected, shown in Fig. 1a, both CIFAR-100 images and real user files exhibit similarly low correlation values (approaching 0) when compared to the watermarked decoys, while the decoys show high correlation with wt (0.7-0.8). This result demonstrates that CIFAR-100 images are suitable proxies for real-world user files in our watermarking detection scenario, since both exhibit the expected low correlation with watermarked content that our system relies on for detection.

Next, we need to define measures to quantify the effectiveness of the attacker in discriminating between user files and decoys. These measures are the *correlation between two watermarked (decoy) files* and the *correlation between a watermarked file and a user file*. An attacker, using these measures, could identify the files that exhibit the highest correlation value within a batch of files and then use it in the later stages of the attack to discriminate between decoys and proper user files.

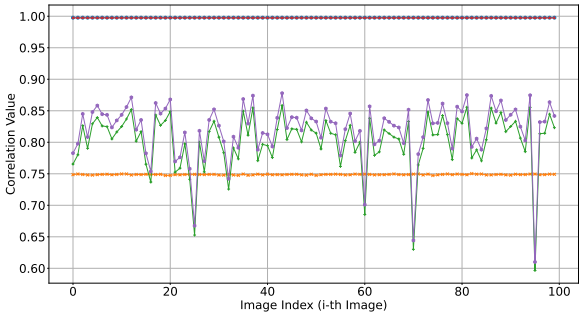
Exploring Watermarking Configurations: In Figs. 1c to 1f, we show several tests using different values of watermark inten-



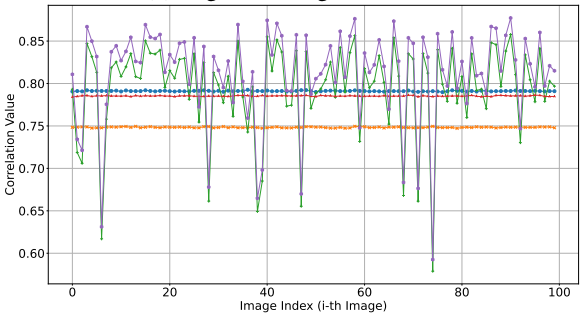
(a) Correlation with CIFAR-100 and Real Files.



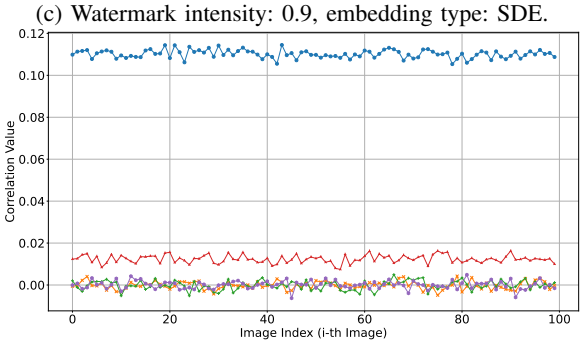
(b) Legend for Figs. 1c to 1f.



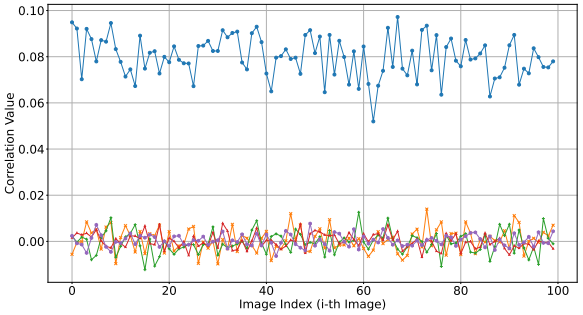
(c) Watermark intensity: 0.9, embedding type: SDE.



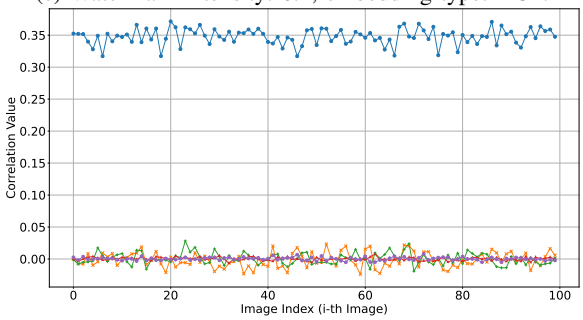
(d) Watermark intensity: 0.1, embedding type: SDE.



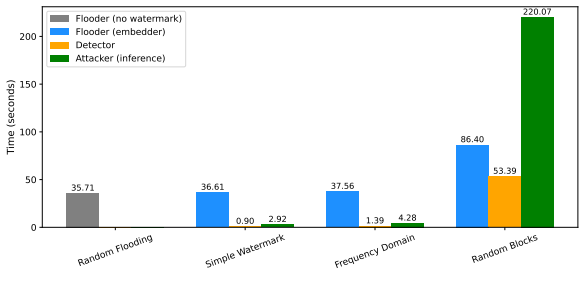
(e) Watermark intensity: 0.1, embedding type: DCT.



(f) Watermark intensity: 0.1, embedding type: DCT, random blocks.



(g) Optimal configuration, as per Table II.



(h) Performance comparisons: flooder/embedder, detector, attacker.

sity and embedding type. These intermediate insights are useful to explain the resulting best configuration parameters from running a dual annealing optimisation heuristic, commented afterwards. In all these tests, we adopt a watermark size of 512 and a matrix column size of 1024.

Figs. 1c to 1f show the results as line charts, where the x-axis represents a file (by its index) and the y-axis represents the correlation value; in this case, we use the stricter normalised correlation to minimise false positives. As visualised in Fig. 1b, in the charts, the blue line with circles plots the correlation between the watermark pattern wt and files that have been watermarked with it (watermarked decoys), the orange line with x's tracks the correlation between wt and files with random content (used as control to represent possible encrypted files by ransomware), the green line with triangles shows the correlation between wt and user files (CIFAR-100), the red line with squares indicates the correlation between a file and the other ones (calculated by the attacker), all watermarked using wt, and the purple line with circles represents the correlation between the watermarked file and user files (calculated by

the attacker). Intuitively, the best configuration maximises the distance between the blue line (legitimate decoy detections) and the other lines, while the latter are close to 0, to thwart unauthorised detection attacks.

Concretely, the opponent's attack generates 10 watermark candidates and selects a separate batch of 10 files suspected to contain watermarks. For each candidate watermark, the attacker calculates its correlation coefficient with every file in the batch, then computes the average correlation across all 10 files. Then, it repeats this process for each of the 10 candidates, creating a profile that reveals which candidate most closely matches the underlying watermarking scheme. The used watermark is most likely the one with the highest average correlation.

We start by analysing the results, shown in Fig. 1c, from a configuration that we expect to be weak, using the spatial domain embedding (SDE), strong watermark intensity (0.9), and no random blocks. As expected, legitimate detection works well, thanks to a compact average correlation of 0.99768. However, also the correlation calculated by the attacker is high – average 0.99735 – leading to an efficient unauthorised detection attack.

One could argue that the strongest factor that determines the weak security of this configuration is the high intensity level we set for the watermark. However, as shown in Fig. 1d, changing the watermark intensity to 0.1 seems to have little effect on decreasing the correlation value obtained by the attacker: the average of legitimate detection is ca. 0.79 while that of the attacker is just slightly lower, ca. 0.78.

For this reason, we move from the spatial domain to the frequency one, using DCT embedding. In Fig. 1e, we have the same parameters of the above configuration with the exception of using frequency-domain embedding. In this case, legitimate detection has an average correlation of ca. 0.11, while the average correlation calculated by the attacker is ca. 0.012. Notably, while both correlation values substantially dropped w.r.t. the previous experiments, the values for legitimate detection are an order of magnitude greater than those of the attacker (and even lower for the user files), indicative of two facts: first, we can safely define a threshold that discriminates between user and decoy files, and second, using DCT substantially hinders the ability of an attacker in discriminating between watermarked and non-watermarked files. However, despite the encouraging result, the distance between the attacker's correlation (the red line) and the other measures (the purple, orange, and green lines in Fig. 1f) provides some attack surface to infer a slight statistical difference between the user and decoy files.

As the last step of our investigation, we extend the above configuration with the usage of random blocks embedding, showing the results in Fig. 1f. Using this feature achieves more satisfactory results in terms of correlation strength for legitimate detection, while all other signals register as "noise", with an order of magnitude lower values than the former.

Optimising Watermarking Configuration Parameters: While our "manual" exploration of the security levels, granted by the tested parameter values, provides insight to establish a set of guidelines for watermarking configuration, we complement this investigation by applying an automatic method to more exhaustively analyse this search space.

Parameter	**Values**
Embedding type	SDE, DCT
Use random blocks	True, False
Watermark intensity	0.1–0.9
Watermark size	128, 256, 512
Matrix format column size	512, 1024, 2048
Block size	8, 16, 32

TABLE I: Watermarking configurable parameters and values.

Concretely, we model the problem of finding configuration values that maximise soundness, completeness, and security of watermarking flooding as an *objective function*. In particular, we want to minimise the difference between the values of a) correlation between a watermarked file and the other watermarked ones and b) correlation between a watermarked file and the user files, to minimise the likelihood that an attacker would be able to distinguish between the two kinds of files. At the same time, we want to maximise the difference (distance) between the values of a) correlation between the watermark and the watermarked files and b) correlation between the watermark and random content/user files, to ensure soundness and completeness.

We report in Table I the variables available to the optimisation function, with their possible values – we discretise some continuous variables, such as the watermark and block sizes, to restrict the search space.

To implement the optimisation model, we use *Scipy*'s [19] *optimize* module, a Python library that provides several functions for minimising/maximising objective functions. In particular, we use a *simulated annealing* [20] algorithm, called dual annealing, implemented therein [21]. The choice for using dual annealing is that this stochastic optimisation method can handle bounded mixed continuous (e.g., watermark intensity) and discrete (e.g., usage of random blocks) variables.

Since we have contrasting optimisation directions – minimise the chance of successful unauthorised attack and maximise the probability of correct legitimate detection – we have to encode one part as a complement of its definition. Indeed, since dual annealing (as most optimisation algorithms do) assumes the maximisation/minimisation of a single objective function, one cannot directly optimise goals with contrasting directions simultaneously. Instead, we must reformulate one objective as the mathematical complement of the other. Specifically, we choose to minimise the objective function, so that we can directly encode the minimisation of successful unauthorised attacks. Thus, instead of maximising legitimate detection probability, we need to reformulate that part of the function as the minimisation of the probability of failed legitimate detection (which is the complement of the detection probability). Also in this case, we use normalised correlation as correlation measure, which ensures more stable results and allows us to compare these results to the ones discussed earlier (and visualised in Figs. 1c to 1f).

Formally, we define the optimisation function f_o as

Parameter	Values
Embedding type	DCT
Use random blocks	True
Watermark intensity	0.3
Watermark size	256
Matrix format column size	1024
Block size	8

TABLE II: Optimised parameters found via dual annealing.

$$f_o = \underbrace{|\mathcal{Z}_{a,w} - \mathcal{Z}_{a,f}|}_{\text{min. detect. attack}} - \underbrace{|\mathcal{Z}_{w,w} - \mathcal{Z}_{w,r}| - |\mathcal{Z}_{w,w} - \mathcal{Z}_{w,f}|}_{\text{maximise legitimate detection}}$$

where $\mathcal{Z}_{a,w}$ is the correlation between a watermarked file and the other watermarked ones and $\mathcal{Z}_{a,f}$ is the correlation between a watermarked file and the user files, of which we consider the difference under absolute value to minimise any correlation distance that could help an attacker identify watermarked content, regardless of whether the correlation is positive or negative. Then, we encode the maximisation part as an element deduced from the above value. We break down the maximisation element into two subparts. The first indicates the maximisation of the distance between $\mathcal{Z}_{w,w}$, the correlation between the watermark and the file decoy files, and $\mathcal{Z}_{w,r}$, the correlation between the watermark and random-content files. The second expresses the maximisation of the distance between $\mathcal{Z}_{w,w}$ and $\mathcal{Z}_{w,f}$, the correlation between the watermark and the user files – by maximising these differences, we strive to maximise the discrimination between decoy files and real ones done by the detector. We take the absolute values of also these parts since we care about having a clear, measurable gap between the considered kinds of files and not for the specific direction of that gap – and, mathematically, the absolute value prevents the optimisation from getting confused by sign changes during the search process, i.e., without the absolute value the algorithm might find solutions where correlations flip signs, leading to unstable or suboptimal watermark parameters.

We report the optimised configuration parameters in Table II. The optimisation confirms the insights we gathered in our previous exploration. Namely, the usage of the frequency domain (DCT) and of spread-spectrum block sequencing. Interestingly, the optimisation procedure finds optimal results by generating both mid-sized watermark patterns (265) and files (with a matrix format column size of 1024), while keeping the block size small (8) – confirming the intuitions about watermark and file-content and block sizes in Sections III-B and III-D. Moreover, also intensity tends to stay on the lower-end spectrum (0.3), limiting the probability that the attacker would "sense" the watermark signal among the files.

To complete this part of our evaluation, we represent, in Fig. 1g, the chart obtained using the optimal configuration we found. Notably, this configuration further increases the gap between legitimate detection correlation (e.g., moving from ca. 0.09-0.11 of Figs. 1e and 1f to ca. 0.35) and compresses the correlation values for the other measures (around 0).

B. Watermarking Flooding Performance: Attack and Defence

We conclude our evaluation by looking at the performance of our proposal, in terms of the time required to embed and detect the watermark of 100 files. We compute these tests on a Debian 12 (bookworm) machine equipped with an Intel i3-8100 (3.6GHz) and 16GB of RAM.

Besides achieving our second benchmarking objective, i.e., measuring the overhead of the proposed system w.r.t. the existing alternative (that has to rely on cumbersome file lists), we also test the performance of an attacker that tries to run an inference unauthorised detection attack, declined over the main watermarking configurations we described: spatial-domain, frequency-domain, and spread-spectrum block sequencing frequency-domain watermarking. Since, in these tests, we are not concerned about the watermarking properties (correctness, soundness, and security) we fix the watermark size and the matrix format column size to 1024 (bigger sizes entail longer computations) to fairly compare with the existing DFaR random flooding methods. For each test, we generate/analyse 100 files and report the average, in Fig. 1h. Besides the performance of the flooder/embedder, we benchmark the performance of the detector through normalised correlation.

To benchmark the overhead between the existing random flooding methods and our proposal, we measure the performance of generating non-watermarked random decoy files – as done following the random flooding algorithm by Berardi et al. [9], which generally corresponds to skipping the watermarking passages in Algorithm 1. The time required to perform a 100-decoy file flooding in this way is 35.71 seconds.[2] We show this piece of data in the left-most histogram, called "Random Flooding" of Fig. 1h, under the "Embedder" column. The "Detector" and "Attacker" ones are immaterial for this test, since we are not using watermarking.

Next, we benchmark the performance of spatial-domain watermarking (called "Simple Watermarking" in Fig. 1h). As one can expect, the additional watermarking computations add a small overhead of ca. 1 second to the performance average of the flooder/embedder. Since this is the first time we benchmark the performance of the detector, we just report the (small) time of 0.90 seconds for analysing the generated files. Similarly, we have the first, baseline value for the attacker that (irrespective of the result, which we detailed and commented on in the first part of our evaluation) takes 2.92.

Moving to the frequency domain increases a bit the complexity of all computations – remember that we consider an advanced attacker that knows about configuration parameters and, thus, attempts an inference attack on the frequency domain. As reported under the "Frequency Domain" columns in Fig. 1h, also in this case we have a ca. 1-second increase for both the flooder/embedder (37.55 seconds) and the Detector (1.38 seconds) while the time spent by the attacker increases by ca. 46% (4.27 seconds).

[2]As discussed in Section VI, we present these benchmarks to compare the performance of random flooding with/out watermarking. Native, compiled implementations provide faster performance, as shown by Berardi et al. [9].

Adding random block sequencing to the mix (which arguably provides the highest level of soundness, completeness, and security) substantially increases all run times. The flooder/embedder and detector respectively record a ca. 230% (86.39 seconds) and ca. 3800% (53.39 seconds) increase. While important, these run times are heavily outweighed by the attacker's one, which records a ca. 5140% increase (220.07 seconds) due to the fact that it has to calculate the correlation for each block of any given file – and not just the watermark block-wise ones, as the flooder and embedder do.

V. Related Work

Ransomware defences have evolved across multiple paradigms, among which honeypot-based deception, backup and restoration, access control, file integrity monitoring. In what follows, we contextualise our contribution within these lines of research, emphasizing the novelty of watermarking-based Data Flooding against Ransomware (DFaR).

Despite the broad application of digital watermarking in copyright protection and data provenance, its use in ransomware defence is largely unexplored. Indeed, many works use honeypots to detect ransomware [22] – considering that the usage of honeypots is traditionally related to detection of fraudulent access – but only Berardi et al. [9] have employed (dynamic) honeypots to contrast ransomware attacks and restore the victim's system. Given the uniqueness and recency of the proposed technique, it is reasonable to have no direct alternatives to compare our work against. To the best of our knowledge, our method is the first to employ watermarking to enhance contrast and restoration in ransomware scenarios, allowing decoy files to be unambiguously identified post-attack without reliance on pre-generated file lists.

Broadening our scope, we find honeypot-based methods that provide detection of malicious activity. Early implementations such as Moore's [22] use static honey-files/folders to bait ransomware. More sophisticated systems, like R-Locker [23], use FIFO-accessed decoy files to detect a ransomware attack and trigger countermeasures. While effective in alerting users, these techniques generally use static or location-specific traps. On the contrary, the DFaR approach generalises the honeypot approach by dynamically flooding the system with decoys – in our case, with random-content ones that are indistinguishable from user files, watermarked to support post-attack restoration.

Traditional ransomware defences include data replication to off-site/cloud locations [24], [25] or the escrow of encryption keys [26]. Hardware-based techniques have also emerged, leveraging firmware characteristics to restore overwritten data [27], [28]. While these methods are effective, they depend heavily on infrastructure and availability. Our watermarking-based DFaR technique offers a lightweight, infrastructure-independent alternative, enabling on-disk recovery without requiring access to cloud services or hardware-level rollback.

Another line of defence focuses on preventing unauthorised file modifications via access control. For instance, McIntosh et al. [29] propose staged, policy-driven access control frameworks. While preventive, these approaches often require deep system integration and are vulnerable to privilege escalation. Our technique works orthogonally: it does not aim to block writes but ensures that even if writes occur, they likely affect decoys, not genuine user data.

Integrity monitoring solutions maintain hashes of critical files to detect changes. Approaches such as that of Kharraz et al. [30] monitor filesystem activity to detect early signs of ransomware. While effective for alerting, such systems do not offer recovery capabilities. Our watermarking system complements integrity monitors by embedding signals directly in decoys, enabling both detection and clean-up in a compromised system.

VI. Conclusion

In this work, we proposed a novel enhancement to Data Flooding Against Ransomware techniques by embedding imperceptible watermarks into random-content decoy files. Our approach enables reliable restoration without depending on fragile pre-attack file lists, improving both usability and resilience. Through rigorous experimental evaluation, we demonstrated that a) the proposed watermarking system achieves high detection accuracy with minimal false positives and negatives, b) offers robust resistance to inference-based attacks, even under advanced threat models, c) maintains good performance compared to standard flooding strategies, introducing limited computational overhead.

In particular, empirical evidence demonstrates that our system can provide high reliability and security using frequency-domain embedding and spread-spectrum block watermarking.

Looking at future work, an immediate step forward from this work is to integrate other watermarking techniques, studying their security and performance profiles, which might lead to supporting multiple watermarking configurations, e.g., given the nature of user files (e.g., of certain formats, that tend to be small/big) and the environmental context (e.g., cloud, edge, and internet-of-things applications).

In this work, for simplicity, we describe the usage of symmetric keys for embedding and detecting watermarking, suggesting the storage of the keys within an encrypted enclave. A method to avoid relying on enclaves is the usage of asymmetric watermarking. Indeed, more sophisticated implementations of our proposal can rely on asymmetric keys schemes where the embedder uses the public key to watermark the decoys and the detector uses the private key to identify the watermarked files. In this refined implementation, the user only provides the public key to the flooding tool, without worrying about keeping it secret, since it is useless to an attacker that wants to detect the watermarking. As usual with asymmetric key schemes, keeping secret the private key is simple, since the user can safely store it in a location detached from the attacked system (e.g., a digital safe/wallet) for later usage, post-attack.

A third future direction involves applying our watermarking-based approach to address exfiltration ransomware attacks. In such attacks, adversaries exfiltrate sensitive data to exert further pressure on victims. One can use watermarked decoys to also pollute exfiltrated data with traceable content. Indeed, by embedding watermarks into decoy files, defenders can afford

forensic tracking of stolen files – e.g., when they appear on leak sites or black markets – by assigning unique watermark signatures that can link exfiltrated files back to a specific machine or user profile. Moreover, watermarked decoys could also serve as digital tripwires for detecting and attributing leaks, helping to identify the source of attacks.

An interesting insight we noticed during our experiments is the correlation values calculated between user files and random files. Indeed, while the average correlation between the user files is relatively low and negative, the correlation of the decoy files (without watermarking) tends to be a little higher and positive. Although this fact cannot directly lead an attacker to recognising random-content decoy files, to ensure higher levels of decoy deception, one could try to study the patterns that characterise the different file formats, in an effort to mimic them when producing random-content decoys related to a specific format – we recall that, in Algorithm 1, we prefix the content of a decoy file with a header corresponding to the file extension assigned to it, so, e.g., mimicking a PDF-like byte pattern distribution for decoy files with a PDF extension would further increase the perceived authenticity of decoy files.

ACKNOWLEDGEMENT

Research partly supported by project PNRR CN HPC - SPOKE 9 - Innovation Grant LEONARDO - TASI - RTMER funded by the NextGenerationEU European initiative through the MUR, Italy (CUP: J33C22001170001). We thank Matteo Cicognani for supporting the collaboration between ARPAE and Università di Bologna.

REFERENCES

[1] H. Oz, A. Aris, A. Levi, and A. S. Uluagac, "A survey on ransomware: Evolution, taxonomy, and defense solutions," *ACM Comput. Surv.*, vol. 54, Sept. 2022.

[2] C. Beaman, A. Barkworth, T. D. Akande, S. Hakak, and M. K. Khan, "Ransomware: Recent advances, analysis, challenges and future research directions," *Computers & security*, vol. 111, p. 102490, 2021.

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

[4] N. Perlroth, M. Scott, and S. Frenkel, "Cyberattack hits ukraine then spreads internationally," Jun 2017.

[5] A. Griffin, "'petya' cyber attack: Chernobyl's radiation monitoring system hit by worldwide hack." https://techbeacon.com/security/ransomware-rise-evolution-cyberattack, 2017.

[6] P. H. Meland, Y. F. F. Bayoumy, and G. Sindre, "The ransomware-as-a-service economy within the darknet," *Comput. Secur.*, vol. 92, p. 101762, 2020.

[7] B. A. S. Al-rimy, M. A. Maarof, and S. Z. M. Shaid, "Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions," *Computers & Security*, vol. 74, pp. 144–166, 2018.

[8] T. McIntosh, A. S. M. Kayes, Y.-P. P. Chen, A. Ng, and P. Watters, "Ransomware mitigation in the modern era: A comprehensive review, research challenges, and future directions," *ACM Comput. Surv.*, vol. 54, Oct. 2021.

[9] D. Berardi, S. Giallorenzo, A. Melis, S. Melloni, L. Onori, and M. Prandini, "Data flooding against ransomware: Concepts and implementations," *Computers & Security*, vol. 131, p. 103295, 2023.

[10] D. Berardi, S. Giallorenzo, A. Melis, S. Melloni, and M. Prandini, "Ranflood: A mitigation tool based on the principles of data flooding against ransomware," *SoftwareX*, vol. 25, p. 101605, 2024.

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

[12] I. Cox, M. Miller, J. Bloom, J. Fridrich, and T. Kalker, *Digital Watermarking and Steganography*. Morgan Kaufmann, 2007.

[13] V. Britanak, P. C. Yip, and K. R. Rao, *Discrete cosine and sine transforms: general properties, fast algorithms and integer approximations*. Elsevier, 2010.

[14] H. Li and X. Guo, "Embedding and extracting digital watermark based on dct algorithm," *Journal of Computer and Communications*, vol. 6, no. 11, pp. 287–298, 2018.

[15] I. J. Cox, J. Kilian, F. T. Leighton, and T. Shamoon, "Secure spread spectrum watermarking for multimedia," *IEEE transactions on image processing*, vol. 6, no. 12, pp. 1673–1687, 1997.

[16] O. S. C. Vision, "Dct." https://docs.opencv.org/4.x/d2/de8/group__core__array.html#ga85aad4d668c01fbd64825f589e3696d4.

[17] G. Marsaglia, "Xorshift rngs," *Journal of Statistical software*, vol. 8, pp. 1–6, 2003.

[18] R. McEvoy, J. Curran, P. Cotter, and C. Murphy, "Fortuna: cryptographically secure pseudo-random number generation in software and hardware," in *2006 IET Irish Signals and Systems Conference*, pp. 457–462, IET, 2006.

[19] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[20] C. Tsallis and D. A. Stariolo, "Generalized simulated annealing," *Physica A: Statistical Mechanics and its Applications*, vol. 233, no. 1-2, pp. 395–406, 1996.

[21] T. S. community, "dual annealing." https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html#id5.

[22] C. Moore, "Detecting ransomware with honeypot techniques," in *Cybersecurity and Cyberforensics Conference, CCC 2016, Amman, Jordan, August 2-4, 2016*, pp. 77–81, IEEE, 2016.

[23] J. A. Gómez-Hernández, L. Álvarez-González, and P. García-Teodoro, "R-locker: Thwarting ransomware action through a honeyfile-based approach," *Comput. Secur.*, vol. 73, pp. 389–398, 2018.

[24] J. Yun, J. Hur, Y. Shin, and D. Koo, "Cldsafe: An efficient file backup system in cloud storage against ransomware," *IEICE Trans. Inf. Syst.*, vol. 100-D, no. 9, pp. 2228–2231, 2017.

[25] D. R. Matos, M. L. Pardal, G. Carle, and M. Correia, "Rockfs: Cloud-backed file system resilience to client-side attacks," in *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018* (P. Ferreira and L. Shrira, eds.), pp. 107–119, ACM, 2018.

[26] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "Paybreak: Defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017* (R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, eds.), pp. 599–611, ACM, 2017.

[27] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, "Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 2231–2244, ACM, 2017.

[28] S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang, "Ssd-insider: Internal defense of solid-state drive against ransomware with perfect data recovery," in *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pp. 875–884, IEEE Computer Society, 2018.

[29] T. R. McIntosh, A. S. M. Kayes, Y. P. Chen, A. Ng, and P. A. Watters, "Applying staged event-driven access control to combat ransomware," *Comput. Secur.*, vol. 128, p. 103160, 2023.

[30] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016* (T. Holz and S. Savage, eds.), pp. 757–772, USENIX Association, 2016.