Side Channel Attacks

Ryan Rearden

December 19, 2023

Abstract

TEMPEST attacks are a type of side-channel attack that exploit electromagnetic radiation to eavesdrop on electronic devices. Such attacks can compromise confidential data by analyzing unintended physical behavior. Vigilance and appropriate measures are crucial to mitigate these threats to data security. It is essential to be aware of the potential risks and implement safeguards against these attacks to protect sensitive information. The purpose of this project is to demonstrate the change in electromagnetic signals with different monitor display variations.

1 Introduction

TEMPEST stands for "Telecommunications Electronics Material Protected from Emanating Spurious Transmissions." Every electronic device emits electromagnetic radiation. Since the 40s, people have attempted to capture the signal of the device for the sake of curiosity, espionage, and hacking. Although modern devices leak much less electromagnetic noise than those from the 80s-2000s, the threat for TEMPEST attacks still exists. Even today, documentation from FOIA requests to the NSA on this are heavily censored because of the potential consequences of these attacks. All someone needs is an antenna, radio, and a way to decode the transmissions. TEMPEST is an essential consideration for anyone working with sensitive electronic equipment, including government agencies, militaries, and private companies. Understanding the principles and techniques of TEMPEST through numerical and simulated techniques is crucial for maintaining the security and confidentiality of electronic communications and data, and can help to prevent unauthorized access and interception of sensitive information.

2 Procedure

Before discussing the steps to code the simulation, the math must first be explained [2]. The change in current in a wire generates a circular changing magnetic field around itself. This can be modeled by the equation:

$$x(t) = I(t)sin(2\pi f_c t) \tag{1}$$

Where $I(t)$ is some kind of data, f_c is the carrier frequency and t is time. If wire A is generating current with the data and that wire is parallel to wire B, the changing magnetic field from wire A will generate a current in wire B. This is shown by:

$$\hat{x}(t) = \hat{I}(f_c) * I * sin(2\pi f_c t + \varphi(f_c)) \tag{2}$$

Where $\hat{I}(t)$ is the attenuation of the peak amplitude and $\varphi(f_c)$ is the phase difference of the signal in wire B. By knowing the carrier frequency and making some assumptions about the signal, $I(t)$ could be estimated. This is the foundation of side channel attacks. Someone can passively receive data due to the nature of current.

In a computer, current is generated in many places, but the most useful place is within the Low-Voltage Differential Signaling cable (LVDS) which is connected by the Flat Panel Display link (FPD link). The FPD link is a protocol for transmitting digital video strams from graphics processors to digital displays. The LVDS cable is the messenger that tells the screen what color pixel to display in which location. This generates current. The duration of transmission for each pixel is modeled by

$$t_b = \frac{1}{x_t * y_t * f_v * n_b} \tag{3}$$

Where x_t is the number of pixels per scan line, y_c is the number of scan lines, f_v is the frames per second of a computer, and n_b is the number of bits per pixel.

The video signal results as

$$\tilde{v}(t) = \sum_{k=-\infty}^{\infty} c_k b(t - k * t_b) \tag{4}$$

Where $b(t)$ is the shape of the digital bit and c_k is the continuous bit stream. A Fourier analysis can be done on this signal, to obtain

$$\mathcal{F}\left\{b(t) * \left(c(t) \cdot \text{Ш}_{t_b}(t)\right)\right\} = \frac{1}{t_\text{b}} B(f) \cdot \left[C(f) * \text{Ш}_{t_b^{-1}}(f)\right]. \tag{5}$$

Using python and Matplotlib, the Fourier transform can be displayed in a simulated environment. This required simulating a monitor with a black screen. The programmer can input how many scan lines on the x and y axis is needed to simulate, then when one of the grids is clicked, the grid goes either from black to white or from white to black. While performing a side channel attack it is difficult to decode each RGB value since they are so close together within the computer. Instead, it is much easier to derive the intensity of the pixel and format that as a greyscale color. For a 256 color scheme, R G and B each have 8 bits attached to them (totaling at 24 bits total). For example purple is 10100000, 00100000, 11110000. As the amount of 1s increases, the closeness it is to white also increases. White and black were used for simplicity and proof-of-concept. When the grid state changes, a list of lists is made giving the pixel intensity (either 1 or 0). From Equation 5, the video signal of the grid screen can be simulated. Below is a short snippet of code showing this transform.

```
# Construct the video signal
video_signal = np.sum(v_i[i] * np.roll(self.pixel_shape(self.t - i * self.t_b), int(i *
    self.t_b * self.f_v)) for i in range(len(v_i)))

# Compute the Fourier transform of the video signal
video_spectrum = np.fft.fft(video_signal)
freq = np.fft.fftfreq(len(self.t), d=self.t[1] - self.t[0])

# Update the video signal and spectrum plots
self.video_signal_line.set_ydata(video_signal)
self.video_spectrum_line.set_xdata(freq)
self.video_spectrum_line.set_ydata(np.abs(video_spectrum))
```

3 Results

The code runs successfully and as expected.

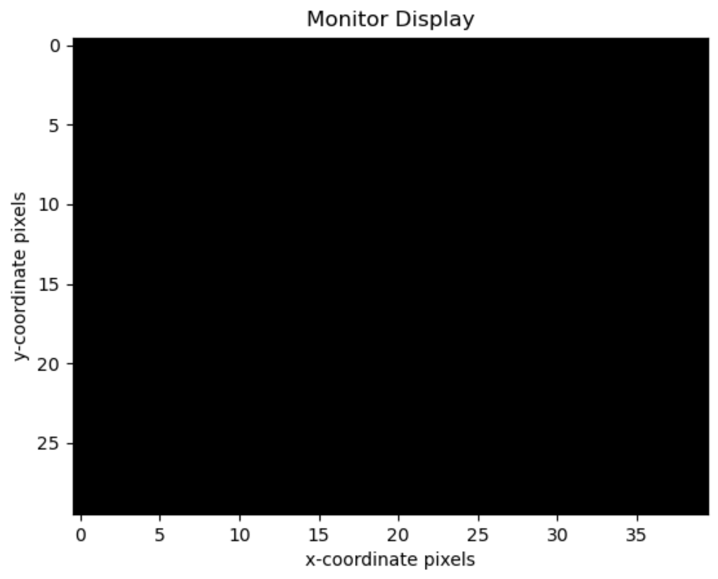


Figure 1: Black screen monitor

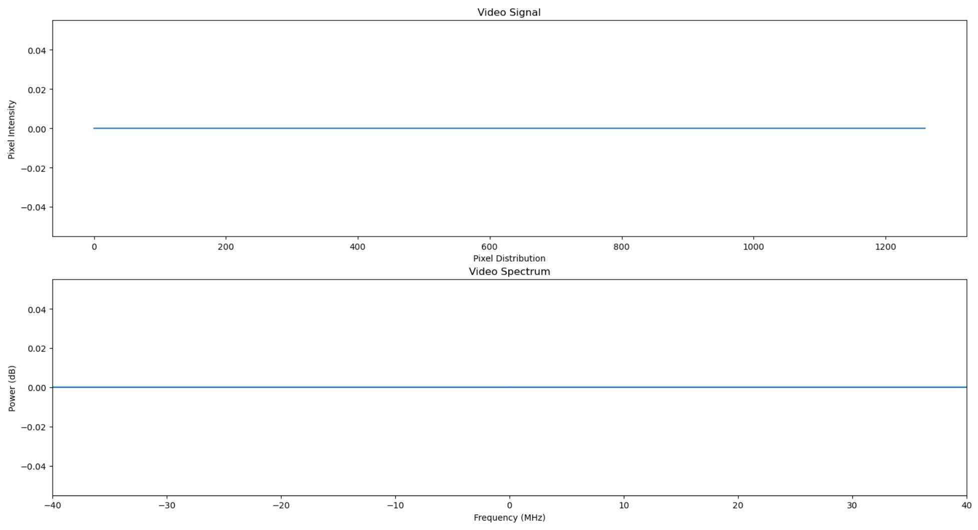


Figure 2: No emitted signal due to empty screen

1 and 2 show the lack of signal when there are no pixels turned on in the monitor. Realistically, there would be some signal movement due to other accidental (or purposeful) signals on the spectrum.

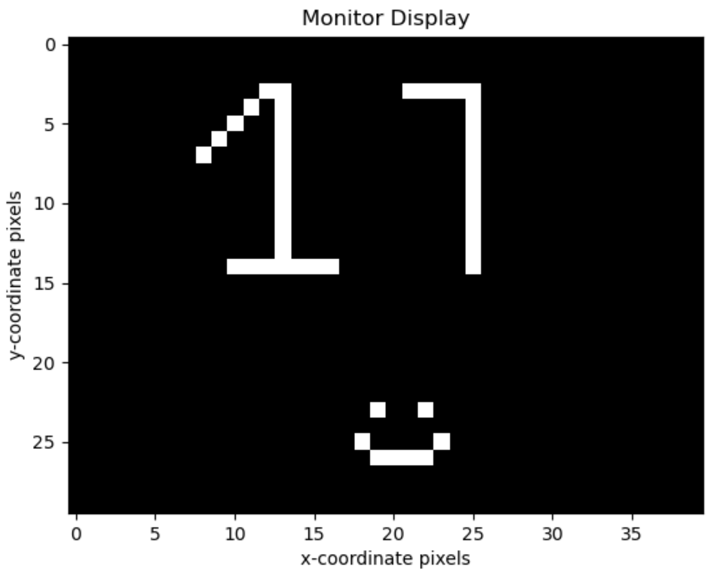


Figure 3: Visuals on the monitor display

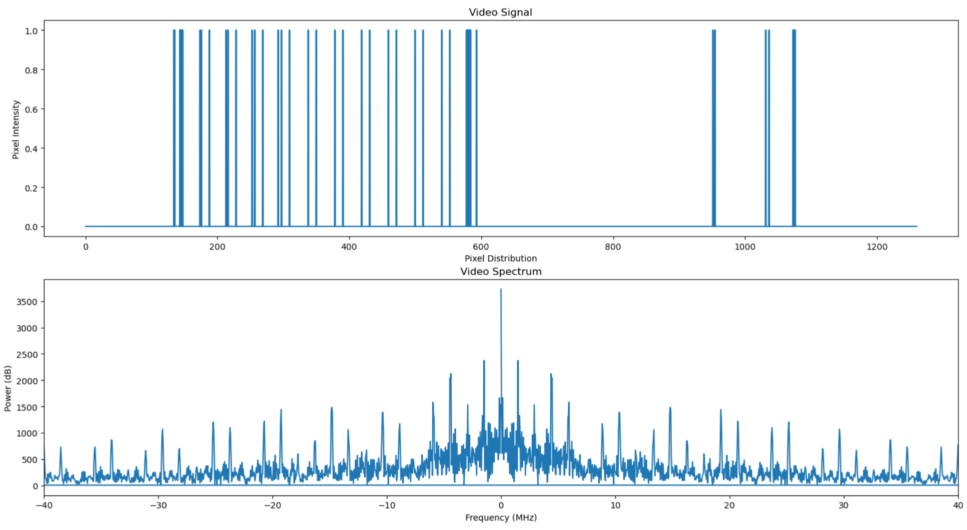
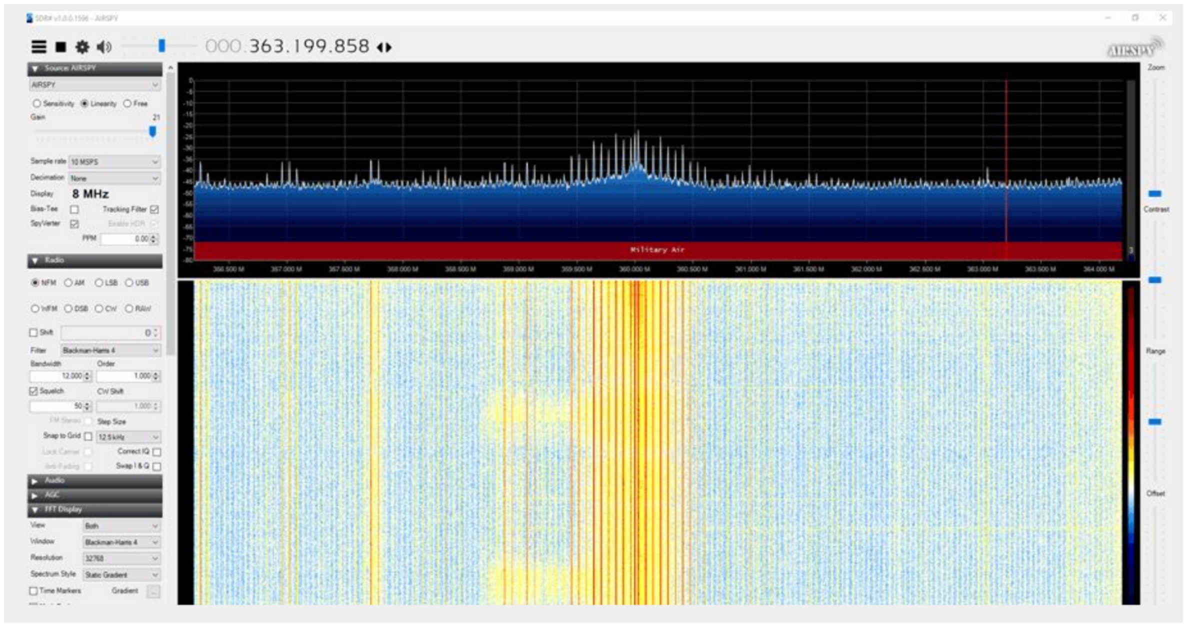


Figure 4: Electromagnetic spectra given from the pixels

3 and 4 show the visuals for a screen that has pixels actively being displayed. The video signal shows what pixels are being shown. Zero represents the pixel in the top left corner and the pixel at coordinate $x * y$ is at the bottom right corner. An attacker would be able to backtrack from the frequency spectrum and, with a little analysis, be able to reverse all the way to the original screen in grey scale.



Source: AIRSPY
AIRSPY
Sensitivity [✓ Linearity] Free
Gain 21
Sample rate 10 MSPS
Decimation None
Display 8 MHz
Bias-Tee ☐ Tracking Filter ☑
SpyVerter ☑ Enable HDR
PPM 0.00
Radio
[✓ NFM] AM LSB USB
WFM DSB CW RAW
☐ Shift 0
Filter Blackman-Harris 4
Bandwidth 12,000 Order 1,000
☑ Squelch 50 CW Shift 1,000
FM Stereo Step Size
Snap to Grid ☐ 12.5 kHz
Lock Carrier Correct IQ ☐
Anti-Fading Swap I & Q ☐
Audio
AGC
FFT Display
View Both
Window Blackman-Harris 4
Resolution 32768
Spectrum Style Static Gradient
☐ Time Markers Gradient

Figure 5: Real-world signal from a monitor in HSC 22

Figure 5 was created by using a software defined radio and an application on my laptop to display the signal of a monitor in the Computer Science Lab. The model of the spectrum looks very close to the real-world signal. The reason why the real-world signal looks less distinct is because of the lack of perfection in reception and the more dark grey in the pixel intensity of the monitor.

4 Discussion and Conclusion

The subject of side channel attacks is a fascinating subject that will always be an issue as long as a wire transmits a current. Papers are continuing to come out investigating properties of electromagnetic radiation leakage and how to harness it as effectively as possible. Due to the explanations given in the original paper, coding the simulation was easier than it could have been. The author made a very complicated topic easy to understand. The simulation gives insight into the theoretical signal a screen would emit based on the pixels displayed. In an actual scenario, the signal would be a little harder to differentiate since, in the relative frequency that the signal would be highest, there are a lot of other electronics emitting at various wavelengths around it. This was a difficulty when I was trying to find the signal on my own in the computer science lab. There were so many monitors and electronics that it was hard to find the correct frequency.

There was a study done in 2020 trying to catch the signal of a phone screen rather than a computer screen. Since the signal was so small, the researchers used their phones to determine what each number at a certain size looked like on the radio spectrum and then used machine learning to detect those numbers on the phone screen they were ”attacking” [1]. With this simulation, if a user knew what device the victim had but did not have strong reception, they could use this simulation in order to quickly experiment with what image might be on the screen.

Overall, this simulation seems very effective in displaying the actual signal. It would be interesting to investigate this further and attempt to push the boundaries of the subject. Much of the information on this is still classified and even blacked out on unclassified documents. Before the end of my time at the University of Dallas, I would like to do a research project on this matter to understand it more clearly and dive deeper into what is possible.

References

[1] Liu, Z., Samwel, N., Weissbart, L., Zhao, Z., Lauret, D., Batina, L., and Larson, M. Screen gleaning: A screen reading tempest attack on mobile devices exploiting an electromagnetic side channel, 2020.

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

5 Appendix

```
import numpy as np
import matplotlib.pyplot as plt

'''
When the program launches, click on the black rectangle to see the signal!
'''


class ToggleGrid:
    def __init__(self, x, y, grid_title="Monitor Display", x_label="x-coordinate pixels",
        y_label="y-coordinate pixels"):
        self.rows, self.cols = y, x
        self.grid_data = np.zeros((self.rows, self.cols), dtype=int) # Initialize with zeros
            (black)

        self.fig, self.ax_grid = plt.subplots()
        self.ax_grid.imshow(self.grid_data, cmap='gray', vmin=0, vmax=1, origin='upper')

        self.fig.canvas.mpl_connect('button_press_event', self.on_click)
        self.ax_grid.set_title(grid_title)
        self.ax_grid.set_xlabel(x_label)
        self.ax_grid.set_ylabel(y_label)

        # Define parameters
        self.t_b = 1/60 # Duration of pixel transmission
        self.f_v = 60 # Frame rate
        self.t = np.linspace(0, (x*y)/1/60 + 1, 100000) # Generate time vector

        # Define pixel shape function (for simplicity, using a rectangular shape)
        self.pixel_shape = lambda t: np.where((t >= 0) & (t <= self.t_b), 1, 0)

        # Plot the video signal and spectrum
        self.fig_video, self.ax_video = plt.subplots(2, 1, figsize=(12, 6))
        self.video_signal_line, = self.ax_video[0].plot(self.t*60, np.zeros_like(self.t))
        self.video_spectrum_line, = self.ax_video[1].plot(self.t, np.zeros_like(self.t))

        self.ax_video[0].set_title('Video Signal')
        self.ax_video[0].set_xlabel('Pixel Distribution')
        self.ax_video[0].set_ylabel('Pixel Intensity')

        self.ax_video[1].set_title('Video Spectrum')
        self.ax_video[1].set_xlabel('Frequency (MHz)')
        self.ax_video[1].set_ylabel('Power (dB)')

        self.update_video()

    def on_click(self, event):
        if event.inaxes == self.ax_grid:
            row, col = int(event.ydata + 0.5), int(event.xdata + 0.5)
            self.grid_data[row, col] = 1 - self.grid_data[row, col]
            self.ax_grid.imshow(self.grid_data, cmap='gray', vmin=0, vmax=1, origin='upper')
            plt.draw()

            # Print the grid state after toggling
            print("Toggled Grid State:")
            print(self.get_grid_state())
            self.update_video()

    def get_grid_state(self):
        return self.grid_data.tolist()

    def update_video(self):
        # Generate pixel intensity values
        arr = np.array(self.get_grid_state())
        v_i = np.array(arr.flatten())

        # Construct the video signal
        video_signal = np.sum(v_i[i] * np.roll(self.pixel_shape(self.t - i * self.t_b), int(i
             * self.t_b * self.f_v)) for i in range(len(v_i)))

        # Compute the Fourier transform of the video signal
        video_spectrum = np.fft.fft(video_signal)
        freq = np.fft.fftfreq(len(self.t), d=self.t[1] - self.t[0])

        # Update the video signal and spectrum plots
        self.video_signal_line.set_ydata(video_signal)
        self.video_spectrum_line.set_xdata(freq)
        self.video_spectrum_line.set_ydata(np.abs(video_spectrum))

        self.ax_video[0].relim()
        self.ax_video[0].autoscale_view()

        f_max = x # Set your desired maximum frequency here
        f_min = -x
        self.ax_video[1].set_xlim([f_min, f_max])
        self.ax_video[1].relim()
        self.ax_video[1].autoscale_view()

        self.fig_video.canvas.draw_idle()


if __name__ == "__main__":
    x, y = 40, 30  # Set your desired width (x) and height (y)
    grid = ToggleGrid(x, y)

    plt.show()
```