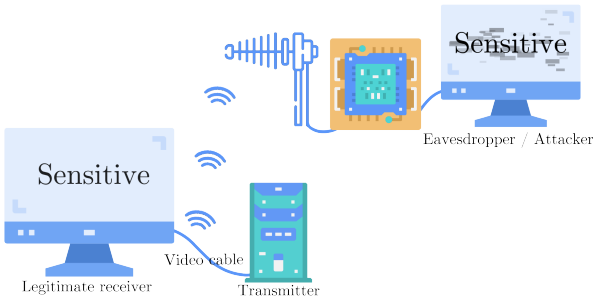
Systémes Numériques 3^{rd} year - SNum-3
2024 - 2025

Hardware Security

TEMPEST attack: Eavesdrop of screen



Robin Gerzaguet

ENSSAT - Université de Rennes
robin.gerzaguet@enssat.fr
6 Rue de Kerampont - CS 80518
22305 LANNION - France

- https://gitlab.enssat.fr/rgerzagu/TempestSDR_runtimeApplication

École Nationale Supérieure de Sciences Appliquées et de Technologie
Université de Rennes
Technopole Anticipa • Lannion

Contents

Chapter 1

Positioning and objective of the lab

1.1 Context

The objective of the lab is to realize an eavesdrop of a screen connected to a remote computer.

An Eavesdrop ?

An eavesdrop is a real life use of a hardware (or software) vulnerability. It may correspond to the passive use of a vulnerability (our case here, as we will process a leakage that is emitted naturally from a device) or an active use of a vulnerability (force the emission of a signal).

We will consider a screen eavesdrop. When a PC is connected to a screen with a cable, it is possible that the RGB stream is emitted though an electromagnetic wave as the cable between the PC and the screen may act as an antenna. It this kind of scenario, the carrier frequency of the leak has to be found, and the received signal has to be post-processing to re-create an image.

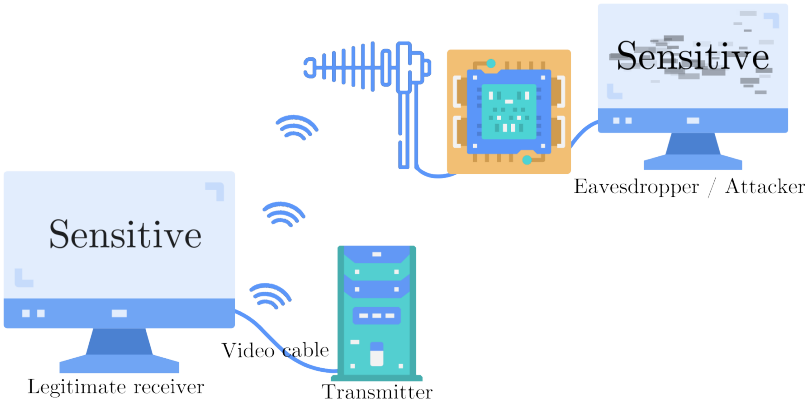


Figure 1.1: Considered eavesdrop

1.2 Setup of the eavesdrop

The setup of the eavesdrop will be as follows **First the legitimate part**

- A transmitter which will be the PC (here a laptop)

- A legitimate receiver which will be the external screen
- An HDMI cable that connects the screen to the PC. DVI connection will also work.

This corresponds to the entity that will be attacked. Nothing particular here, just a classic scenario where you have a PC and a screen.
Then the attacker

- A software Defined Radio (SDR) that will passively listen to the electromagnetic emanation and get the signal with the vulnerability
- An antenna connected to the SDR that receive (and amplify) the signal at the desired carrier frequency. It can greatly increase the quality (and thus the range) of the eavesdrop.
- A processing PC that will convert this electromagnetic stream into 2D images.

1.3 Steps for screen recovering

Using this kind of eavesdrop requires several processing steps that are summarized here

1. Find the carrier frequency of the leak and configure the radio to observe at this frequency
2. Stores the signal with the screen information
3. Find the configuration of the eavesdropped screen (screen size, screen resolution)
4. Re-create an image of the screen
5. Post process this image to have a proper render
    - Synchronize the image to have the top left corner at its appropriate location
    - Filtering the image to reduce the noise

In this lab, we will do all these steps, in a particular manner

- Step 1 and 2 will be done during the lab but not necessary at the beginning. We will use the proposed application to analyse the correlation of the signal and stores the IQ signal. Each group will have is own acquisition so depending on the order you can do this task very soon in the lab (or not)
- Step 3 and 4 will be first done on a signal that have already been captured. We are sure there is a compromission there and the goal is to be able to generate the image of the screen, in gray scale
- Step 5 will be done all other steps and will use some basic (and more advanced) image processing techniques to improve the quality of the renderer screen.

► **First read Chapter 2 !**

1.4 Lab assets and evaluation

The project will be done in Python language.
In the repo you have cloned, you have

- A bash script `installTempest.sh` that install and initialize the graphical application that will be used for signal recovering.
- A bash script `launchGUI.sh` that launches the application.
- A folder `src_tempest` will all the source code of the application (not necessary to do deep into there).

Evaluation process will be defined at the beginning of the lab !

Chapter 2

Eavesdrop capture using the given application

The purpose of this part is to get the signal that has potentially a vulnerability. To do so, we will first become familiar with what an eavesdrop is and how we can detect one.

2.1 A small historical perspective

This lab has been intensively inspired by the work of Martin Arimov [Mar] and Markus Khun [Kuh] Computer monitors started to become widespread in the end of the 20th century. Wim van Eck published the first unclassified technical analysis of the security risks of emanations from computer monitors in 1985[Eck]. The next important publication regarding video emanations came from Markus Kuhn nearly 20 years later in 2003 [Kuh]. Later in 2013, Elibol, Sarac and Erer demonstrated that emanations could be picked up from considerable distance with a mobile and lower cost equipment [ESE].

Note that screen eavesdrop is only a part of the potential TEMPEST attacks and there are plenty of different scenarios and use-cases. Have a close look on [LGG+] for a deeper analysis of the side channels, and especially the electromagnetic ones.

2.2 Capture and find a eavesdrop !

Please read Appendix 4.2

In this part, you will try to find a real eavesdrop. You have to do this part with the teacher with your SDR not that far from the screen that leaks.

Small reminder

The SDR you use are fragile and are very low cost SDR. Besides the antenna you have no gains. It means that it can be tricky to have a very good image capture with strong resolution.
Anyway, be able to distinguish an image is a very good sign you have discover a compromission !

1. Be sure you have a proper setup for the compromission
    - The SDR plugged into the PC and its antenna pointing to the attacked screen
    - The `config.jl` file modified to use the SDR instead of a stored signal (line 13, replace `:radiosim` by `:pluto`)
    - The GUI launched with the command `./launchGUI.sh`
2. Be sure the SDR bandwidth is set to 20MHz, otherwise the capture band will be too small to be able to reconstruct the image.
3. We have to find a potential carrier frequency for the leakage. To do so, we have to find a relationship between the DVI screen connection and electro-magnetic leakage. Have a look on the scientific paper that studies the EM leakage of a DVI connector "EMI Analysis of DVI Link Connectors" [PDZ+]. You can have a try on the different frequencies that can bear a leakage

Harmonics and intermodulation

You have observed a leakage linked to a harmonic of the main DVI clock. In practice, the leakage can be due to 2 different phenomena

(a) A direct harmonic, e.g a multiple of the clock frequency

(b) An intermodulation product. In this case, the leakage is not a multiple of the frequency but rather due to an internal mixing of two frequencies. This process is the basis of modulation is depicted on Figure 2.1

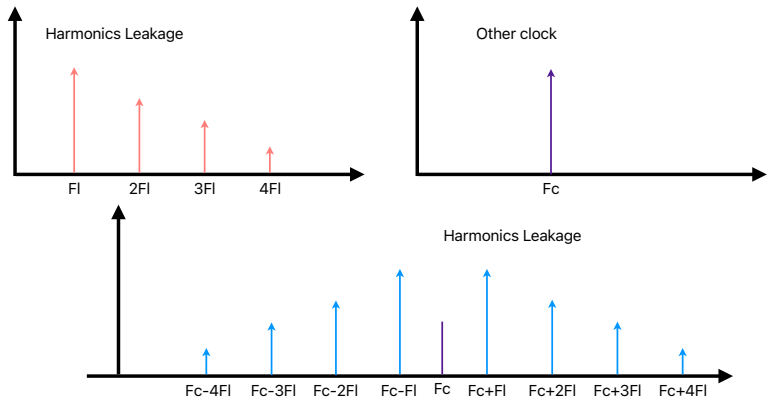


Figure 2.1: Scheme of the intermodulation process

4. There is also in the DVI controller an internal clock of 20MHz. Deduce the location of another potential leakage.
5. Choose the frequency where you have the best leakage. Store the signal if you have a readable image.

At this end of this part

You should have

- Understand how the leakage can appear
- Find a carrier frequency with potential leakages
- Explore the correlations to deduce image configurations
- Find a real eavesdrop and store a signal with an embedded screen image !

Chapter 3

Reconstruct a Grayscale image

In this part, you can use two different signals

- The one that is given in the project
- The one you have stored from your practical eavesdrop. Note that for the latter case, you should have seen the screen image on the image panel of the GUI.

The processing chain we will create is highly dependent on the nature of the eavesdrop. **Be sure you have read the Appendix 4.1.**
The purpose of this part is to decompose, understand and develop the processing bricks that are mandatory to display a grayscale image from the recovered EM signal.
The work will be done on a Python notebook. The very first thing to do is to load the python dependencies that will be used through the project.

```
from IPython.display import clear_output
import numpy as np
import scipy.signal as sg
from scipy.fft import fft, ifft
import matplotlib.pyplot as plt
from PIL import Image
```

3.1 Loading a complex signal and analyse it

1. First load the signal in the python environment.

   ```
   # Load the binary file
   sig = np.fromfile('./src_tempest/TempestSDR/dumpIQ_0.dat', ...
       dtype=np.csingle)
   ```

   The `csingle` parameter states that the SDR stores signal as a complex one (with Float32 format). Why the signal is complex ?

2. The leakage implies that the RGB is modulated using an amplitude modulation. Writer the python line that demodulates the signal. As a reminder, an amplitude modulation implies that the information is not on the phase of the signal but only on its envelop (i.e modulus).

3. If no leakage were here, we should have only stores noise. How should be the signal spectrum if only noise were there ? You can find below the python skeleton of a simple power spectral density estimator. This function should be applied on the complex signal.

```
def mag2db(sig):
    """ Simple magnitude to Decibel convertion using array ...
        comprehension
    """
    return [10*math.log10(x) for x in sig]
# It can be interessting to define a FFT method
def spectralAnalysis(sig):
    """ Simple spectral analysis based on Wiener-Kinchine ...
        theorem
    """
    # PSD calculation
    c = sg.correlate(sig, sig, mode='same') # ...
        Autocorrelation of the input signal
    y = np.abs(fft(c))**2 # square modulus of the fourier ...
        transform of the autocorr
    Y = mag2db(y) # In dB
    # PSD plot
    fig, ax = plt.subplots()
    plt.plot(Y)
    ax.set_xlabel('Lag in seconds')
    ax.set_ylabel('Correlation magnitude')
```

At this end of this part

You should have

- A functional Python environment
- The visualisation of the signal you have chosen (the default one or the one you have stored in the experiment)

3.2 Find the screen rate

As you know if you have carefully read the appendix 4.1, the screen rate is dependent on the autocorrelation of the signal.

1. Define the sampling frequency you have used as `Fs = 20e6`.
2. Calculate the autocorrelation of the input signal. We will compute the circular autocorrelation function expressed as

$$\Gamma[n] = \mathscr{F}^{-1}\left(X(n) \times X^*(n)\right) \tag{3.1}$$

where $\mathscr{F}^{-1}$ is the inverse Fourier transform operator, X is the discrete Fourier transform of the signal and * denotes the complex conjugates. Implement a Python function that does that.

3. Plot the result. You should see groups of peaks. The main one is associated to the energy while periodic peaks appears.

4. The autocorrelation is computed for different lags. Calculate the lags indexes and plot the correlation with respect to these lags, expressed in milliseconds.

5. As you see there are a lot of points here and we want to zoom on the appropriate area. This zoomed area should be such that we observe refresh rates between 40Hz and 100Hz. Calculate the delay associated to these 2 extrema and extract the subvector of the autocorrelation associated to this desired area.

At this end of this part

You should have

- Find the screen refresh rate !

3.3 Find the screen configuration

A screen is defined by its refresh rate but also by the size of the screen.

1. We have to zoom around the position of the max. The x axis should be expressed as a timing in second, and we will find the link between the sample and the line after. You can select a zoom between 0 (the position of the max) and $40\mu s$.

2. While the first max corresponds to the repetition of image, the second maximum corresponds to the repetition of the line (more precisely the blanking area between lines). Select and measure the timing associated to this value.

3. This timing is the duration of a line. What we want is to convert this timing in a number of pixels. Based on the formula given by Marinov page 40 [Mar] we have

$$y_t = \frac{1}{t_{\text{peak}} \times f_v} \tag{3.2}$$

   Find the value of y_t.

4. The value of x_t is unknown and cannot be fund easily. We will have to find the value based on a dictionary of configuration.

```
# Defining the Class
class VideoMode:
   def __init__(self, x_t, y_t, refresh_rate):
      self.refresh_rate = refresh_rate
      self.x_t = x_t
      self.y_t = y_t
# Create the dictionnary with all the supported configurations
video_modes = {
   "640x400 @ 85Hz": VideoMode(832, 445, 85),
   "720x400 @ 85Hz": VideoMode(936, 446, 85),
   "640x480 @ 60Hz": VideoMode(800, 525, 60),
```

```
    "640x480 @ 100Hz": VideoMode(848, 509, 100),
    "640x480 @ 72Hz": VideoMode(832, 520, 72),
    "640x480 @ 75Hz": VideoMode(840, 500, 75),
    "640x480 @ 85Hz": VideoMode(832, 509, 85),
    "768x576 @ 60 Hz": VideoMode(976, 597, 60),
    "768x576 @ 72 Hz": VideoMode(992, 601, 72),
    "768x576 @ 75 Hz": VideoMode(1008, 602, 75),
    "768x576 @ 85 Hz": VideoMode(1008, 605, 85),
    "768x576 @ 100 Hz": VideoMode(1024, 611, 100),
    "800x600 @ 56Hz": VideoMode(1024, 625, 56),
    "800x600 @ 60Hz": VideoMode(1056, 628, 60),
    "800x600 @ 72Hz": VideoMode(1040, 666, 72),
    "800x600 @ 75Hz": VideoMode(1056, 625, 75),
    "800x600 @ 85Hz": VideoMode(1048, 631, 85),
    "800x600 @ 100Hz": VideoMode(1072, 636, 100),
    "1024x600 @ 60 Hz": VideoMode(1312, 622, 60),
    "1024x768i @ 43Hz": VideoMode(1264, 817, 43),
    "1024x768 @ 60Hz": VideoMode(1344, 806, 60),
    "1024x768 @ 70Hz": VideoMode(1328, 806, 70),
    "1024x768 @ 75Hz": VideoMode(1312, 800, 75),
    "1024x768 @ 85Hz": VideoMode(1376, 808, 85),
    "1024x768 @ 100Hz": VideoMode(1392, 814, 100),
    "1024x768 @ 120Hz": VideoMode(1408, 823, 120),
    "1152x864 @ 60Hz": VideoMode(1520, 895, 60),
    "1152x864 @ 75Hz" : VideoMode( 1600, 900 , 75),
    "1152x864 @ 85Hz" :VideoMode( 1552, 907 , 85),
    "1152x864 @ 100Hz" :VideoMode( 1568, 915 , 100),
    "1280x768 @ 60 Hz" :VideoMode( 1680, 795 , 60),
    "1280x800 @ 60 Hz" :VideoMode( 1680, 828 , 60),
    "1280x960 @ 60Hz" :VideoMode( 1800, 1000, 60),
    "1280x960 @ 75Hz" :VideoMode( 1728, 1002, 75),
    "1280x960 @ 85Hz" :VideoMode( 1728, 1011, 85),
    "1280x960 @ 100Hz" :VideoMode( 1760, 1017, 100),
    "1280x1024 @ 60Hz" :VideoMode( 1688, 1066, 60),
    "1280x1024 @ 75Hz" :VideoMode( 1688, 1066, 75),
    "1280x1024 @ 85Hz" :VideoMode( 1728, 1072, 85),
    "1280x1024 @ 100Hz":VideoMode( 1760, 1085, 100),
    "1280x1024 @ 120Hz":VideoMode( 1776, 1097, 120),
    "1368x768 @ 60 Hz" :VideoMode( 1800, 795 , 60),
    "1400x1050 @ 60Hz" :VideoMode( 1880, 1082, 60),
    "1400x1050 @ 72 Hz":VideoMode( 1896, 1094, 72),
    "1400x1050 @ 75 Hz":VideoMode( 1896, 1096, 75),
    "1400x1050 @ 85 Hz":VideoMode( 1912, 1103, 85),
    "1400x1050 @ 100 Hz":VideoMode( 1928, 1112, 100),
    "1440x900 @ 60 Hz" :VideoMode( 1904, 932 , 60),
    "1440x1050 @ 60 Hz":VideoMode( 1936, 1087, 60),
    "1600x1000 @ 60Hz" :VideoMode( 2144, 1035, 60),
    "1600x1000 @ 75Hz" :VideoMode( 2160, 1044, 75),
    "1600x1000 @ 85Hz" :VideoMode( 2176, 1050, 85),
    "1600x1000 @ 100Hz":VideoMode( 2192, 1059, 100),
    "1600x1024 @ 60Hz" :VideoMode( 2144, 1060, 60),
    "1600x1024 @ 75Hz" :VideoMode( 2176, 1069, 75),
    "1600x1024 @ 76Hz" :VideoMode( 2096, 1070, 76),
```

```
    "1600x1024 @ 85Hz" :VideoMode( 2176, 1075, 85),
    "1600x1200 @ 60Hz" :VideoMode( 2160, 1250, 60),
    "1600x1200 @ 65Hz" :VideoMode( 2160, 1250, 65),
    "1600x1200 @ 70Hz" :VideoMode( 2160, 1250, 70),
    "1600x1200 @ 75Hz" :VideoMode( 2160, 1250, 75),
    "1600x1200 @ 85Hz" :VideoMode( 2160, 1250, 85),
    "1600x1200 @ 100 Hz":VideoMode( 2208, 1271, 100),
    "1680x1050 @ 60Hz (reduced blanking)":VideoMode( 1840, ...
       1080, 60),
    "1680x1050 @ 60Hz (non-interlaced)":VideoMode( 2240, ...
       1089, 60),
    "1680x1050 @ 60 Hz":VideoMode( 2256, 1087, 60),
    "1792x1344 @ 60Hz" :VideoMode( 2448, 1394, 60),
    "1792x1344 @ 75Hz" :VideoMode( 2456, 1417, 75),
    "1856x1392 @ 60Hz" :VideoMode( 2528, 1439, 60),
    "1856x1392 @ 75Hz" :VideoMode( 2560, 1500, 75),
    "1920x1080 @ 60Hz" :VideoMode( 2576, 1125, 60),
    "1920x1080 @ 75Hz" :VideoMode( 2608, 1126, 75),
    "1920x1200 @ 60Hz" :VideoMode( 2592, 1242, 60),
    "1920x1200 @ 75Hz" :VideoMode( 2624, 1253, 75),
    "1920x1440 @ 60Hz" :VideoMode( 2600, 1500, 60),
    "1920x1440 @ 75Hz" :VideoMode( 2640, 1500, 75),
    "1920x2400 @ 25Hz" :VideoMode( 2048, 2434, 25),
    "1920x2400 @ 30Hz" :VideoMode( 2044, 2434, 30),
    "2048x1536 @ 60Hz" :VideoMode( 2800, 1589, 60)
}
```

For this we have to write a function that returns the closest configuration. A configuration is close if the distance between the estimated y_t and one existing configuration is minimal. As there are many configurations that have identical y_t but different rates, we also have to check that we choose the configuration with the closest refresh rate.Â
You can find below a skeleton for such a function to be completed

```
def find_closest(dictionary, value, fv):
    """
    Find the key in a dictionary that has the closest value ...
       to the given value.
    """
    closest_key = None
    closest_conf = None
    closest_fv = None
    min_fv = float('inf')
    min_distance = float('inf')
    for key, val in dictionary.items():
        # Compute Euclidean distance between our y_t and the ...
           real from the configuration
        distance = ?????
        if distance < min_distance:
            # A new challenger for minimal gap between y_t and ...
               what we have ==> Update the chosen key and ...
               configuration, and the min_distance
            ????
```

```
        elif distance == min_distance:
            # If configuration is the same, overwrite if the ...
                rate is closer
            d_f = ???
            if d_f < min_fv:
                ?????
    return closest_key,closest_conf
```

5. Using the class and function described just before, find the best configuration. It should have

   - The value y_t you have found
   - The value f_v you have found
   - The value x_t based on the best found configuration

At this end of this part

You should have

- A complete screen configuration !

3.4 First Image rendering

We now have all the information to be able to reconstruct an image.

1. First, take a buffer that corresponds to one frame. Remember that the frame duration is linked to the refresh rate.

2. We have to convert this buffer into a 2D image. Note that we will only use arrays in this lab, but we can use tools and libraries from the large Python ecosystem such as Pillow. Below is the function that takes a vector as input and returns a matrix whose dimensions match the provided input parameters.

```
def resize_image(img, x_t, y_t):
    """ Resize an array into a 2D array, using PIL """
    # Switch to image to use PIL
    im = Image.fromarray(img);
    # Resize the image based on the configuration and cast ...
        as np.array
    img_resized = np.asarray(im.resize(1,x_t* y_t))
    img_reshape = np.reshape(img_resized, (y_t, x_t))
    return img_reshape
```

3. Display the 2D image and comment on the result.

Figure size

In a Python notebook, you can control the size of the image when importing the Python package.

```
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 8]
```

4. Complete the code to have an interactive plot of the complete signal acquisition.

```
def update_plot():
    for n in range(nbImage):
        # --- Processing to have an image
        # Returns an image that corresponds to the current ...
            processing block
        # --- Routine to have the interactive display
        clear_output(wait=True)
        plt.imshow(img, cmap='gray')
        plt.show()
    print("done")
update_plot()
```

Note that the given signal is quite short, so do not hesitate to use your longer acquisition.

5. The image can be a little blurry due to additive noise. One way to reduce this noise is to filter the image. As proposed in the original work of [Mar], we can perform a first-order IIR filter. This filter computes an average image based on the following equation:

$$y[i,j] = \alpha \cdot y[i,j] + (1-\alpha) \cdot x_k[i,j] \tag{3.3}$$

where $[i,j]$ is the pixel position in the image, x_k is the kth blurred image, and y is the filtered image. α is the step-size of the IIR filter, ranging from 0 to 1. The closer α is to 1, the more the image is averaged: the extreme values are 0 (no filtering) and 1 (the image is fixed to y). Implement an `iirfilter` method and apply it.

At the end of this part

You should have:

- Seen your first eavesdrop!
- An unsynchronized image.

3.5 Synchronisation algorithm

A key issue we have when recreating the image is that the image is not perfectly aligned as we do not synchronize the beginning of the image with the frame we process. For instance if you select a frame with the sample delay 420000 the reconstructed image is as the one depicted on Figure 3.1.

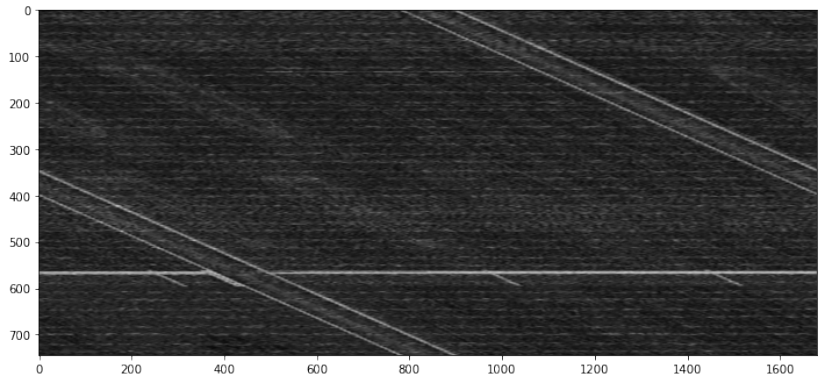


Figure 3.1: Reconstructed image without proper synchronisation

You can observe the black lines that corresponds to both vertical and horizontal lines. This is a key behavior of the way image are transmitted using the HDMI interface and that is a legacy from older screen transmission standard. Although CRT screens are now considered obsolete, they have laid the foundations for modern screens that still use some standards created for CRT monitors. Since a CRT screen does not store any data, a continuous stream of data is used. However, as screens have different resolutions, control signals must be added for line and image synchronizations, labeled *hsync* and *vsync* in Fig. 3.2. A gap interval labeled *vblank* (vertical blanking interval) can be shown in Fig. 3.2 between the end of the *vsync* pulse and the start of a new image frame. During this interval, no image is displayed but data are still sent to the pixel lines all with the same value. There is also a small horizontal blanking interval after the *hsync* pulse for the same reason. Modern CRT screens do not require such long blanking intervals, and LCD displays require none, but the standards were established when the blanking was needed and thus still remain.

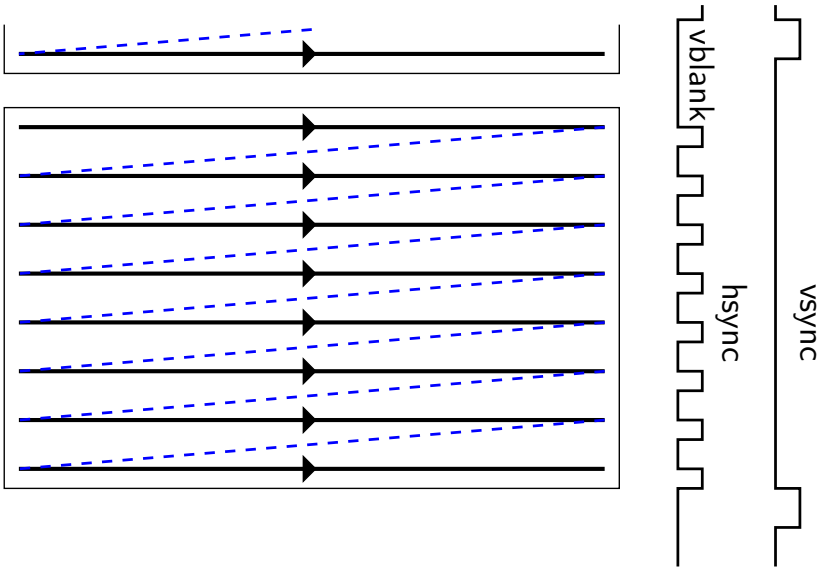


Figure 3.2: CRT rendering strategies

In these blanking interval, the value of the pixel is maintained constant which can be used to find

the location. The complex part here is that both the position of the blank **and** the size of the blank is unknown.
The purpose of the algorithm you will have to write is twofold

- If the image is tilted, you can change the value of y_t around the vlaue of have chosen to have a non-tilted image.
- Find the precise location of the start of the image, in both x and y scale. This position is marked for the previous image on the left hand corner by the red circle on Figure 3.3.
- Modify the image rendering to start by the synchronized area. This can be done by shifting the matrix. This corresponds to the yellow arrow of Figure 3.3.

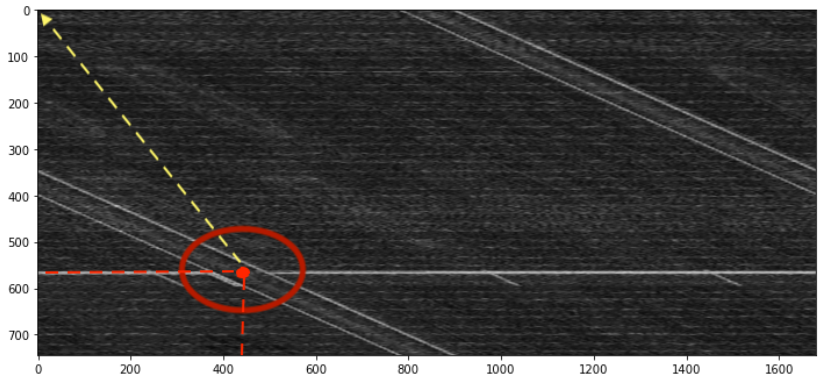


Figure 3.3: Reconstructed image without proper synchronisation. The synchronisation position is marked by the red circle.

The problem has **two dimensions** but these two dimensions are independent: the synchronized pixel x_{sync} for the axis x is independent of the synchronisation pixel y_{sync} for the axis y. In the latter image, the synchronisation position ($x$sync, $y$sync) is (420,580) It means that the problem we have is not "one problem with 2 dimensions" but rather "2 problems of one dimensions". We will consider independently the x and y synchronisations and considering a vector problem and not a matrix problem.
We will use the similar approach to the one exposed by Marinov in his manuscript [Mar]. An illustration from the manuscript is given on Figure 3.4.
We need to introduce the following metrics

- n is the size of the dimension expressed as a number of pixels. If the resolution is (1680,744), then n is equal to 1680 for the x synchronisation and 744 for the y synchronisation process.
- c is the current position of the pixel that is evaluated to be the best one. We have $c \in 0; N-1$.
- w is the half duration of the blank. It means that the image will have a constant (and high) pixel value for 2w pixels.

Remember that both the position of the sync and the size of the blank are unknown. The algorithm has both to find $c$sync and $w$sync the **optimal** candidates for the synchronisation. Optimal does not make any sense if we do not consider a metric: the best candidates are optimal with respect

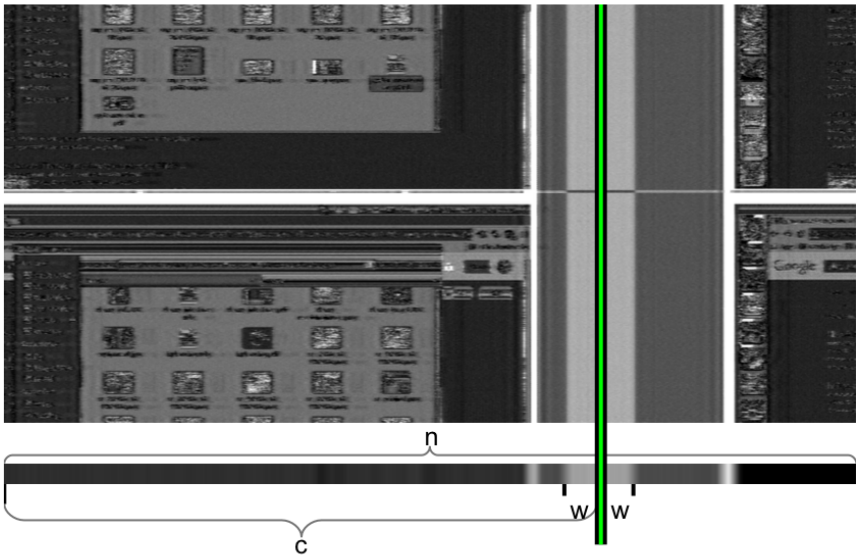


Figure 5.5: Visualising the relationship between c, w and n. The figure shows the values for w and c that give the maximum value of β for the frame.

Figure 3.4: Projection and metrics used in the synchronisation algorithm.

to a cost function. We will have to find the couple ($c$sync, $w$sync) that **maximises** a chosen cost function.
This cost function is related to what we want to do. As the transition between images corresponds to constant high pixel value, we will calculate an average pixel value, positioned at pixel c and on a range of $2w$ pixels. Note that this metric is simple to calculate in the middle of the image, but it can be tricky on the edge. This is the reason why we will consider a circular image where edge are connected to each other. The cost function is called β and is expressed as

$$\beta(w,c) = \left[\sum_{k=c-w}^{c+w} \frac{p[k \mod n]}{2w} - \sum_{k=2w-c}^{2(n-w)-c} \frac{p[k \mod n]}{2n-4w} \right]^2 \tag{3.4}$$

with p the vector associated to the projection, which is of size n. β is thus a matrix, evaluated for each possible blank window and each possible pixel location.
The algorithm is performed independently for both x and y axis and is done in several steps.

1. Project the matrix in a one dimension vector by summing the component
2. Filtering this vector with a Kernel filter. This filter is a FIR filter of size L (with L odd) and defined as

$$\alpha = \frac{L-1}{2} \tag{3.5}$$

$$h[k] = exp\left(\frac{-2k^2}{L^2}\right) \text{for k } \in [-\alpha;\alpha] \tag{3.6}$$

You can use the Python function `np.convolve` to perform the convolution.

3. For each possible blank window and for each pixel calculate the cost function β that is based on the average pixel value. This cost function is computed for many different value of range interval w and for all possible positions on the scale of interest.

4. Find the maximum of the cost function. It gives both the position of the frame start, and the duration of the blank.

This algorithm is computational intensive and Marinov proposes to limit the intervals of search for c and w.

- Synchronisation region are not that small, and we know that w should be greater than 2% of n
- The blanking region is always smaller than the video region. This will put an upper bound for the value of $2w < \frac{n}{2}$.

We now have all the information to implement the synchronisation algorithm

1. Propose an interpretation of the cost function: why maximizing it leads corresponds to the start of frame position ?

2. Implement the synchronisation algorithm for the 2 dimension and apply it to find the synchronisation pixel ($x$sync, $y$sync).

3. Implement a shifting algorithm to ensure that the rendered image starts at ($x$sync, $y$sync)

4. An issue with the naive implementation of the algorithm is that is calculates a lot of intermediate pixel values that is used for many indexes of the cost function. Proposes a refinement of the algorithm to accelerate the synchronisation search. Using `timeit` module in Python, measure the timing of your naive algorithm and of the optimized one.

Chapter 4

Appendix

4.1 Signal Structure

The link between the screen and the PC creates an electromagnetic emanation that can be passively processed by a Software Defined Radio. This signal has some patterns and is constructed to be compatible with screens: we will exploit this symmetry in this signal to reconstruct the image.

- A screen corresponds to a matrix of pixels with x_t columns (the width of the image) and y_t lines (the height of the image). There are many possible configurations, but these two values cannot be arbitrary chosen.
- The screen displays f_r images per second. This is known as the frame rate, and it is typically between 30Hz and 120Hz. 60Hz is a very common value.

These three parameters are mandatory to reconstruct the image, and we have to find them based on the EM signature we have.

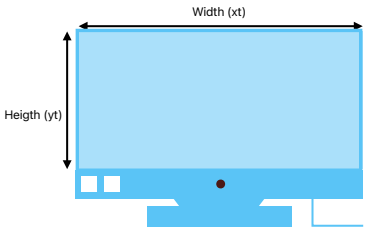


Figure 4.1: Screen dimension

The image will be serialized to be sent to the screen. It means that we will have all lines transmitted after the other. One image (we will call this a *frame*) will thus be transmitted in a duration T_f. To maintain the desired rate, a frame (i.e. all pixels of the image) has to be transmitted in a duration $1/f_r$. The pixel rate (duration of one pixel) is thus $x_t \times y_t \times f_r$. On the receiver side we will only get the gray scale of the image as we image is emitted through an *Amplitude modulation* techniques. It means that we have to apply the square values of the received signal (which is complex as we have done an IQ demodulation with the SDR)

A word on the SDR sampling rate

The SDR should capture the entire bandwidth of the screen emanation. Based on the configuration, it can be a lot and low cost SDR have a limited bandwidth. For good screen reconstruction, Marinov suggest to capture at least 8MHz bandwidth and shows better results with 20MHz. With the Pluto we will not be able to stream contiguously, but we will be able to capture most of the signal.

Old screens were designed as CRT (cathode-ray tube) and it was a vacuum tube containing one or more electron guns, which emit electron beams that are manipulated to display images. It means that transient (or silent) times were required between the lines and the columns to give the beam enough time to realign itself. It will help us to synchronize the beams in the Part III !

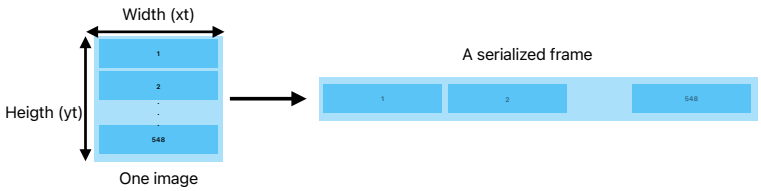


Figure 4.2: Serialization of the image

An image in a screen does not change that often. It means that consecutive frames will often have very similar content. We will be able to find f_r by looking at the correlation of the signal. We should have a peak associated to the frame duration and thus deduce the rate.

2 consecutive lines of an image also does not change that many. By zooming into the correlation, we should also find peaks associated to the number of lines and deduce the value of y_t. Based on this we will deduce the value of x_t as we have a limited number of configurations.

Signal signature

We have to look into the autocorrelation of the given signal. If we have many peaks, it can mean that we have an image inside the signal !

4.2 Graphical User Interface

The Graphical User Interface is presented on Figure 4.3. It shows the correlation plot, the renderer image and the control panel.

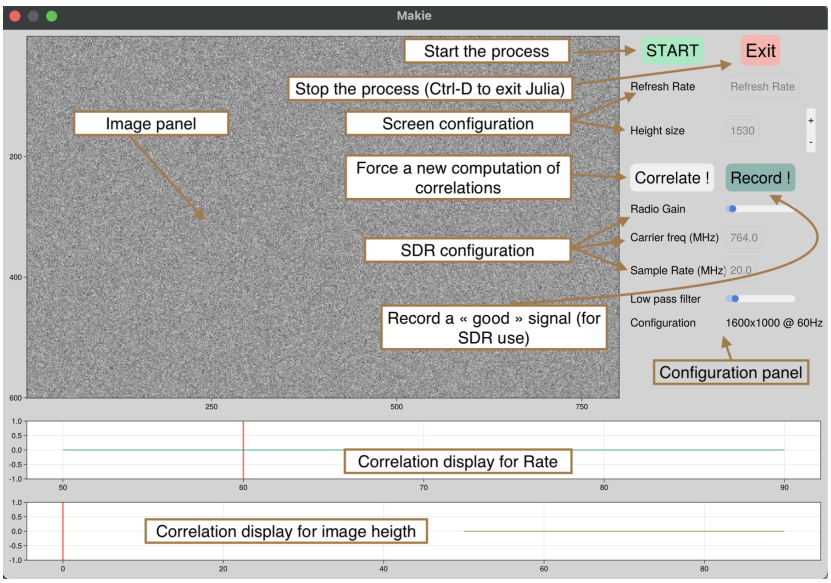


Figure 4.3: GUI when it is open. The different panes are described.

The configuration of the GUI is defined in the file `config.jl`. In particular, we can

- Use a file that have been stored to find the compromission. This is the `:radiodsim` mode (note the : before radiosim).
- Directly use an SDR. As we have Pluto from Analog devices, the GUI can interface the radio by setting `:pluto` on the SDR parameter.

Then in a terminal, simply run `./launchGUI.sh`. The GUI should appear after a few seconds.

- The image and the correlations will appear when you click on `start`. If the configuration is not properly set you will only see noise !
- Correlations are interactive. By clicking on the different location you will update the video configuration (based on the position of the max). Find the configuration that leads to a good image !
- First select the best rate. Do not hesitate to zoom to select the best peak probably around 60Hz.
- Then in the "correlation for image height" zoom to select also the maximum. These values are used to find the screen configuration and should give an image in the "image panel".
  Note that there are some uncertainty when calculating the parameters (Marinov explains a lot those in his thesis [Mar], in section 3.4.4). It means that when you have selected the maximum value (which leads to a certain value for y_t you should move around few pixels to get a good image Use the + and the - in the screen configuration panel.

After hard work, you should have something like this (the image depends of course of what is currently on the screen you eavesdrop). The key elements here are

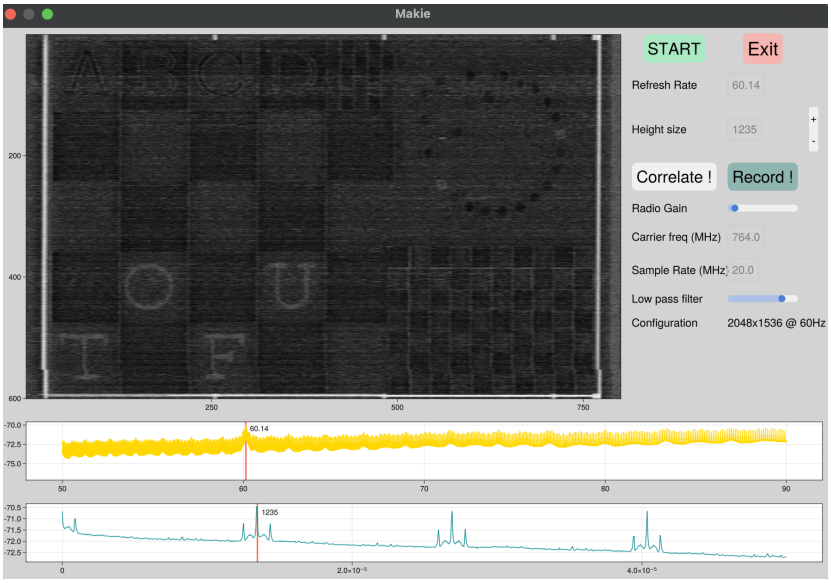


Figure 4.4: A practical eavesdrop !

- the link between the first autocorrelation and the rate (the one depicted in "correlation for rate" panel
- the link between the zoom of the second autocorrelation (the one depicted in "correlation display for image height" panel) and the height
- the reconstruction of the image (note that you should low pass filter the image to limit the noise)

Bibliography

[Eck] W. Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? 4(4):269–286.

[ESE] Furkan Elibol, Ugur Sarac, and Isin Erer. Realistic eavesdropping attacks on computer displays with low-cost and mobile receiver system. In *2012 Proceedings of the 20th European Signal Processing Conference (EUSIPCO)*, pages 1767–1771.

[Kuh] Markus G Kuhn. Compromising emanations: Eavesdropping risks of computer displays.

[LGG+] Corentin Lavaud, Robin Gerzaguet, Matthieu Gautier, Olivier Berder, Erwan Nogues, and Stephane Molton. Whispering Devices: A Survey on How Side-channels Lead to Compromised Information. 5(2):143–168.

[Mar] M. Marinov. Remote video eavesdropping using a software-defined radio platform.

[PDZ+] Abhishek Patnaik, Soumya De, Yao-Jiang Zhang, James L. Drewniak, Chen Wang, Charles Jackson, and David Pommerenke. EMI Analysis of DVI Link Connectors. 60(1):149–156.