Capture the mark
"Battle Report"

Paolo Chistè Claudio Facchinetti Matteo Franzil Tommaso Fanelli

University of Trento

November 9, 2021

Summary

Initial idea

We can 'concentrate' the mark in parts that are better at hiding it (like wavelet embedding)

Embedding steps

Given a mark of size N:

1. Divide the image into M sub-blocks
2. Find M blocks with the highest variance ("candidate blocks")
3. Embed N/M parts of the mark into each candidate block, where each new coefficient of the watermarked image is defined as:

$$x_{ij}^{*} = x_{ij}[1 + (\alpha m_i)]$$

Figure: Example of candidate blocks in the 'rollercoaster' image, with different block sizes

Embedding steps

Given a mark of size N:

1. Divide the image into M sub-blocks
2. Find M blocks with the highest variance ("candidate blocks")
3. Embed N/M parts of the mark into each candidate block, where each new coefficient of the watermarked image is defined as:

$$x_{ij}^{*} = x_{ij}[1 + (\alpha m_i)]$$

Figure: Example of candidate blocks in the 'rollercoaster' image, with different block sizes

Embedding steps

Given a mark of size N:

1. Divide the image into M sub-blocks
2. Find M blocks with the highest variance ("candidate blocks")
3. Embed N/M parts of the mark into each candidate block, where each new coefficient of the watermarked image is defined as:

$$x_{ij}^{*} = x_{ij}[1 + (\alpha m_i)]$$

Figure: Example of candidate blocks in the 'rollercoaster' image, with different block sizes

The texture_areas function

The texture_areas function is called by both the embedding and detection functions and provides them with a list of ordered blocks, in which the watermark will be inserted or detected.

The function computes the variance of the image and then divides it into $n \times n$ blocks. Each block's variance is computed and then tested against the image's. If it is greater, then the block is added to the list.

Since some images may have an unsatisfactory number of high-variance blocks (e.g. very flat images with small contrast areas), the function will recursively call itself, lowering the variance theshold each time until either it runs out of iterations or finds enough blocks.

The texture_areas function

The texture_areas function is called by both the embedding and detection functions and provides them with a list of ordered blocks, in which the watermark will be inserted or detected.

The function computes the variance of the image and then divides it into $n \times n$ blocks. Each block's variance is computed and then tested against the image's. If it is greater, then the block is added to the list.

Since some images may have an unsatisfactory number of high-variance blocks (e.g. very flat images with small contrast areas), the function will recursively call itself, lowering the variance theshold each time until either it runs out of iterations or finds enough blocks.

The texture_areas function

The texture_areas function is called by both the embedding and detection functions and provides them with a list of ordered blocks, in which the watermark will be inserted or detected.

The function computes the variance of the image and then divides it into $n \times n$ blocks. Each block's variance is computed and then tested against the image's. If it is greater, then the block is added to the list.

Since some images may have an unsatisfactory number of high-variance blocks (e.g. very flat images with small contrast areas), the function will recursively call itself, lowering the variance theshold each time until either it runs out of iterations or finds enough blocks.

The `texture_areas` function

The function uses a threshold based on the `sqrt` of the variance in order to decide when to stop searching for blocks. Its choice is purely experimental: we tried with different indicators but found the square root the most flexible.

Its great flexibility means it can easily be adapted to the situation by fastening or slowing the search, changing the block size, and so on.

Finally, found blocks are sorted by variance and sent back to the caller.

The `texture_areas` function

The function uses a threshold based on the `sqrt` of the variance in order to decide when to stop searching for blocks. Its choice is purely experimental: we tried with different indicators but found the square root the most flexible.

Its great flexibility means it can easily be adapted to the situation by fastening or slowing the search, changing the block size, and so on.

Finally, found blocks are sorted by variance and sent back to the caller.

The `texture_areas` function

The function uses a threshold based on the `sqrt` of the variance in order to decide when to stop searching for blocks. Its choice is purely experimental: we tried with different indicators but found the square root the most flexible.

Its great flexibility means it can easily be adapted to the situation by fastening or slowing the search, changing the block size, and so on.

Finally, found blocks are sorted by variance and sent back to the caller.

The texture_areas function

In short, the parameters are:

- `block_size`: the size of the searched blocks.
- `coeff`: a float that divides the variance (and since it's usually > 0, it reduces it) when the algorithm decides whether or not to choose a block
- `max_iteration`: maximum recursion depth
- `var_multiplier`, `coeff_step`: at the recursion point, the algorithm does the following:

```
if len(result) < var_multiplier * math.sqrt(image_variance):
    texture_areas(image, block_size,
                  coeff / var_multiplier,
                  result, it=it + 1)
```

The texture_areas function

In short, the parameters are:

- `block_size`: the size of the searched blocks.
- `coeff`: a float that divides the variance (and since it's usually > 0, it reduces it) when the algorithm decides whether or not to choose a block
- `max_iteration`: maximum recursion depth
- `var_multiplier`, `coeff_step`: at the recursion point, the algorithm does the following:

```
if len(result) < var_multiplier * math.sqrt(image_variance):
    texture_areas(image, block_size,
                  coeff / var_multiplier,
                  result, it=it + 1)
```

The texture_areas function

In short, the parameters are:

- `block_size`: the size of the searched blocks.
- `coeff`: a float that divides the variance (and since it's usually > 0, it reduces it) when the algorithm decides whether or not to choose a block
- `max_iteration`: maximum recursion depth
- `var_multiplier`, `coeff_step`: at the recursion point, the algorithm does the following:

```
if len(result) < var_multiplier * math.sqrt(image_variance):
    texture_areas(image, block_size,
                  coeff / var_multiplier,
                  result, it=it + 1)
```

The texture_areas function

In short, the parameters are:

- `block_size`: the size of the searched blocks.
- `coeff`: a float that divides the variance (and since it's usually > 0, it reduces it) when the algorithm decides whether or not to choose a block
- `max_iteration`: maximum recursion depth
- `var_multiplier`, `coeff_step`: at the recursion point, the algorithm does the following:

```
if len(result) < var_multiplier * math.sqrt(image_variance):
    texture_areas(image, block_size,
                  coeff / var_multiplier,
                  result, it=it + 1)
```

The texture_areas function, changing parameters

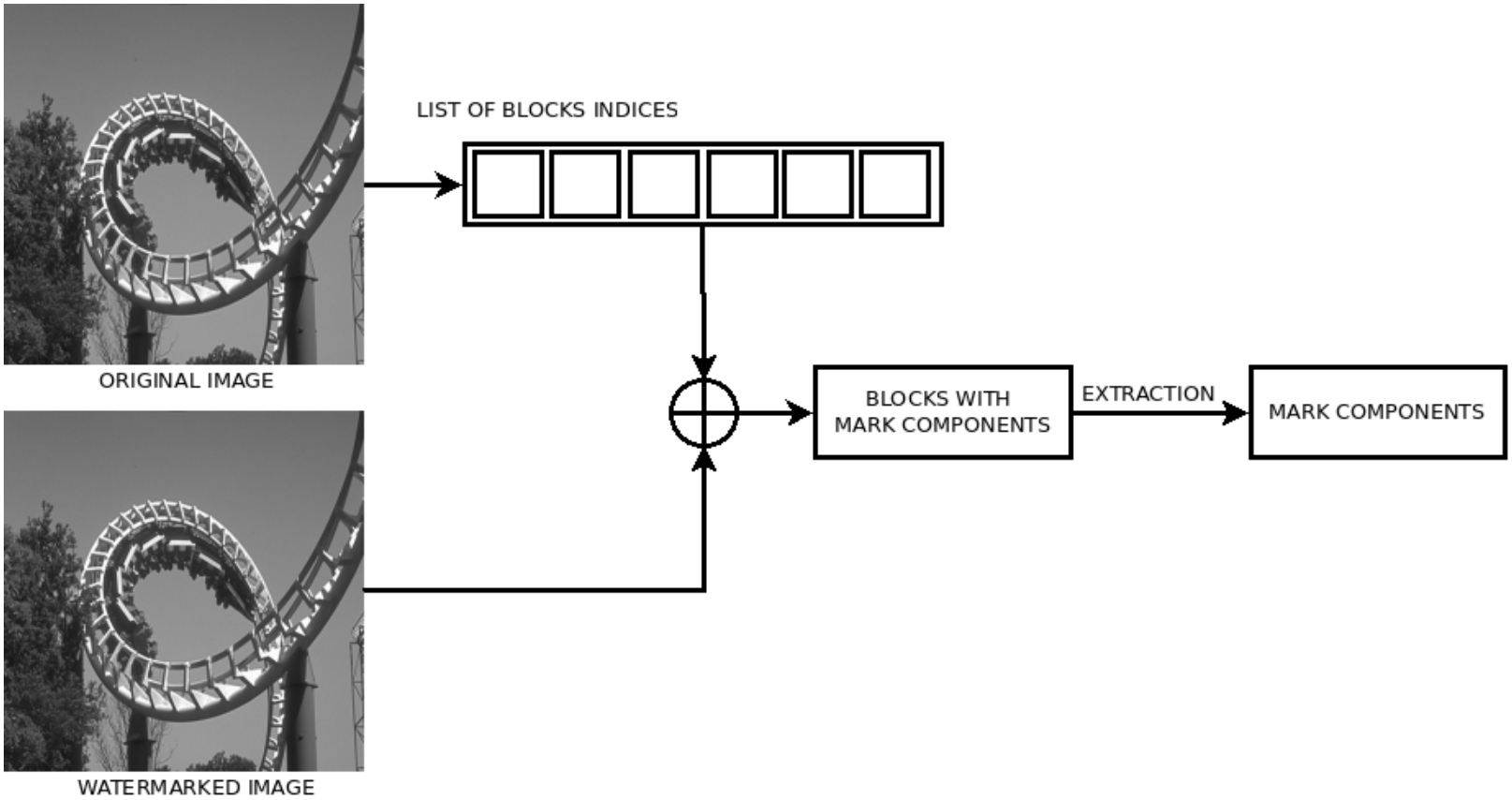
Figure: Example of candidate blocks in the 'rollercoaster' image, when the *coeff* coefficient is changed

The texture_areas function, changing parameters


Figure: Example of candidate blocks in the 'rollercoaster' image, when the var_coeff coefficient is changed

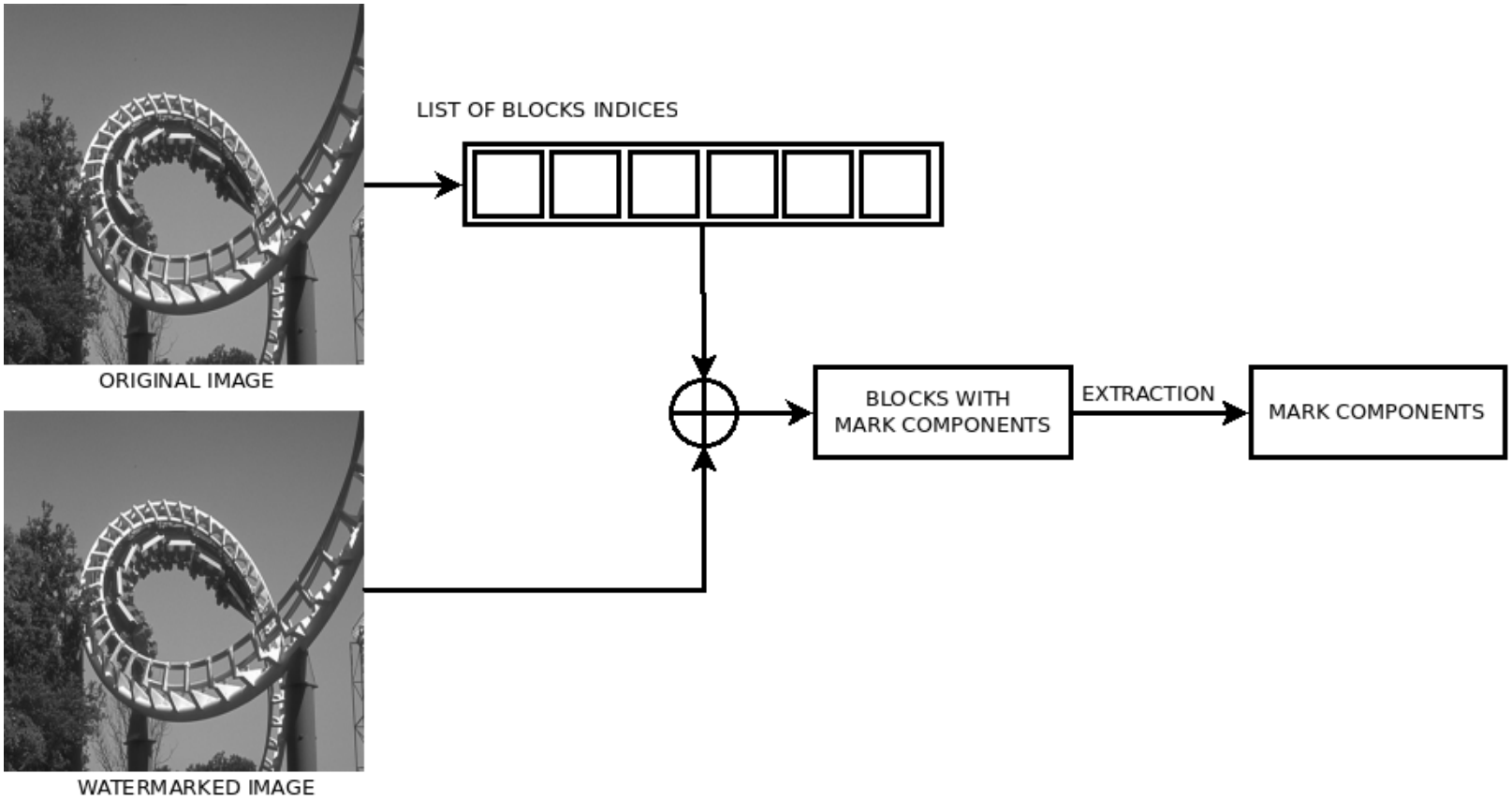
Detection steps

1. Find candidate blocks using the original image $\Rightarrow$ this is possible because detection is non-blind.
2. Extract mark from candidate blocks
3. Reassemble mark



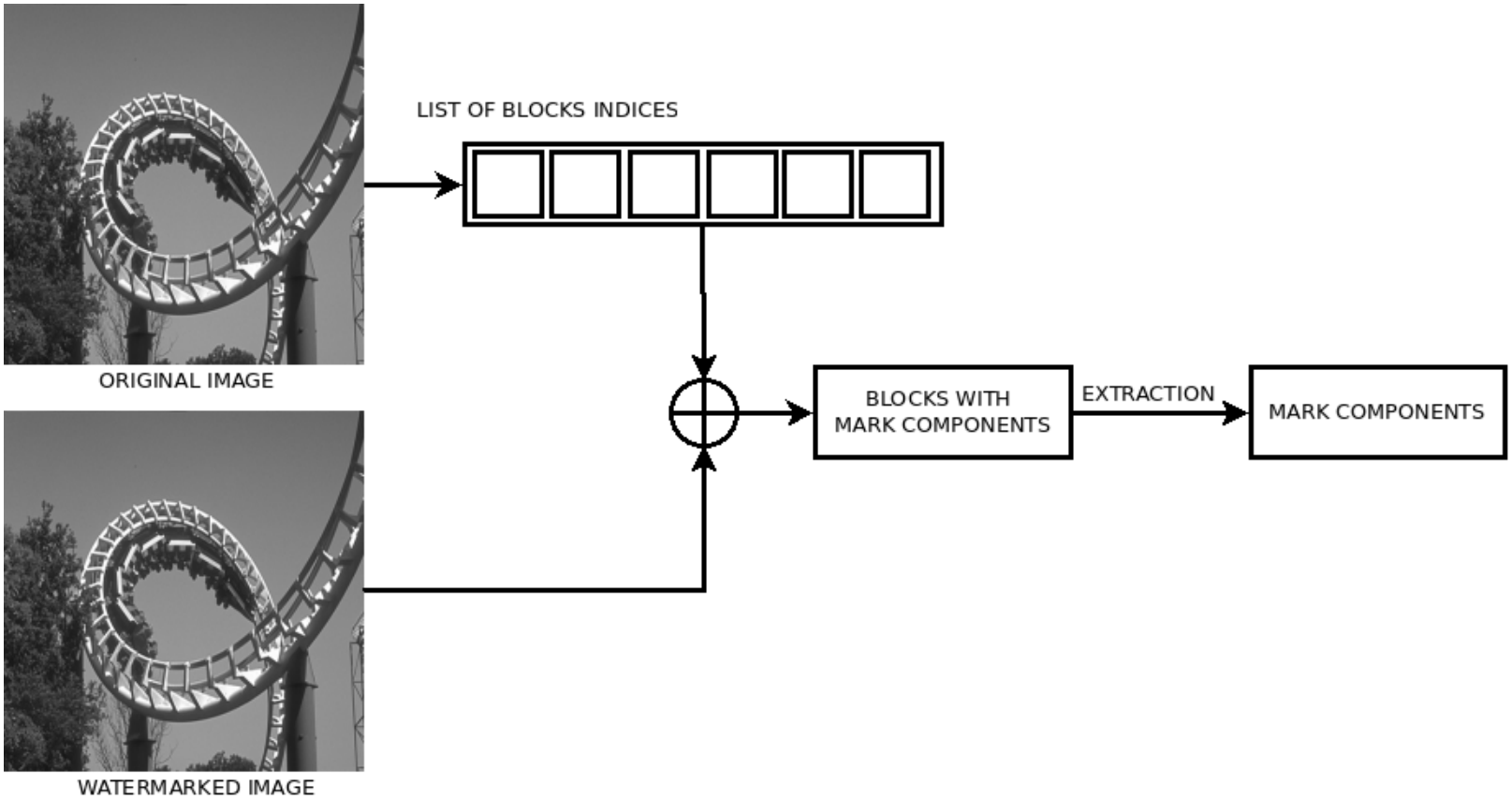
Detection steps

1. Find candidate blocks using the original image $\Rightarrow$ this is possible because detection is non-blind.
2. Extract mark from candidate blocks
3. Reassemble mark



Detection steps

1. Find candidate blocks using the original image $\Rightarrow$ this is possible because detection is non-blind.
2. Extract mark from candidate blocks
3. Reassemble mark



Strength and weaknesses

Strengths:

- Strong parametrization and flexibility
- Fast (statistically insignificant slowdowns compared to standard SS watermarking)
- Easily extensible by changing employed metrics
- At lower α, very high WPSNR and undetectable by the naked eye

Weaknesses:

- Extremely dependent on the image features
- Embedded blocks locations are deterministic and predictable to the attacker
- **Underflow/overflow problems** when converting from DCT domain to pixel domain

Strength and weaknesses

Strengths:

- Strong parametrization and flexibility
- Fast (statistically insignificant slowdowns compared to standard SS watermarking)
- Easily extensible by changing employed metrics
- At lower α, very high WPSNR and undetectable by the naked eye

Weaknesses:

- Extremely dependent on the image features
- Embedded blocks locations are deterministic and predictable to the attacker
- **Underflow/overflow problems** when converting from DCT domain to pixel domain

One-attack test results

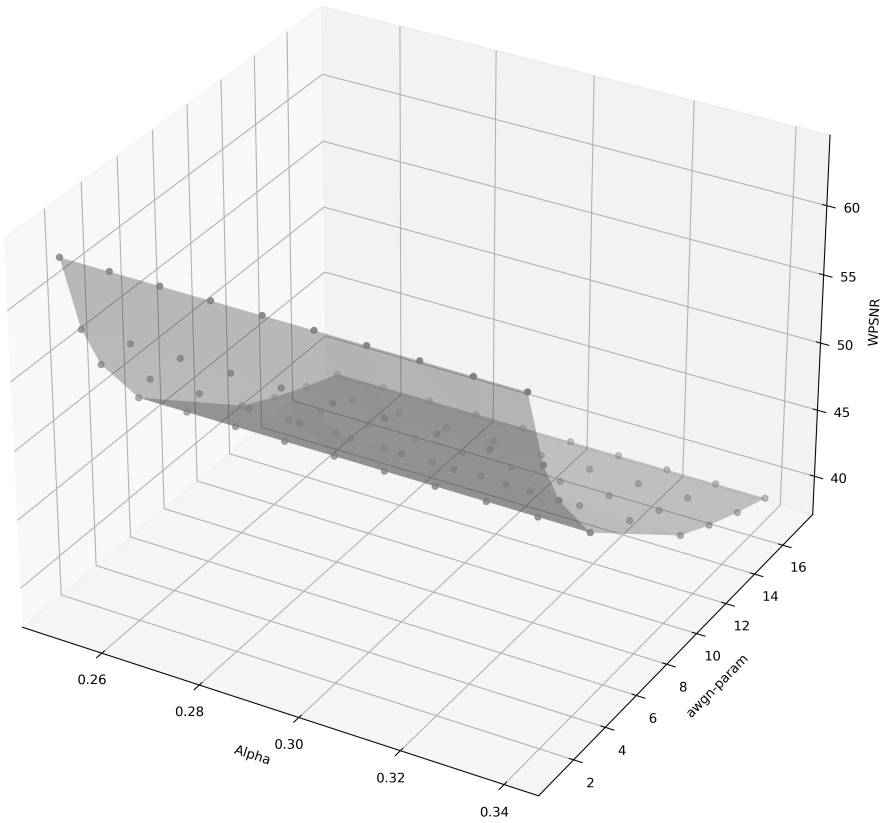


Figure: Plot of α, WPSNR and AWGN attacks (random seed). Red dots are successful attacks.

One-attack test results

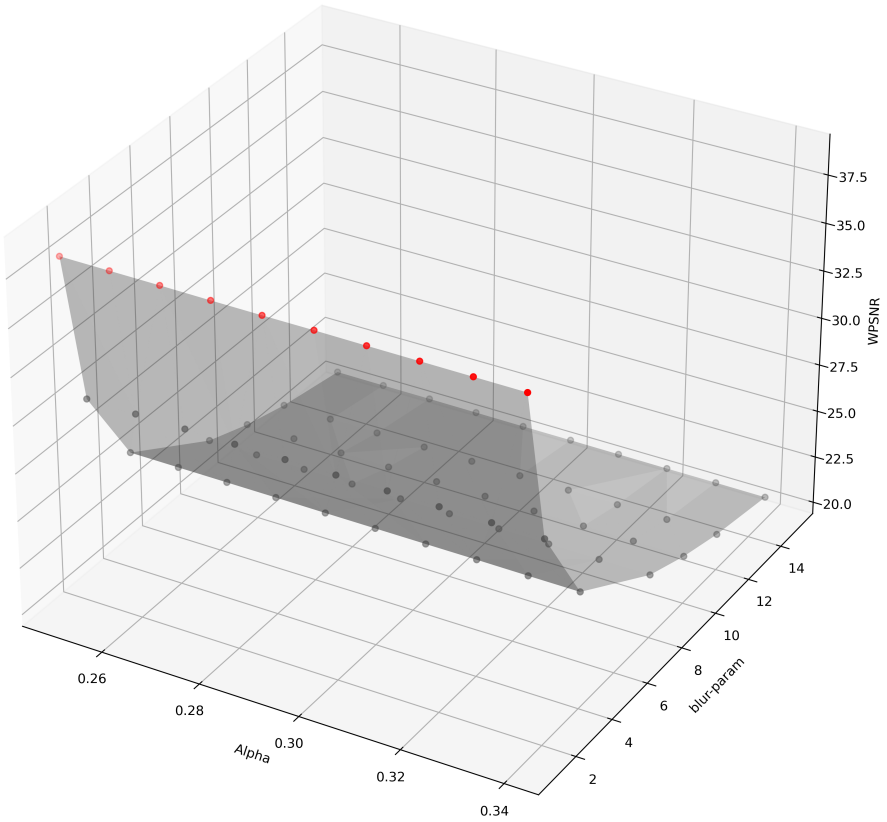


Figure: Plot of α, WPSNR and blur attacks. Red dots are successful attacks.

One-attack test results

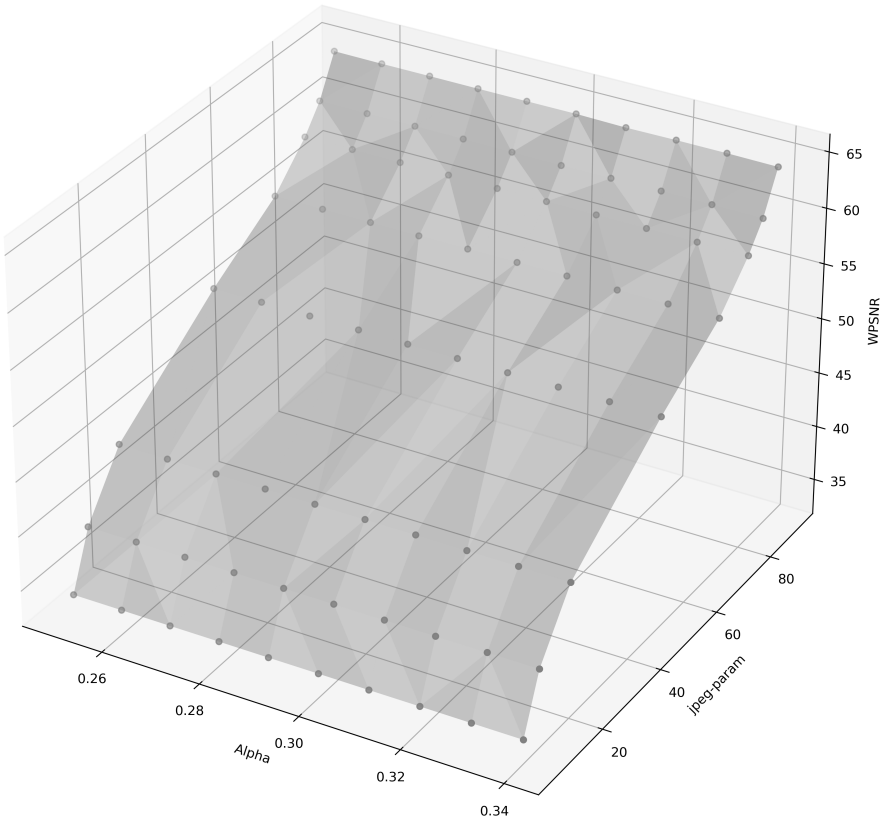


Figure: Plot of α, WPSNR and JPEG attacks. Red dots are successful attacks.

One-attack test results

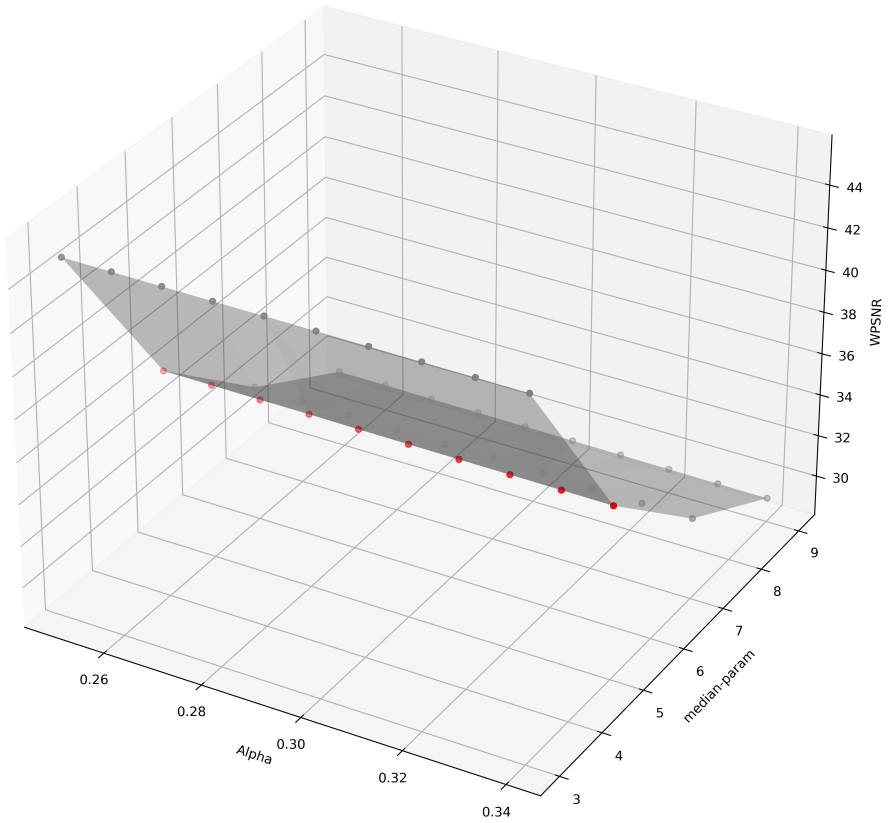


Figure: Plot of α, WPSNR and median attacks (square sliding window). Red dots are successful attacks.

One-attack test results

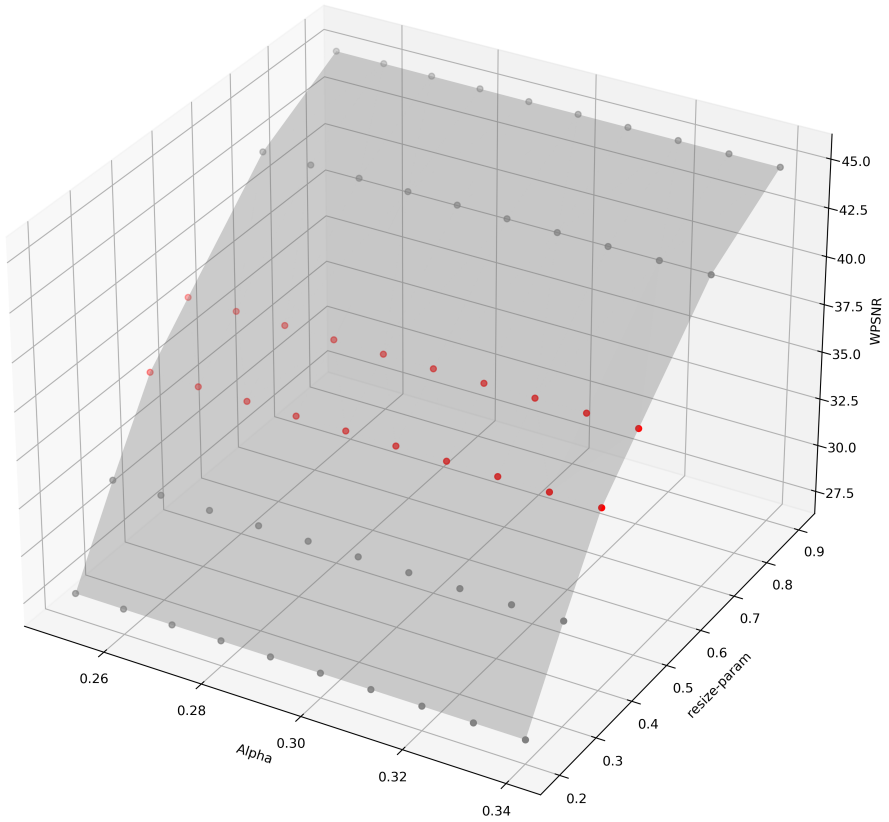


Figure: Plot of α, WPSNR and resize attacks. Red dots are successful attacks.

One-attack test results

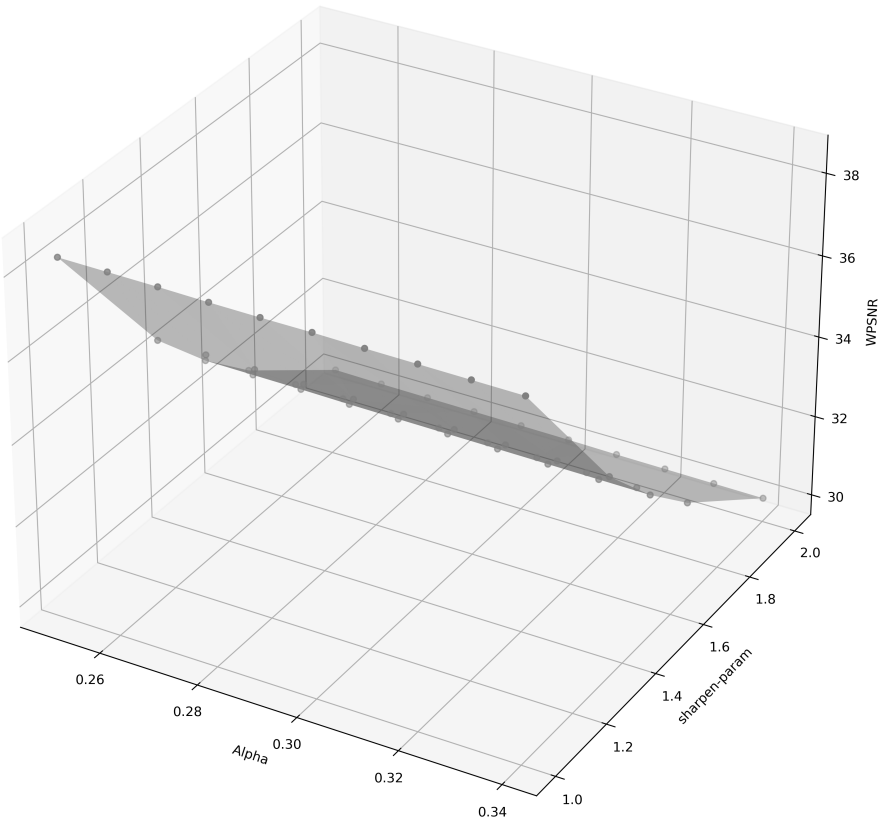


Figure: Plot of α, WPSNR and sharpen attacks (fixed sigma). Red dots are successful attacks.

Underflow/overflow problems

The underflows and overflows destroys parts of the mark during the embedding phase, resulting in information loss.

A multitude of factors contributed to this game-losing, still-open issue:

- Bad typecasting in both embedding and detection caused conversion errors
- Editing a large number of coefficients in a small DCT and re-converting caused pixels to go negative or over 255

During the re-conversion, the offending bits flipped and created white and dark patches in the image, razing the WPSNR and ruining the detection by up to 50/60%.

Underflow/overflow problems

The underflows and overflows destroys parts of the mark during the embedding phase, resulting in information loss.

A multitude of factors contributed to this game-losing, still-open issue:

- Bad typecasting in both embedding and detection caused conversion errors
- Editing a large number of coefficients in a small DCT and re-converting caused pixels to go negative or over 255

During the re-conversion, the offending bits flipped and created white and dark patches in the image, razing the WPSNR and ruining the detection by up to 50/60%.

Underflow/overflow problems

The underflows and overflows destroys parts of the mark during the embedding phase, resulting in information loss.

A multitude of factors contributed to this game-losing, still-open issue:

- Bad typecasting in both embedding and detection caused conversion errors
- Editing a large number of coefficients in a small DCT and re-converting caused pixels to go negative or over 255

During the re-conversion, the offending bits flipped and created white and dark patches in the image, razing the WPSNR and ruining the detection by up to 50/60%.

Underflow/overflow problems

The underflows and overflows destroys parts of the mark during the embedding phase, resulting in information loss.

A multitude of factors contributed to this game-losing, still-open issue:

- Bad typecasting in both embedding and detection caused conversion errors
- Editing a large number of coefficients in a small DCT and re-converting caused pixels to go negative or over 255

During the re-conversion, the offending bits flipped and created white and dark patches in the image, razing the WPSNR and ruining the detection by up to 50/60%.

Underflow/overflow problems

The underflows and overflows destroys parts of the mark during the embedding phase, resulting in information loss.

A multitude of factors contributed to this game-losing, still-open issue:

- Bad typecasting in both embedding and detection caused conversion errors
- Editing a large number of coefficients in a small DCT and re-converting caused pixels to go negative or over 255

During the re-conversion, the offending bits flipped and created white and dark patches in the image, razing the WPSNR and ruining the detection by up to 50/60%.

Underflow/overflow problems

We did not manage to fully fix the problem in time, and had to resort to approximations and a reduction in the α range for the day of the competition.

Only a small subset $[0.4, 0.25]$ of the α managed to recover 100% of the watermark without causing underflow/overflow problems. A range too low, that left us vulnerable to localized and resizing attacks.

With enough time, the problem could have been fixed by choosing a different DCT/pixel approach (e.g. working in the $-1/+1$ domain instead of the 0/255 one)

Underflow/overflow problems

We did not manage to fully fix the problem in time, and had to resort to approximations and a reduction in the α range for the day of the competition.

Only a small subset $[0.4, 0.25]$ of the α managed to recover 100% of the watermark without causing underflow/overflow problems. A range too low, that left us vulnerable to localized and resizing attacks.

With enough time, the problem could have been fixed by choosing a different DCT/pixel approach (e.g. working in the $-1/+1$ domain instead of the 0/255 one)

Underflow/overflow problems

We did not manage to fully fix the problem in time, and had to resort to approximations and a reduction in the α range for the day of the competition.

Only a small subset $[0.4, 0.25]$ of the α managed to recover 100% of the watermark without causing underflow/overflow problems. A range too low, that left us vulnerable to localized and resizing attacks.

With enough time, the problem could have been fixed by choosing a different DCT/pixel approach (e.g. working in the $-1/+1$ domain instead of the 0/255 one)

Possible improvements

What could be fixed or improved?

- Add heuristics to `texture_areas` (e.g. randomizing the search, with a seed based on the hash of the image)
- Increase the block size $\Rightarrow 16 \times 16$
- Use machine learning to identify texture areas instead of metric-based deterministic function: CNN are great at finding patterns in images
- Employ smarter ways to optimize parameters instead of brute force

Possible improvements

What could be fixed or improved?

- Add heuristics to `texture_areas` (e.g. randomizing the search, with a seed based on the hash of the image)
- Increase the block size $\Rightarrow 16 \times 16$
- Use machine learning to identify texture areas instead of metric-based deterministic function: CNN are great at finding patterns in images
- Employ smarter ways to optimize parameters instead of brute force

Possible improvements

What could be fixed or improved?

- Add heuristics to `texture_areas` (e.g. randomizing the search, with a seed based on the hash of the image)
- Increase the block size $\Rightarrow 16 \times 16$
- Use machine learning to identify texture areas instead of metric-based deterministic function: CNN are great at finding patterns in images
- Employ smarter ways to optimize parameters instead of brute force

Possible improvements

What could be fixed or improved?

- Add heuristics to `texture_areas` (e.g. randomizing the search, with a seed based on the hash of the image)
- Increase the block size $\Rightarrow 16 \times 16$
- Use machine learning to identify texture areas instead of metric-based deterministic function: CNN are great at finding patterns in images
- Employ smarter ways to optimize parameters instead of brute force

Possible improvements

What could be fixed or improved?

- Add heuristics to `texture_areas` (e.g. randomizing the search, with a seed based on the hash of the image)
- Increase the block size $\Rightarrow 16 \times 16$
- Use machine learning to identify texture areas instead of metric-based deterministic function: CNN are great at finding patterns in images
- Employ smarter ways to optimize parameters instead of brute force

Summary

Wavelet embedding

We experimented with wavelet embedding [1]. The embedding changes the coefficients on the HH and HL sub-band of the wavelets, where each coefficient gets the value:

$$x_{ij}^* = x_{ij} + \alpha m_i$$

The extraction function is just the inverse of the embedding function.

$$m_i = (x_{ij}^* - x_{ij})/(x_{ij}\alpha)$$

[1] Improved Wavelet-Based Watermarking Through Pixel-Wise Masking

Wavelet embedding

We experimented with wavelet embedding [1]. The embedding changes the coefficients on the HH and HL sub-band of the wavelets, where each coefficient gets the value:

$$x_{ij}^* = x_{ij} + \alpha m_i$$

The extraction function is just the inverse of the embedding function.

$$m_i = (x_{ij}^* - x_{ij})/(x_{ij}\alpha)$$

[1] Improved Wavelet-Based Watermarking Through Pixel-Wise Masking

Embedding and detection

Both procedures are quite simple.

For the embedding:

1. Extract DWT from image
2. Embed contents of the mark into HL and HH subbands of the image
3. Re-convert the image from DWT to pixel domain

For the detection:

1. Bring image into DWT domain
2. Use inverse of embedding function to extract the mark components

Embedding and detection

Both procedures are quite simple.

For the embedding:

1. Extract DWT from image
2. Embed contents of the mark into HL and HH subbands of the image
3. Re-convert the image from DWT to pixel domain

For the detection:

1. Bring image into DWT domain
2. Use inverse of embedding function to extract the mark components

Embedding and detection

Both procedures are quite simple.

For the embedding:

1. Extract DWT from image
2. Embed contents of the mark into HL and HH subbands of the image
3. Re-convert the image from DWT to pixel domain

For the detection:

1. Bring image into DWT domain
2. Use inverse of embedding function to extract the mark components

Mark visibility

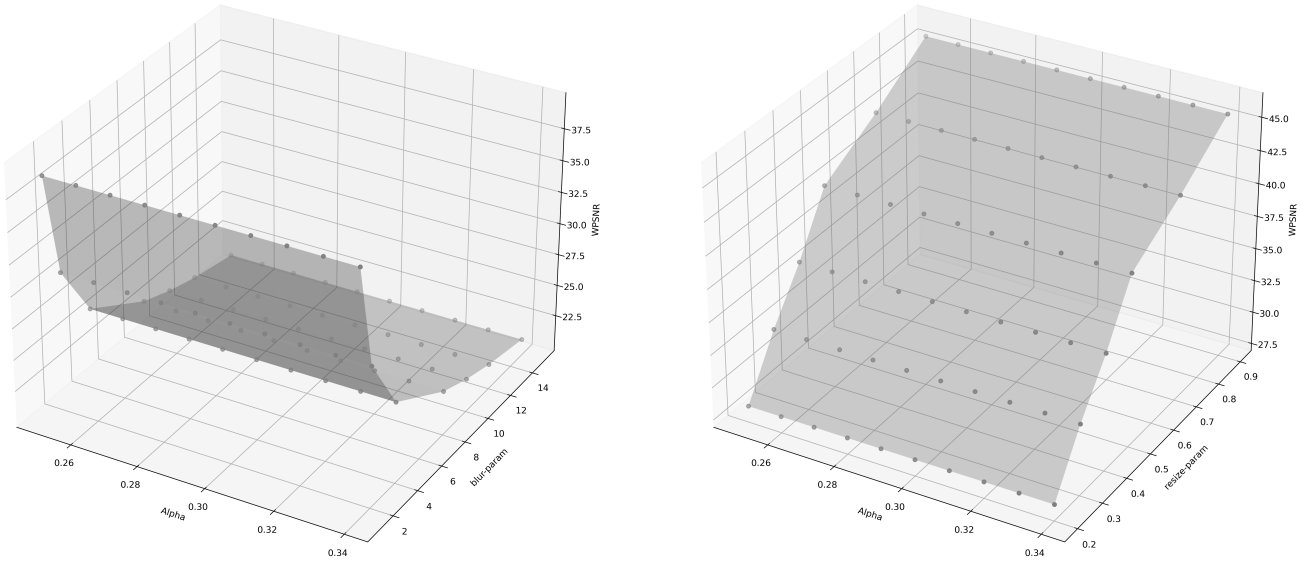
Due to the simple implementation, the performance is not very good. The mark components are not weighted on the image contents, thus the mark is quite visible even with low α values (note "sawtooth pattern" on oblique lines).

(a) (b)

Figure: Detail of original (a) and watermarked image (b)

Performance

As expected by the simple embedding function, the defense against attacks is quite poor:



(a) (b)

Figure: Plots of relation between WPSNR and attacks intensity

Conclusions

- The mark is embedded with the same intensity on all components of the image, so is quite vulnerable to attacks and is visible even with low alphas
- Since we were short on time, we dropped this approach and kept the block-based embedding

Conclusions

- The mark is embedded with the same intensity on all components of the image, so is quite vulnerable to attacks and is visible even with low alphas
- Since we were short on time, we dropped this approach and kept the block-based embedding

Summary

Attack strategy

We used a in-house interactive CLI program that mapped each pyc file to the each group image and allowed automated chain attacks.

```
› python3 attacker.py
Choose the image to attack:
0: buildings; 1: rollercoaster; 2: tree; 0
G:0: duebrioches; 1: eagleview; 2: hideseekers; 3: mario; 4: mastersof
entropy; 5: nonsonoprompt; 6: panebbianco; 7: paradisepark; 8: pythoni
zed; 9: thelenasboys; 10: unitrentomag; 11: watermarkcharmer; 0
A:0: awgn; 1: blur; 2: jpeg_compression; 3: median; 4: resizing; 5: sh
arpening; 2
⟹ QF = ____ [99, 98, 97, ... , 75, ... ]
99
```

Figure: A snippet of the attacker tool.

Attack strategy

Some groups had very resistant watermarking, yet detectable by the eye.

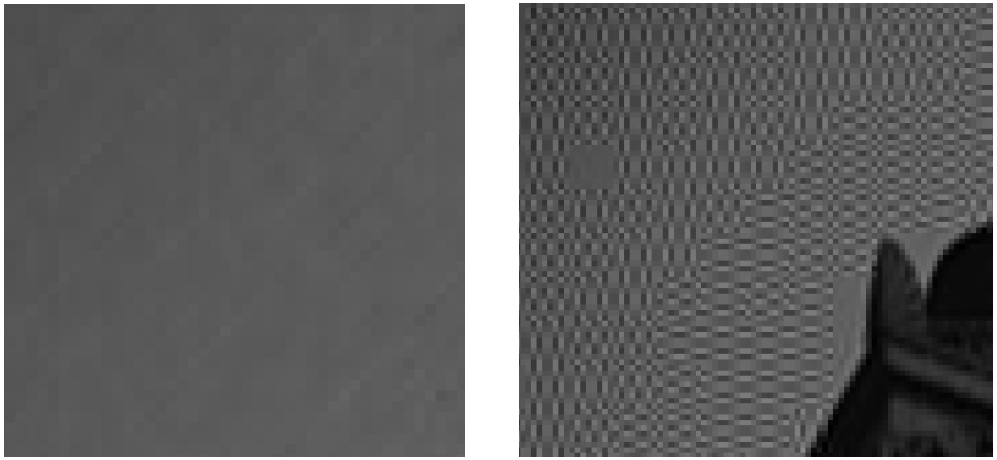


Figure: Example of very visible watermark implementations by groups ~~redacted~~ and ~~redacted~~

These edge cases allowed us to narrow down possible attacks and focus on those who better affected that particular image (e.g. `median` for the right picture)

Attack strategy

Some groups had very resistant watermarking, yet detectable by the eye.

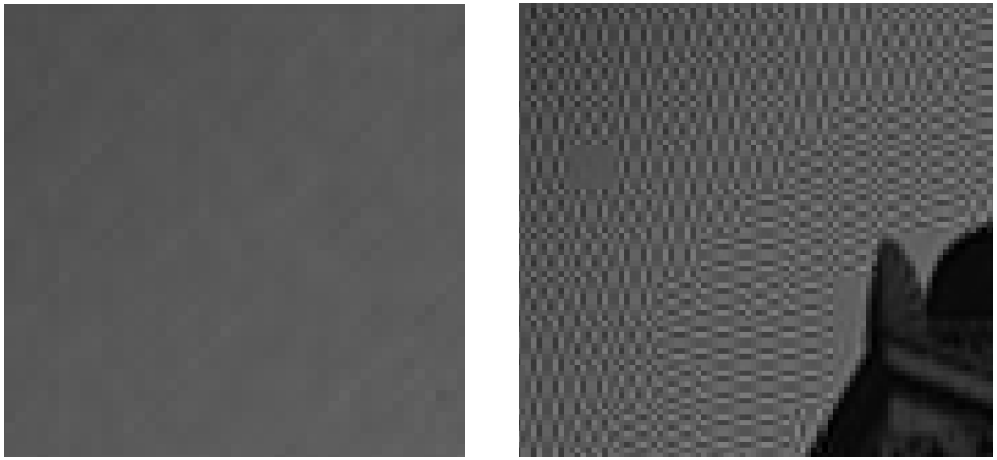


Figure: Example of very visible watermark implementations by groups ~~redacted~~ and ~~redacted~~

These edge cases allowed us to narrow down possible attacks and focus on those who better affected that particular image (e.g. `median` for the right picture)

Attack strategy

Some groups had very resistant watermarking, yet detectable by the eye.

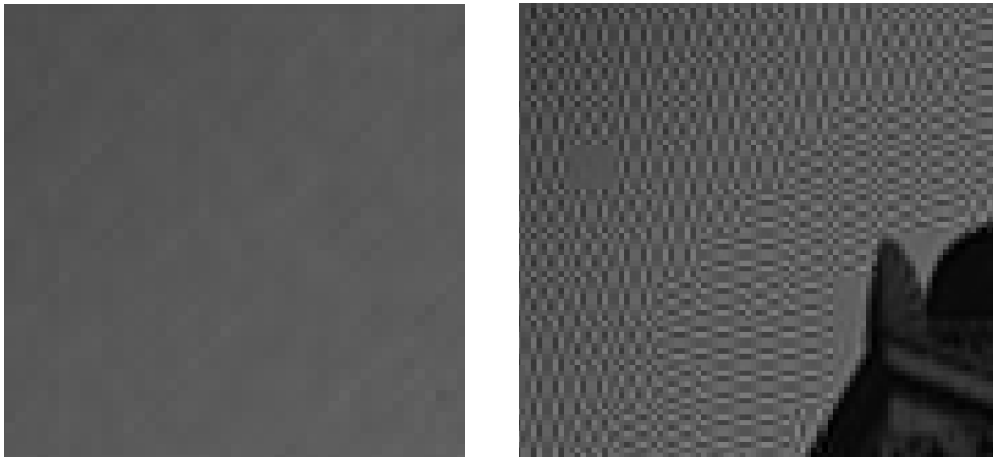


Figure: Example of very visible watermark implementations by groups ~~redacted~~ and ~~redacted~~

These edge cases allowed us to narrow down possible attacks and focus on those who better affected that particular image (e.g. `median` for the right picture)

Attack strategy

We managed to attack all groups in a rather short amount time, and this allowed to focus us on more resistant ones, in which we manually chained attacks and observed the behaviour in real-time in order to correct the strategy.

Interestingly enough, a number of pictures were easily destroyable with minimal amounts of resizing attacks (sometimes even scaling it by just 95%)!

For some of the images maybe localized attacks were ideal: we did try them but with little to no success.

Attack strategy

We managed to attack all groups in a rather short amount time, and this allowed to focus us on more resistant ones, in which we manually chained attacks and observed the behaviour in real-time in order to correct the strategy.

Interestingly enough, a number of pictures were easily destroyable with minimal amounts of resizing attacks (sometimes even scaling it by just 95%)!

For some of the images maybe localized attacks were ideal: we did try them but with little to no success.

Attack strategy

We managed to attack all groups in a rather short amount time, and this allowed to focus us on more resistant ones, in which we manually chained attacks and observed the behaviour in real-time in order to correct the strategy.

Interestingly enough, a number of pictures were easily destroyable with minimal amounts of resizing attacks (sometimes even scaling it by just 95%)!

For some of the images maybe localized attacks were ideal: we did try them but with little to no success.

Summary

Results

- **Invisibility**: visual quality of the image was neither excellent nor horrible; the final WPSNR achieve was in the $[50, 60]$ range for all images
- **Activity**: we managed to successfully attack all groups, albeit with mediocre results
- **Robustness**: underflow/overflow problems were a pain in the neck, making us suffer on this side
- **Attack quality**: neither excellent nor horrible; some of the techniques were pretty easy to crack, while others were pretty hard

Results

- **Invisibility**: visual quality of the image was neither excellent nor horrible; the final WPSNR achieve was in the $[50, 60]$ range for all images
- **Activity**: we managed to successfully attack all groups, albeit with mediocre results
- **Robustness**: underflow/overflow problems were a pain in the neck, making us suffer on this side
- **Attack quality**: neither excellent nor horrible; some of the techniques were pretty easy to crack, while others were pretty hard

Results

- **Invisibility**: visual quality of the image was neither excellent nor horrible; the final WPSNR achieve was in the $[50, 60]$ range for all images
- **Activity**: we managed to successfully attack all groups, albeit with mediocre results
- **Robustness**: underflow/overflow problems were a pain in the neck, making us suffer on this side
- **Attack quality**: neither excellent nor horrible; some of the techniques were pretty easy to crack, while others were pretty hard

Results

- **Invisibility**: visual quality of the image was neither excellent nor horrible; the final WPSNR achieve was in the $[50, 60]$ range for all images
- **Activity**: we managed to successfully attack all groups, albeit with mediocre results
- **Robustness**: underflow/overflow problems were a pain in the neck, making us suffer on this side
- **Attack quality**: neither excellent nor horrible; some of the techniques were pretty easy to crack, while others were pretty hard