



UNIVERSITY OF
CAMBRIDGE

Department of Computer
Science and Technology

RGB DisplayPort Eavesdropping

Stefany Nizamova

Pembroke College

June 2026

Submitted in partial fulfillment of the requirements for the
Computer Science Tripos, Part III

Total page count: 55

Main chapters (excluding front-matter, references and appendix): 44 pages (pp 9–52)

Main chapters word count: 11960

Methodology used to generate that word count:

```
texcount -inc -sum -1 main.tex
```

Declaration

I, Stefany Nizamova of Pembroke College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Signed: Stefany Nizamova

Date: June 1, 2026

Abstract

Electromagnetic emissions from digital circuits can unintentionally leak information. Video interfaces, including VGA, DVI, HDMI, and LVDS, are known to be vulnerable to Tempest-style eavesdropping attacks that recover displayed images by amplitude demodulation of radiated signals. DisplayPort, a newer interface, was considered more resistant owing to its scrambled, non-linear 8b/10b encoding, but recent work has demonstrated the first practical attack.

This project extends that work by resolving the open challenge of *leap bytes*: periodic single-byte insertions that, until now, prevented byte-level synchronisation with the symbol stream. By recovering this synchronisation, the attack achieves full RGB colour reconstruction of the eavesdropped display.

Acknowledgements

I would like to thank my supervisor Markus Kuhn, for his guidance and support throughout this project.

I am also grateful to Dimitrije Erdeljan, whose work this dissertation builds upon, and who generously shared his insights and advice throughout.

Finally, thanks to Liam, for his support and encouragement.

Contents

Notation	7
1 Introduction	9
1.1 Compromising emanations	9
1.2 Eavesdropping on digital displays	9
1.3 Scope of this project	10
1.4 Outline	11
2 Background	12
2.1 Signal representation	12
2.1.1 Software-defined radio (SDR)	12
2.1.2 Operators on IQ samples	13
2.2 DisplayPort specification	14
2.2.1 Data framing and leap bytes	14
2.2.2 Scrambling	17
2.2.3 8b/10b encoding and running disparity	18
3 DisplayPort eavesdropping-attack pipeline	19
3.1 Scrambler synchronisation	20
3.2 Image reconstruction	21
3.3 Code phase tracking	22
3.4 Carrier phase tracking	23
3.5 Coherent averaging	24
4 Design and implementation	26
4.1 Detecting leap bytes	26
4.1.1 Target image	26
4.1.2 Horizontal and vertical alignment	26
4.1.3 Fine alignment using grid search	28
4.1.4 Bresenham sequence	31
4.2 Colour reconstruction	33
4.2.1 Leap-aware line tracker	33

4.2.2	Coherent RGB colour decoding	35
4.2.3	Removing fill regions	35
4.2.4	Running disparity and colour decoding	37
4.2.5	Full colour enumeration	38
4.3	Refined leap byte detection	39
4.3.1	Leap byte detection without known image content	40
4.4	Required precision of Bresenham parameters	40
5	Evaluation	44
5.1	Experimental setup	44
5.1.1	Capture hardware	44
5.1.2	Source hardware	45
5.2	10-bit colour depth	45
5.3	RGB attack pipeline	46
5.4	Attack results	47
5.4.1	Intel HD Graphics 630 and 10-bit colour depth	47
5.4.2	Leap characterisation across distance	47
5.5	Remaining limitations of the RGB colour attack	48
6	Summary and conclusions	50
6.1	Future work	50
6.1.1	Multi-lane video modes	50
6.1.2	Spread-spectrum clocking	51
6.1.3	Efficiency and convergence improvements	51
A	Device screening	55

Notation

Mathematical

j	Imaginary unit satisfying $j^2 = -1$
e	Euler's constant $e = 2.71828\dots$
$\mathbb{Z}_n$	Set of natural numbers smaller than n : $\mathbb{Z}_n = \{m \in \mathbb{Z} \mid 0 \leq m < n\}$
$a \oplus b$	Bitwise exclusive-or of integers a and b
$\Re\{x\}$	Real part of $x \in \mathbb{C}$
$\bar{x}$	Complex conjugate of $x \in \mathbb{C}$
$ x $	Magnitude of complex number x : $ \mu e^{j\phi} = \mu$ for $\mu, \phi \in \mathbb{R}$
$\angle x$	Phase of complex number x : $\angle(\mu e^{j\phi}) = \phi$ for $\mu, \phi \in \mathbb{R}$, $0 \leq \phi < 2\pi$
$\lfloor x \rfloor$	Floor of $x \in \mathbb{R}$
$\lceil x \rceil$	Ceiling of $x \in \mathbb{R}$
$\{x\}$	Fractional part of $x \in \mathbb{R}$: $\{x\} = x - \lfloor x \rfloor$

Signals

$s[n]$	Received baseband signal at sample index n
ξ	Bitstream
$z(\sigma_0, \mathbf{b}, \rho_0)$	Template signal; see (3.2)
$z_{\mathbf{b}}^\ell$	Blanking template at line ℓ : $z(\sigma_\ell, \mathbf{0}, \rho_0)$

Indices and positions

n	Sample index into the received signal s
ℓ	Line index
n_ℓ	Sample index into the received signal s starting at line ℓ
$\sigma[n]$	Scrambler state (in bytes) at sample n ; see (3.4)
σ_ℓ	Scrambler state at the start of line ℓ
σ_0	Initial scrambler state

Frequencies and rates

- f_c Center frequency
- f_b DisplayPort link bitrate
- $\bar{f}_b$ Average tracked bitrate: $\bar{f}_b = \frac{1}{|s|} \sum_n f_b[n]$
- f_s Receiver sampling rate
- $\bar{f}_B$ Average tracked byte rate: $\bar{f}_B = \bar{f}_b/10$

Template parameters

- $\mathbf{b}$ Byte sequence used to synthesise a template
- $\rho_0 \in \{+1, -1\}$ Initial 8b/10b running disparity

Template synthesis operators

- $\text{scr}(\lceil \sigma_0 \rceil, \mathbf{b})$ Scrambles $\mathbf{b}$ starting from byte $\lceil \sigma_0 \rceil$ of the scrambler sequence
- $\text{enc}(\cdot, \rho_0)$ 8b/10b encodes a byte sequence with initial running disparity ρ_0
- DS_{f_s} Downsamples to f_s , aligning sample $n=0$ with position σ_0 in the bitstream

Chapter 1

Introduction

1.1 Compromising emanations

The logic transitions inside digital circuits can produce unintentional signals that carry enough information to reconstruct processed data at a distance. These *compromising emanations* — termed *TEMPEST* in government standards — have been a subject of military and intelligence research since the mid-twentieth century [12, 1].

Electromagnetic emissions were encountered as a problem within the intelligence community when cipher machines were found to leak plaintext as a signal [3]. The suppression of such emissions motivated a dedicated NSA research programme. The phenomenon remained a highly classified subject for several decades, only entering the open literature in 1985 when van Eck demonstrated that the image on a cathode-ray-tube monitor could be reconstructed at a distance using a modified television receiver [14].

1.2 Eavesdropping on digital displays

The van Eck demonstration on cathode-ray tube (CRT) displays worked by detecting the pulses emitted when the analog video signal switched the CRT electron beam on or off. Representing this information as an image with the same line and frame rate as the source results in a reconstruction where horizontal transitions between dark and bright areas are visible.

Van Eck’s publication prompted a broader research interest in compromising emanations: civilian devices have standards to prevent interference between equipment, but are not designed to guard against eavesdropping. Attacks were demonstrated on a range of targets [13, 5], including Ethernet, cash machines [9] and voting machines [8].

Video interfaces became a particular focus of research: text on a screen occupies many pixels and is retransmitted at the frame rate, giving eavesdroppers both spatial and

temporal redundancy to exploit. Early digital panel interfaces used *low-voltage differential signalling* (LVDS) with a simple non-return-to-zero (NRZ) line encoding, in which pixel brightness values are transmitted directly as high or low voltages. Because different pixel values produce distinct amplitude spectra, an AM demodulator is sufficient to distinguish them.

Three standardised external digital video interfaces have become commonly used: *Digital Visual Interface* (DVI), *High-Definition Multimedia Interface* (HDMI), and *DisplayPort*. DVI and HDMI both use *Transition Minimized Differential Signalling* (TMDS) line encoding, transmitting each byte as one of several possible 10-bit symbols. Despite this more complex non-linear encoding, DVI and HDMI are still vulnerable to eavesdropping using AM demodulation: although a given byte may map to several possible symbols, each colour corresponds to only a small number of distinct symbol patterns, allowing them to be distinguished.

DisplayPort is a newer external video interface that is commonly used for higher resolution desktop computer displays. Eavesdropping on DisplayPort is massively complicated by an additional *scrambling* stage that applies a pseudo-random XOR mask to the data before encoding. This is designed to remove image-dependent spectral characteristics in electromagnetic emissions so that the device does not interfere with other equipment: scrambling the data flattens the spectrum. AM, FM or QAM demodulating this scrambled stream yields only noise. Moreover, it is common practice to use periodic patterns in display emissions to perform periodic averaging and recover timing information. In DisplayPort, the period of the scrambler is unrelated to the number of bytes in a line or frame, so eavesdroppers cannot benefit from such periodicity in the signal. Finally, DisplayPort has a high bitrate of at least 1.62 Gbit/s, requiring specialised equipment to capture the signal with a wide enough bandwidth.

As a result of these challenges, the risk of DisplayPort eavesdropping had been considered low. A 2016 TEMPEST guidance document for designers of electronic voting machines recommends the use of display panels with embedded DisplayPort (eDP) interfaces [10].

In 2023 Erdeljan [6] demonstrated the first practical attack. The attack is based on synthesising a local, synchronised copy of the scrambler; then template signal can be constructed for each target colour. The colour of each pixel can be inferred by examining the correlations between the received signal and the templates. Using this approach, Erdeljan was able to reconstruct displayed images at a distance of up to 9 meters under realistic conditions.

1.3 Scope of this project

A remaining challenge in Erdeljan’s work is the presence of so called *leap bytes* – extra bytes that are inserted into the symbol stream to satisfy certain timing requirements

of the interface. Their exact pattern is implementation-defined and not specified by the DisplayPort standard. This introduces uncertainty into the scrambler-reset timing model, preventing byte-level synchronisation with the symbol stream. Without byte-level synchronisation, it is impossible to determine which RGB channel a given byte corresponds to, making full colour recovery unachievable. The work described in this dissertation extends Erdeljan’s work by resolving this open challenge.

1.4 Outline

The remainder of this document is structured as follows. In chapter 2 I cover necessary background material on electromagnetic signal representation and relevant details of the DisplayPort standard. I then explain the existing algorithms for DisplayPort eavesdropping and the open challenge of leap-bytes in more detail (chapter 3). In chapter 4 I describe my initial methodology for detecting leap bytes, what my results show, and how knowledge of the leap byte pattern can be used to achieve byte-level synchronisation and full RGB sensitive recovery. I then discuss algorithms to integrate leap byte detection into the existing attack pipeline, to allow for RGB colour recovery in practice. In chapter 5 I describe the experimental setup I used to evaluate my algorithms, and present results demonstrating the approach. Finally, in chapter 6 I summarise my contributions and discuss possible future work.

Chapter 2

Background

In this chapter, I cover the foundations of electromagnetic signal representation and processing. I then cover the details of the DisplayPort specification that are relevant to this project.

2.1 Signal representation

2.1.1 Software-defined radio (SDR)

A *software-defined radio (SDR)* receiver is a general-purpose device for sampling a part of the radio spectrum. The samples are streamed to a computer, where demodulation or decoding steps can be implemented in software.

The SDR is configured to sample at a specified *sampling rate* f_s and *center frequency* f_c . The sampling rate sets the *bandwidth* of the spectrum that can be captured: to cover an interval of width B about f_c , the Nyquist limit requires $f_s \geq B$. In practice f_s is chosen somewhat above this to leave room for the anti-aliasing filter's transition band; a typical margin gives $f_s \approx 1.25B$. The process of sampling the signal is called *IQ downconversion*: an incoming antenna waveform $s_0(t)$ has its frequency spectrum shifted down by f_c , so that the target interval $[f_c - B/2, f_c + B/2]$ is mapped to the baseband interval $[-B/2, B/2]$. This is done by multiplying the signal with a phasor rotating with frequency $-f_c$:

$$s_d(t) = s_0(t) \cdot e^{-j2\pi f_c t}. \quad (2.1)$$

This is implemented in hardware as two separate multiplications by $\cos(2\pi f_c t)$ and $-\sin(2\pi f_c t)$, producing the *in-phase* (I) and *quadrature* (Q) components respectively:

$$I(t) = s_0(t) \cdot \cos(2\pi f_c t) \quad (2.2)$$

$$Q(t) = s_0(t) \cdot (-\sin(2\pi f_c t)) \quad (2.3)$$

These are combined into the complex-valued *baseband signal* $s_d(t) = I(t) + jQ(t)$.

Since $s_0(t)$ is a real-valued signal, its spectrum is conjugate-symmetric: every component at frequency f has a mirror image at $-f$. After multiplication by $e^{-j2\pi f_c t}$, the positive-frequency components are shifted to baseband as desired, but the negative-frequency mirror images are shifted to $-(f + f_c)$. These images are removed by multiplying the spectrum by a window function that zeros out all frequencies outside $[-B/2, B/2]$; by the convolution theorem, this is equivalent to convolution with the filter impulse response $h(t)$ in the time domain:

$$s_f(t) = s_d(t) * h(t) \quad (2.4)$$

$$= \int s_d(t - \tau) h(\tau) d\tau \quad (2.5)$$

The filtered signal is then sampled into a discrete time sequence

$$s_r[n] = s_f\left(\frac{n}{f_s}\right), \quad n \in \mathbb{Z} \quad (2.6)$$

Each complex sample produced simultaneously captures the amplitude and phase of the signal at the point in time when it was sampled.

In the experiments described in this dissertation, an *Ettus USRP X300* is used as the SDR receiver. The USRP contains an analog-to-digital converter (ADC) that supports a maximum sampling rate of 200 M samples per second. Analog signal processing (downconversion and filtering) is implemented via replaceable daughterboards, I use the *UBX-160* which supports a tunable center frequency range of 10 MHz to 6 GHz and a bandwidth of 160 MHz.

Experiments used a sampling rate of $f_s = 50$ MHz, with each sample represented as a pair of 32-bit floating point numbers, the required data rate is 400 MB/s, or 3.2 Gbit/s. This was supported using a 10Gb Ethernet over fibre-optic cable connected to the receiving desktop's network interface card (NIC).

2.1.2 Operators on IQ samples

The reconstruction algorithms in this dissertation often reduce to one of three related operators on complex baseband samples – the pointwise product:

$$s[n] \overline{z[n]} \quad (2.7)$$

the inner product:

$$\langle s, z \rangle = \sum_n s[n] \overline{z[n]} \quad (2.8)$$

and the cross correlation:

$$R_{z,s}[d] = \sum_n s[n+d] \overline{z[n]} \quad (2.9)$$

where $s[n]$ is the received signal and $z[n]$ is a *template signal* (see section 3.1). All three are views of the matched filter, which is the optimal linear filter for maximising the signal-to-noise ratio in the presence of additive stochastic noise [16].

Intuitively, the pointwise product of two samples $s[n]\overline{z[n]}$ has a magnitude of $|s[n]||z[n]|$ and a phase of $\arg(s[n]) - \arg(z[n])$. Where the template matches the signal and the carrier phase is aligned, $\arg(s[n]) - \arg(z[n])$ will be near zero, so the product will have a large positive real part. Summing over n to compute the inner product $\langle s, z \rangle$ constructively combines contributions along the real axis, so the real part of the inner product can be used as a measure of how well the template matches the signal. The cross correlation $R_{z,s}[d]$ is a generalisation of the inner product that allows for an offset d between the signal and template, so it can be used to search for the best alignment between the two.

2.2 DisplayPort specification

DisplayPort [15] is a digital video interface standard developed by the *Video Electronics Standards Association (VESA)* and used to connect a video source like a computer to a display device such as a monitor. It is designed for high-throughput unidirectional video data transfer. Its *main link* consists of four twisted wire pairs, known as *lanes*, using differential transmission: signal is transmitted over two separate paths using complementary currents. The purpose of this is to reduce electromagnetic interference emitted by high-speed interfaces – as the distance between the two paths approaches zero, the electromagnetic fields generated cancel each other out. In practise, the distance cannot be reduced to zero, and the signal is not perfectly balanced, so some electromagnetic emissions are still produced.

The original DisplayPort standard allowed only two possible bit rates per lane: 1.62 Gbit/s and 2.7 Gbit/s, later versions of the standard also support 5.4, 8.1, 10.0, 13.5 and 20.0 Gbit/s. The lowest bitrate that provides enough capacity for the current video mode is chosen, and the remaining capacity between packets of image data and blanking is padded with zero-valued *fill* bytes.

The padded datastream is then scrambled by XORing it with the output of a 16-bit linear feedback shift register and encoded using an 8b/10b encoder.

2.2.1 Data framing and leap bytes

The video mode specifies the display image dimensions (w_d, h_d) in pixels; the total frame dimensions (w_t, h_t) ; and the pixel clock frequency f_p . The total dimensions exceed the im-

age dimensions by the *blanking intervals* — periods inherited from analogue CRT displays during which no pixel data is transmitted. From this, the frame rate f_v can be calculated

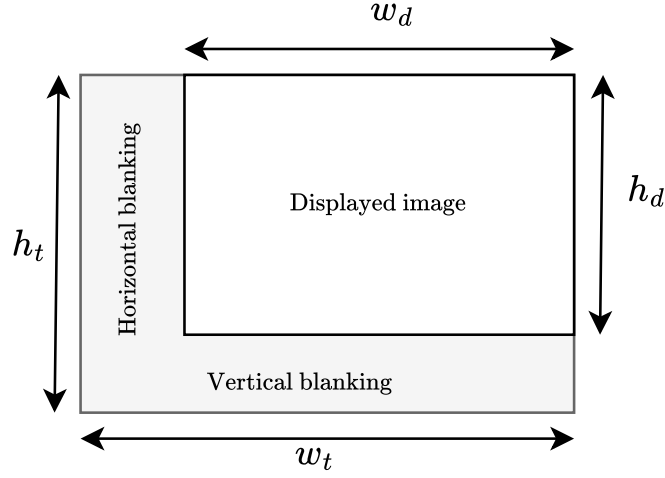


Figure 2.1: Diagram of a transferred frame, showing the displayed image area and the blanking areas.

as $f_v = f_p / (w_t \cdot h_t)$. Since each pixel is represented by three bytes (R, G and B) and each byte is encoded as a 10-bit symbol, the *data*¹ bitrate of the signal is $3 \cdot 10 \cdot h_t \cdot w_t \cdot f_v$ bit/s, or simply $30f_p$. The transmitter selects the minimum supported bitrate and number of lanes $n_L \in \{1, 2, 4\}$ that can support this data rate. Intel drivers prioritise a lower bitrate over fewer lanes [2].

For example, for a video mode 1280×800 at a pixel clock of 83.5 MHz, the data bitrate is $30 \times 83.5 \text{ MHz} = 2.505 \text{ Gbit/s}$. Since this is below $1.62 \text{ Gbit/s} \times 4 = 6.48 \text{ Gbit/s}$, the 1.62 Gbit/s link rate is used; and since $1.62 \text{ Gbit/s} \times 2 = 3.24 \text{ Gbit/s}$ exceeds the data bitrate, two lanes suffice ($n_L = 2$). The remaining capacity of $3.24 - 2.505 = 0.735 \text{ Gbit/s}$ will be satisfied using fill bytes.

Where the number of lanes is greater than one, the data pixels are interleaved across lanes, so for $n_L = 4$, pixels 0, 4, 8... are transmitted on lane 0, pixels 1, 5, 9... on lane 1, and so on.

The DisplayPort standard leaves the fill-byte padding algorithm as an implementation detail, specifying only that the correct ratio of data bytes to transferred bytes be satisfied. Erdeljan [6] reverse engineered the specific algorithm used by two Intel graphics controllers — the *Intel HD Graphics 520* and *Intel HD Graphics 4600* — from oscilloscope recordings, and notes that the details may be implementation-specific. In the course of this work, the same algorithm was found to apply to the *Intel HD Graphics 630*.

This works as follows: image and fill bytes are grouped into $l_{TU} = 64 \text{ B}$ blocks known as

¹I refer to the displayed image pixels plus the blanking pixels as the *data* pixels, and the corresponding bitrate as the *data* bitrate.

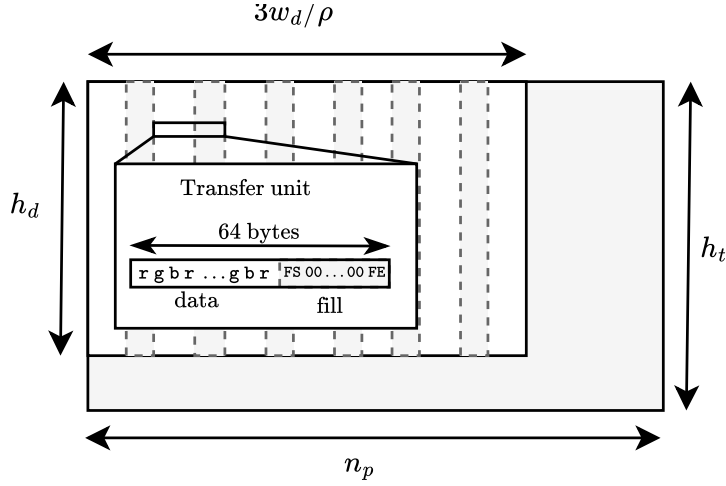


Figure 2.2: Packing of image and fill bytes into transfer units.

transfer units (TUs), such that the required average ratio of data bytes to transmitted bytes is satisfied:

$$\rho = \frac{30\text{bits} \cdot f_p}{n_L \cdot f_b} \quad (2.10)$$

The number of fill bytes in each TU is determined using an approach similar to the Bresenham line drawing algorithm² [4]. The ratio of data bytes to link symbols is represented internally as a fraction M/N , where $N = 2^{23}$ and $M = \lfloor \rho \cdot N \rfloor$, both represented as 24-bit integers.

Since every M data bytes must occupy N link symbols, each data byte emitted is padded with $N-M/M$ fill bytes on average. The GPU maintains an accumulator n_f representing the *fill-byte debt*: the number of fill-bytes owed but not yet emitted. For every image byte emitted, n_f is incremented by $N-M/M$. Once the next byte would cause the total to overflow the transfer unit length: $n_i + n_f + 1 \geq l_{TU}$, the remaining capacity $l_{TU} - n_i$ is packed with fill bytes, and the accumulator is decremented by the same amount, ‘crediting’ the debt. The image byte counter n_i then resets to zero for the next TU.

This results in a pattern where the number of image bytes in each TU is either $\lfloor \rho \cdot l_{TU} \rfloor$ or $\lceil \rho \cdot l_{TU} \rceil$, such that the average ρ is satisfied. The fill byte padding algorithm does not respect pixel boundaries, so the bytes of a pixel may span transfer units. The fractional accumulator that drives the stuffing pattern resets at the start of each line, so the TU structure — and therefore the byte offsets of image data within the line — is identical across all lines of a frame. In particular, the first image byte of a line is always placed as the first byte of the first TU of that line. The last transfer unit at the end of horizontal active display period will be cut short, ending with the *blanking start* control symbol BS

²Used in graphics to approximate a straight line across a discrete pixel grid.

rather than fill bytes. The zero-valued blanking bytes between BS and BE are also stuffed with fill bytes, although without the FS/FE delimiters used during the active period. Since BS is defined as occurring after the last active pixel and BE immediately before the first, line boundaries are determined in the pixel clock domain, while the link transmits symbols at the independent link symbol rate. The total number of link symbols transmitted per line therefore averages

$$n_p = \frac{w_t \cdot 3\text{bytes/pixel}}{\rho} \quad (2.11)$$

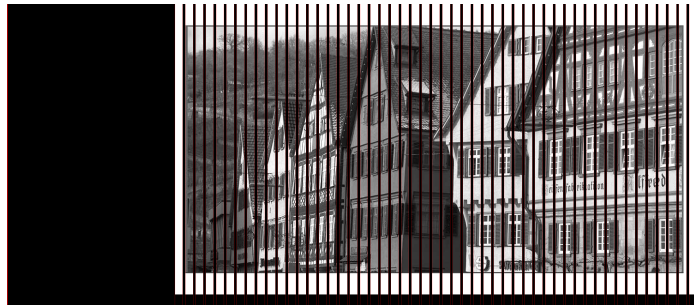
but as n_p is not necessarily an integer, the total line length varies by one byte between lines, in a proportion that achieves the correct long-term average.

For example, for a video mode with $800 \times 600 @ 60.3$ fps, we use a single lane at 1.62 Gbit/s, the total width is $w_t = 1056$ pixels and $f_p = 40$ MHz, so the number of bytes per line *should* be $n_p = 4276.8$. Each line will output either $\lfloor n_p \rfloor = 4276$ or $\lceil n_p \rceil = 4277$ bytes. Where the interface outputs a ‘long’ line, the extra byte is known as a *leap byte*.

The pattern of leap bytes — which lines receive the extra byte — is not specified by the DisplayPort standard, and the mechanism by which the GPU’s display controller determines it is not publicly documented. Erdeljan [6] observed the presence of leap bytes but did not reverse engineer the model that generates them, absorbing the resulting uncertainty as a byte-level error in the scrambler synchronisation model.



(a) Displayed image (from the *Kodak Lossless True Color Image Suite*)



(b) Visualisation after data framing. Control bytes are shown with red pixels. Leap bytes are ignored.³

Figure 2.3: Frame before and after packing

2.2.2 Scrambling

Link data is XORed with *scrambler bytes* in order to flatten the spectrum of electromagnetic emissions. The values of the scrambler are generated as the output of a 16-bit linear feedback shift register, defined by the polynomial:

$$x^{16} + x^5 + x^4 + x^3 + 1 \quad (2.12)$$

³As transfer units do not respect pixel boundaries, Figure 2.3b is visualised by representing each data byte value v as a pixel with RGB values (v, v, v) , that is then rescaled to $1/3$ of the width.

For each byte or control symbol transmitted, the shift register is clocked eight times to produce a scrambler byte $\Xi[n]$. Data bytes are replaced by $b[n] \oplus \Xi[n]$, while control symbols pass through unmodified but still advance the scrambler state. This makes the output bytes appear pseudo-random, and prevents the signal from having image-dependent spectral characteristics that could interfere with other equipment.

Where the number of lanes n_L is greater than one, the same scrambler state is applied across all lanes, but delayed by 0, 2, 4 and 6 byte intervals for lanes 0, 1, 2 and 3 respectively.

As it is defined by a degree-16 primitive polynomial, the scrambler sequence has a period of $2^{16} - 1$ bits. Since $2^{16} - 1$ is coprime with 8, the scrambler byte sequence has a period of $2^{16} - 1$ bytes, meaning the scrambler overflows to its initial state exactly every 65,535 bytes — on the order of tens of lines. Additionally, every 512 lines the scrambler is explicitly reset to its initial state at the start of the horizontal blanking period, with the receiver notified by replacing the blanking start symbol BS with a scrambler reset symbol SR. Since the number of bytes per line can vary due to leap bytes, the number of bytes across 512 lines is not necessarily constant, so the scrambler reset timings are not perfectly periodic.

2.2.3 8b/10b encoding and running disparity

Finally, the output from the scrambling stage is encoded using an 8b/10b encoder, which maps each 8-bit byte to a 10-bit symbol.

Each byte maps to either a single DC *balanced* symbol with five 1-bits and five 0-bits, or one of two complementary symbols, that may be *unbalanced* with six 1s and four 0s, or four 1s and six 0s. The choice between the two complementary symbols is determined by the *running disparity* (RD), which is the difference between the number of 1s n_1 and 0s n_0 transmitted so far, initialised to -1 : $RD = n_1 - n_0 - 1$. The encoder always chooses the symbol that maintains a running disparity of either $+1$ or -1 , so if it has transmitted more 1s than 0s so far, it will choose the symbol with more 0s to balance it out, and vice versa; balanced symbols leave the running disparity unchanged, while unbalanced symbols flip it. This is done to ensure a DC-balanced signal with frequent transitions, which helps with clock recovery and allows links to be AC-coupled.

The encoding process works by splitting the input byte into a 5-bit and 3-bit block, that are mapped to 6-bit and 4-bit blocks respectively, using a lookup table. The resulting 10-bit symbols are transmitted least significant bit first.

Chapter 3

DisplayPort eavesdropping-attack pipeline

In this chapter, I describe the attack pipeline for eavesdropping on DisplayPort, as first created and demonstrated by Erdeljan [6, 7]. The goal of the attack is to reconstruct the image displayed on a monitor from its captured electromagnetic emissions.

The attack begins with recording emissions of the target monitor using a SDR. Since the available sampling rates (maximum of 200 MHz) are much lower than the DisplayPort link bitrate (at least 1.62 Gbit/s), the attack is designed to work with a heavily band-limited

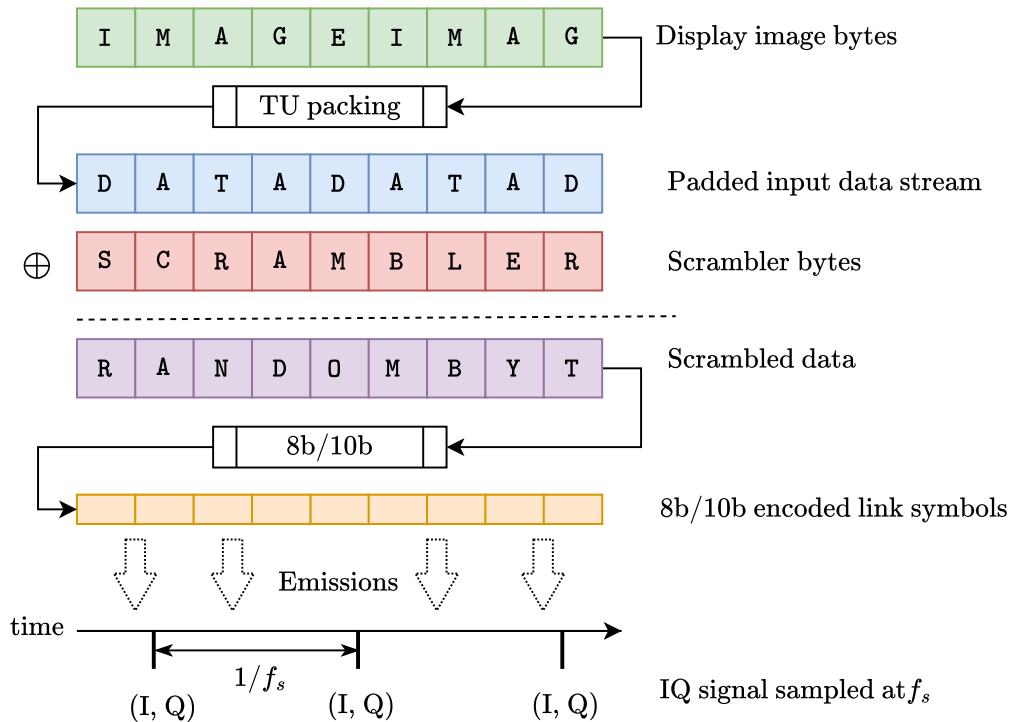


Figure 3.1: Pipeline from displayed image to captured signal.

signal in which individual bits cannot be distinguished.

3.1 Scrambler synchronisation

The scrambler will be in its initial state whenever it resets (every 512 lines) and whenever it overflows its state (every 65,535 bytes). Where the scrambler resets on zero-valued data bytes, we have $0x00 \oplus \Xi[n] = \Xi[n]$, so scrambler reset timings can be found by searching for the presence of the known scrambler sequence in the received signal.

A *template signal* is synthesised by simulating the full transmitter and SDR pipeline for a known byte sequence: the bytes are scrambled, 8b/10b encoded, downconverted to baseband, and downsampled to the receiver sampling rate f_s . The *scrambler state* σ counts how many bytes the scrambler has clocked since initialisation; at state σ it outputs the fixed pseudo-random byte $\Xi[\sigma]$, which is XORed with the data byte. A template can be parameterised by the initial scrambler state σ_0 , the data byte sequence $\mathbf{b}$, and the initial running disparity $\rho_0 \in \{+1, -1\}$:

$$\xi(\sigma_0, \mathbf{b}, \rho_0) = 0^{10(\lceil \sigma_0 \rceil - \sigma_0)} \parallel \text{enc}(\text{scr}(\lceil \sigma_0 \rceil, \mathbf{b}), \rho_0) \quad (3.1)$$

$$z(\sigma_0, \mathbf{b}, \rho_0) = \text{DS}_{f_s}(\xi(\sigma_0, \mathbf{b}, \rho_0)[n] \cdot e^{-2\pi j n f_c / f_b}) \quad (3.2)$$

where $\text{scr}(\lceil \sigma_0 \rceil, \mathbf{b})$ scrambles $\mathbf{b}$ starting from scrambler byte $\lceil \sigma_0 \rceil$, $\text{enc}(\cdot, \rho_0)$ performs 8b/10b encoding with initial running disparity ρ_0 , and DS_{f_s} downsamples to f_s such that sample $n = 0$ aligns with position σ_0 in the transmitter bitstream.

For scrambler reset detection, the data bytes are $\mathbf{b} = \mathbf{0}$ and the initial scrambler state is $\sigma_0 = 0$, so the template reduces to $z(0, \mathbf{0}, \rho_0)$.

The scrambler state σ_0 may be real-valued because in practice it is derived from a signal sample position, which need not fall on a byte boundary.

Since the 512-line scrambler reset happens at the *start* of the horizontal blanking period, it will land on at least one horizontal blanking interval's worth of zero-valued data bytes. Plus, most video modes are guaranteed to include at least one scrambler-reset or overflow during the vertical blanking interval, due to the number of bytes being greater than the scrambler period.

The extracted scrambler-reset timings are used to fit the *scrambler-reset model*. Let X be the set of time offsets when the scrambler is in its initial state. With the simplifying assumption that all line lengths are equal, we can fully describe X by an initial offset x_0 , a *reset period* R representing the time between 512-line scrambler resets, and *LFSR period* P .

$$X = \{x_0 + aR + bP \mid a, b \in \mathbb{Z}, 0 \leq b < R/P\} \quad (3.3)$$

Where a is an integer representing the number of scrambler resets that have occurred since x_0 , and b is an integer representing the number of scrambler overflows that have occurred since the last reset.

The scrambler state $\sigma[n]$ at any point in the signal $s[n]$ is estimated by converting the time offset of the sample from the most recent scrambler reset into a byte offset, and applying the corresponding number of scrambler bytes to the initial state.

$$\sigma[n] = \begin{cases} 0 & \text{if } n \in X \\ \sigma[n-1] + f_b/10f_s & \text{otherwise} \end{cases} \quad (3.4)$$

In reality, leap bytes cause line lengths to vary, so scrambler resets do not occur at perfectly periodic intervals — there is no single true 512-line period in bytes. The parameter R is estimated directly from the signal, so it reflects the observed average interval rather than assuming a fixed one.

The scrambler reset model is fit by finding the parameters $\theta = (x_0, R, P)$ that produce a predicted set of scrambler reset timings X_θ that matches the observed timings X_0 as closely as possible. This is done by finding the (θ, a, b) that minimises the sum of squared error:

$$\mathcal{E}(X_0|\theta, a, b) = \sum_{x \in X_0} (x - (x_0 + a(x)R + b(x)P))^2 \quad (3.5)$$

using an expectation-maximisation approach that alternates between updating $a(x)$ and $b(x)$ (i.e. determining which reset and overflow an observation x corresponds to), and updating θ to best fit the observed timings given the current a and b assignments.

3.2 Image reconstruction

The image reconstruction phase assumes that the target image contains a set of colours C . The received samples $s[n]$ are iterated through, using the scrambler reset model to estimate the scrambler state σ at each sample. For each colour $c \in C$, a pair of template signals are synthesised assuming scrambler state σ and the data bytes $\mathbf{b}_c$ corresponding to colour c :

$$z_c^\pm = z(\sigma, \mathbf{b}_c, \pm 1) \quad (3.6)$$

The two templates differ only in their running disparity assumption; the underlying bit-streams are identical except for unbalanced 8b/10b blocks, where they are inverted. A window of w samples is scored against a colour by taking the magnitude of the inner

product of the received samples with the better-matching template:

$$u_c[n] = \max_{\rho \in \{+1, -1\}} \left\{ \left| \sum_{k=0}^{w-1} s[n+k] \overline{z(\sigma, \mathbf{b}_c, \rho)[k]} \right| \right\} \quad (3.7)$$

Periodic averaging is applied on the frame level by summing $u_c[n]$ across samples corresponding to the same pixel position p in multiple frames, where n_p denotes the sample index of pixel p in the first frame:

$$\text{scores}_c[p] = \sum_{k=0}^{N_f-1} u_c \left[n_p + \frac{kh_t R}{512} \right] \quad (3.8)$$

The colour of each pixel is then decoded by taking the colour with the highest score:

$$\hat{c}[p] = \operatorname{argmax}_{c \in C} \text{scores}_c[p] \quad (3.9)$$

One limitation of this approach is that it can struggle to distinguish *complementary* colour pairs in certain cases. For the 72 out of 256 byte values whose 8b/10b encoding is fully balanced (i.e. both the 5b/6b and 3b/4b output sub-blocks are balanced), the encoded 10-bit symbol is disparity-independent, and a complementary input byte produces an exactly inverted symbol. For such bytes, z_c^+ and z_c^- are identical, and the templates for a colour c and its complement $\neg c$ satisfy $z_c^\pm = -z_{\neg c}^\mp$, giving equal-magnitude but opposite-sign correlations. Since $u_c[n]$ takes the magnitude, the sign is discarded and the two colours receive identical scores. This limitation is addressed in section 3.5.

3.3 Code phase tracking

Apart from the leap-byte error, the scrambler synchronisation model described above also suffers from the simplifying assumption that the bitrate of the DisplayPort link is exactly equal to the nominal 1.62Gbit/s specified by the standard. In reality, there is some clock inaccuracy up to manufacturer-defined tolerances, plus thermal drift, that will cause the link bitrate to deviate from the nominal value, and hence the synthesised scrambler will drift out of sync between scrambler resets. Scrambler tracking can be improved by instead maintaining an estimate of a time-variable bitrate $f_b(t)$, by periodically measuring the offset between tracked and real scrambler positions.

The tracking algorithm is based on a delay-locked loop that maintains an estimate of the bitrate $f_b[\ell]$. Once per line, the tracking algorithm measures the difference between the predicted scrambler position and the real scrambler position, this is known as the *code phase error*. This error is fed into a proportional-integral-derivative (PID) controller that

produces an estimate of the bitrate, that is in turn fed into Equation 3.4 so that the scrambler state estimate is as precise as possible.

For the first few hundred lines, tracking is in the acquisition phase. The code phase error is measured by aligning with the bytes of the horizontal blanking region. Since the bytes in this region have a known value (0x00), templates can be constructed for any scrambler byte offset $\delta \in \mathbb{R}$. We define the correlator against the blanking region at offset δ as:

$$R(\ell, \delta) = \max_{\rho \in \{+1, -1\}} \left| \sum_{k=0}^{N_b-1} s[n_l[\ell] + k] \overline{z(\sigma_\ell + \delta, \mathbf{0}, \rho)[k]} \right| \quad (3.10)$$

where $N_b = \lfloor n_b \cdot 10 f_s / f_b \rfloor$ is the number of received samples spanning n_b blanking bytes, and $n_l[\ell]$ is the estimated sample index of the start of line ℓ .

During the acquisition phase, $R(\ell, \delta)$ is evaluated over a range of offsets $\delta \in D$ and the best-matching offset is selected:

$$e_s[\ell] = \operatorname{argmax}_{\delta \in D} R(\ell, \delta) \quad (3.11)$$

During the tracking phase, the error is measured using an *early power minus late power* discriminator, which is cheaper to compute but only reliable while the code phase error is low. It computes the difference between the squared magnitude of an early and late correlator, offset by $\pm \Delta$ bytes from the current scrambler state estimate, normalised by the prompt power at zero offset:

$$e_f[\ell] = \frac{R(\ell, -\Delta)^2 - R(\ell, \Delta)^2}{R(\ell, 0)^2} \quad (3.12)$$

3.4 Carrier phase tracking

For a carrier at frequency f_c , a time offset of τ seconds corresponds to a phase rotation of $2\pi f_c \tau$ radians. At sample n , the scrambler state $\sigma[n]$ gives a time offset of $\sigma[n]/\bar{f}_B$ seconds, so an approximate phase correction can be computed as:

$$\Delta_\phi[n] = 2\pi f_c \cdot \sigma[n] / \bar{f}_B \quad (3.13)$$

where $\bar{f}_B = \bar{f}_b/10$ and $\bar{f}_b$ is the average tracked bitrate.

A second PLL then eliminates the remaining phase drift and initial offset. It measures the carrier phase error on each line by correlating against the blanking template, low-pass filters it, and feeds it into a PID controller that computes a fine tuning adjustment $\Delta'_\phi[\ell]$.

Let $z_b^\ell = z(\sigma_\ell, \mathbf{0}, \rho_0)$ denote the blanking template at line ℓ . The carrier phase tracking error is then:

$$e_\phi[\ell] = \angle \left(e^{-j\Delta'_\phi[\ell-1]} \sum_{k=0}^{N_l-1} s[n_l[\ell] + k] \overline{z_b^\ell[k]} e^{-j\Delta_\phi[n_l[\ell]+k]} \right) \quad (3.14)$$

where k ranges over the N_l samples spanning one full line, $e^{-j\Delta_\phi[n_l[\ell]+k]}$ is the coarse phase correction applied per sample, and $e^{-j\Delta'_\phi[\ell-1]}$ is the fine correction from the PID controller, updated once per line.

3.5 Coherent averaging

After carrier phase tracking, the signal and templates are phase-aligned, so the inner product is a *coherent* correlation. Colour scores can be averaged across frames coherently by summing complex values rather than magnitudes: signal contributions share a common phase and combine constructively, growing as N , while noise contributions have random phase and grow only as $\sqrt{N}$, improving the SNR by $\sqrt{N}$. The improved SNR means that the short term correlation window w used in Equation 3.7 is not longer necessary, and the correlation score reduces to a pointwise product:

$$u_c[n] = s[n] \overline{z_c^b[\sigma[n]]} \quad (3.15)$$

The appropriate template to use here is the *balanced template*:

$$z_c^b = \frac{z_c^+ + z_c^-}{2} \quad (3.16)$$

which retains only contributions from balanced 8b/10b blocks. To see why, note that across frames a given data byte is XORed with different scrambler values, and the running disparity may cause an unbalanced scrambled byte to be encoded as either $+x$ or $-x$ with roughly equal frequency. Coherently averaging these contributions is destructive, so unbalanced blocks contribute nothing to the average regardless.

Since balanced blocks are unaffected by running disparity, each block's contribution to z_c^b depends only on that byte's value and not on surrounding bytes. This independence enables an efficient implementation in which the template for any RGB colour can be expressed as a sum of seven terms drawn from a precomputed bank of 61 sub-templates.

As a consequence of preserving the sign of correlations, coherent averaging also resolves the complementary colour ambiguity noted in section 3.2: complementary colours, whose balanced templates satisfy $z_{-c}^b = -z_c^b$, now produce opposite-sign correlations and are distinguishable. After periodic averaging across frames, the colour of each pixel is decoded

as the colour with the largest real part:

$$\hat{c}[p] = \operatorname{argmax}_{c \in C} \Re \{ \text{scores}_c[p] \} \quad (3.17)$$

Chapter 4

Design and implementation

In the previous chapters, I introduced the DisplayPort standard and described the eavesdropping attack pipeline used to reconstruct displayed images from emissions. Leap bytes are a source of uncertainty in the scrambler synchronisation model, preventing full RGB decoding. In this chapter, I describe how I first measured the leap byte pattern, allowing me to extend the attack to fully decode RGB values. I then describe methods to automate the leap byte detection process, and how to extend to a full RGB decoding attack pipeline.

4.1 Detecting leap bytes

To satisfy a non-integer average line length, the transmitter extends the length of some lines by one byte: these are leap bytes. The effect of a leap byte is to shift all subsequent data bytes by one position, or equivalently, to clock the scrambler forward by one relative to the data bytes following the leap. Detecting leap bytes therefore requires aligning the signal against the *data* byte stream directly. To do this, we initially use known image content.

4.1.1 Target image

I used the test image designed by Erdeljan[6] shown in Figure 4.1, as it contains multiple features that could be of interest in targeting: decoding fonts tests the resolution of the attack, the monochromatic image demonstrates how the attack handles a range of visually similar colours, and the blocks of colour and cartoons allow test RGB decoding.

4.1.2 Horizontal and vertical alignment

Aligning with the data byte stream requires both horizontal and vertical alignment. Horizontal alignment identifies where each video line begins in the received signal; vertical alignment identifies which row of the frame each line corresponds to.

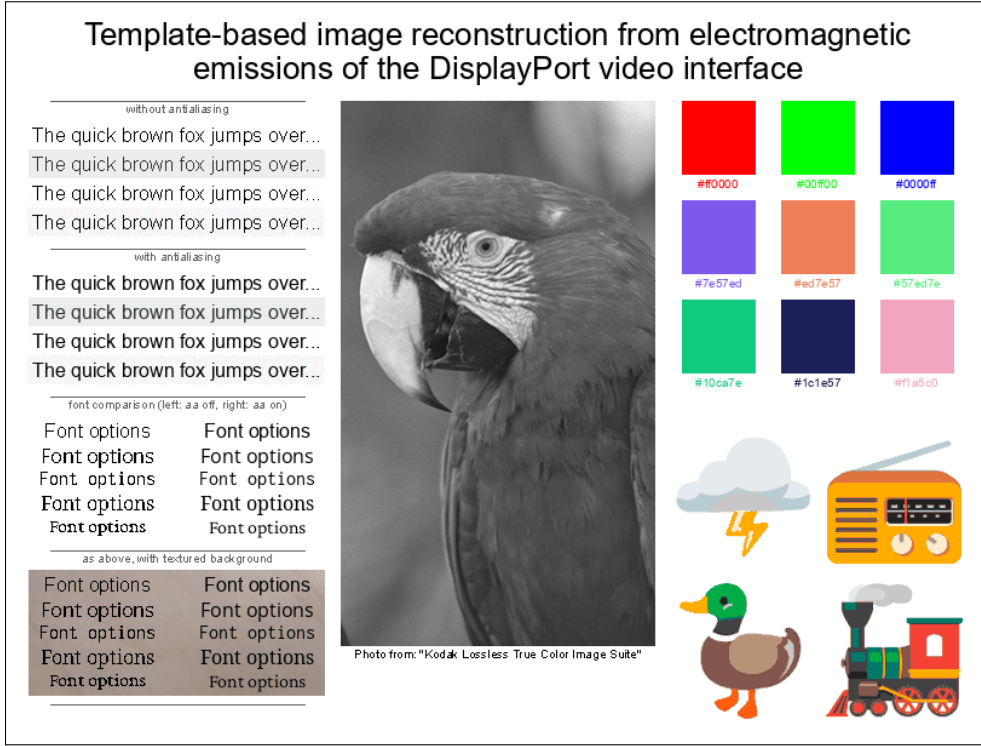


Figure 4.1: Target image [6].

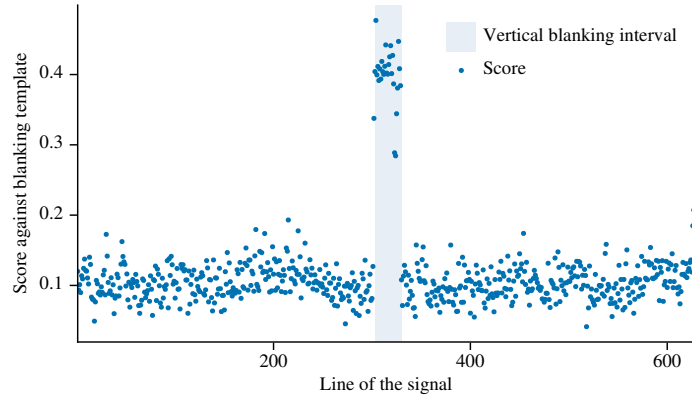


Figure 4.2: Vertical alignment is found by correlating each line against an all-zero template, and searching for the contiguous run of n_{vb} lines with the greatest total score. The lines of the vertical blanking interval are visible as a peak in the scores.

Horizontal alignment follows directly from the scrambler reset: the SR symbol is placed at the start of the horizontal blanking interval, so the reset timings recovered by the scrambler tracker mark horizontal blanking starts.

Vertical alignment is found by locating the vertical blanking interval, a contiguous band of n_{vb} lines in which every byte is zero-valued. Each line ℓ of the signal is correlated against an all-zero template $z(\sigma_\ell, \mathbf{0}, \rho)$; lines falling within the vertical blanking interval score highly, since their content matches the template everywhere. Searching for the contiguous run of n_{vb} rows with the greatest total score identifies the vertical blanking interval, and hence fixes the mapping between transmitted lines and frame rows.

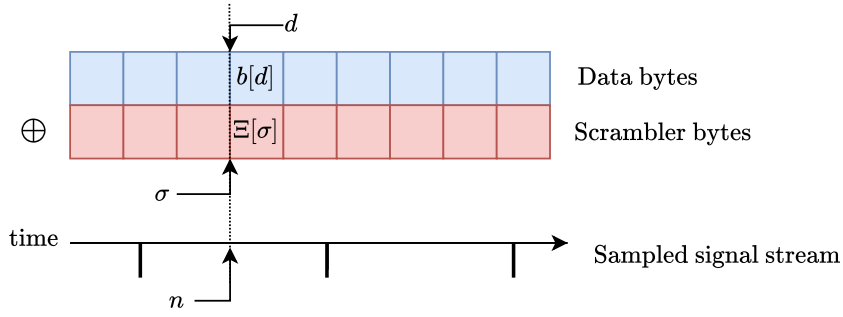


Figure 4.3: The three coordinate systems: the received signal sample index n , the scrambler state σ , and the data byte position d .

4.1.3 Fine alignment using grid search

Due to leap bytes, the scrambler state at the start of a line is not known a priori: up to n leaps across the preceding n lines since the scrambler reset may have additionally clocked the scrambler. The leap pattern must instead be inferred indirectly, by measuring how the scrambler state at the start of each line changes from one line to the next.

Define an *alignment* as a triple (n, σ, d) relating the three coordinate systems at a fixed point in time: the received signal sample index n , the scrambler state σ , and the data byte position d (Figure 4.3). All three may be real-valued — since byte and sample boundaries need not coincide. The goal of fine alignment is to recover a correct alignment for each line; the per-line scrambler states then reveal the leap pattern.

My initial approach was a local grid search. For each line ℓ , the scrambler tracker (Equation 3.4) provides an approximate signal index n_ℓ and scrambler state σ_ℓ for the start of the line, and the vertical alignment provides an approximate data byte position $\mathbf{b}_\ell$.

The true alignment is then refined by searching a small grid of candidate alignments around the seed.

The search has two dimensions, each correcting one degree of freedom in the seed alignment. The first is an integer scrambler offset $\Delta\sigma$, which shifts the scrambler state relative to the data bytes; this accounts for the unknown number of leap bytes accumulated across preceding lines. The bit-domain template for a candidate offset is generated by scrambling and encoding the line’s data bytes from the shifted scrambler state:

$$\xi_{\Delta\sigma} = \text{enc}(\text{scr}(\sigma_\ell + \Delta\sigma, \mathbf{b}_\ell), \rho_0) \quad (4.1)$$

The second dimension corrects the signal offset δ between the template and the received signal. It is convenient to parameterise δ into an integer part and a sub-sample part

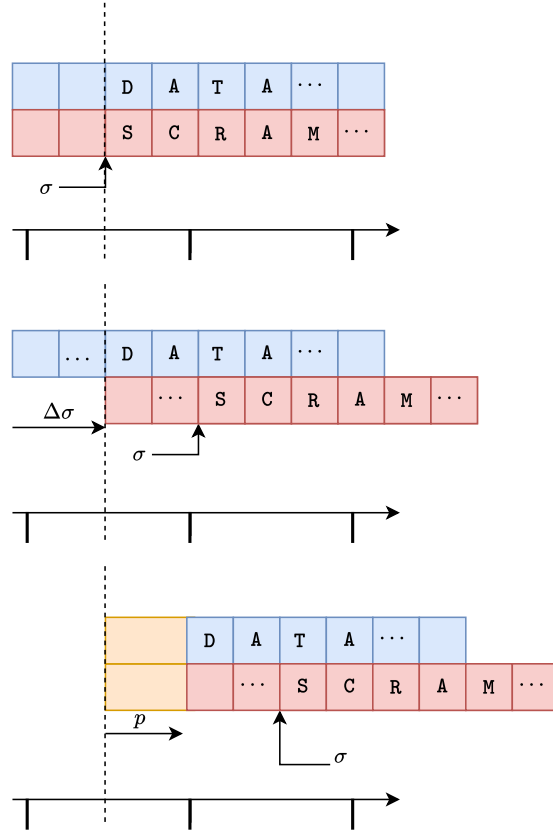


Figure 4.4: Fine alignment grid search template shifting.

$$\delta = m + p \frac{f_s}{f_b} \quad (4.2)$$

$$m \in \mathbb{Z}, \quad p \in [0, \lceil f_b/f_s \rceil)$$

as the two may be applied at different stages. The sub-sample part is realised by prepending p zero bits to the bitstream before downconversion and downsampling:

$$z_{\Delta\sigma, p} = \text{DS}_{f_s}((0^p \parallel \xi_{\Delta\sigma}) \cdot e^{-2\pi j n f_c / f_b}) \quad (4.3)$$

where $0^p \parallel \cdot$ denotes prepending p zero bits. Since one sample spans f_b/f_s bits, the range $0 \leq p < \lceil f_b/f_s \rceil$ covers all sub-sample shifts. The integer part m need not be searched by constructing further templates: all integer lags are obtained together as the lags of a single cross-correlation against $z_{\Delta\sigma, p}$, which avoids expensive template constructions.

Since the image content is known, the running-disparity ambiguity is resolved by taking the maximum scoring correlation over both possible initial starting states. The grid search maximises a single correlation magnitude over two axes: the integer scrambler offset $\Delta\sigma$ and the signal offset δ ,

$$r(\Delta\sigma, \delta) = \max_{\rho \in \{+1, -1\}} \left| \sum_{k=0}^{N_\ell-1} s[n_\ell + m + k] \overline{z_{\Delta\sigma, p}^\rho[k]} \right|, \quad \delta = m + p \frac{f_s}{f_b} \quad (4.4)$$

The grid cell $(\Delta\sigma_\ell^*, \delta_\ell^*)$ maximising r gives the refined alignment for line ℓ :

$$(n_\ell + \delta_\ell^*, \sigma_\ell + \Delta\sigma_\ell^*, d_\ell) \quad (4.5)$$

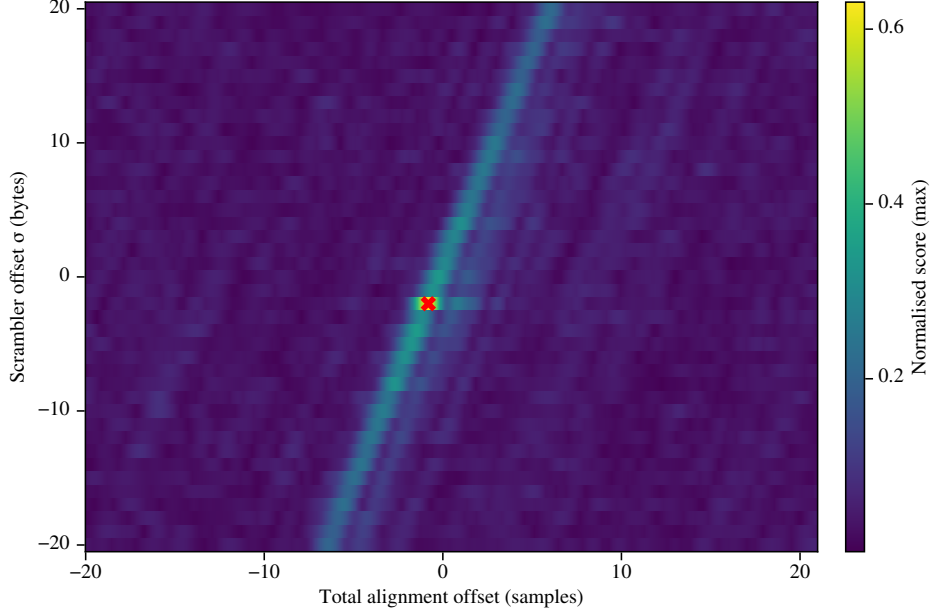


Figure 4.5: Output of the fine-alignment grid search for a single line. Each cell is the correlation magnitude $r(\Delta\sigma, \delta)$ between the received signal and a candidate template (Equation 4.4), maximised over the two running-disparity polarities. The vertical axis is the integer scrambler offset $\Delta\sigma$; the horizontal axis is the signal offset δ in samples. The refined alignment for the line is the cell of maximum correlation, marked $\times$.

The heatmap shows correlation scores concentrated along a diagonal with slope matching the bytes per sample rate. Along this diagonal, a shift of the scrambler relative to the data is paired with an opposite shift of the data and scrambler relative to the signal, with the net effect that the scrambler stays fixed relative to the signal while only the data bytes move. Where the diagonal passes through $(0, 0)$, the initial sample index n_ℓ and scrambler state σ_ℓ from the tracker are already well aligned. Since the true alignment lies within this band, the grid search can be restricted to the narrow diagonal strip: reducing the area that needs to be searched.

The variation along the diagonal reflects how discriminating the data bytes are for leap detection. Within a run of identical bytes, shifting the data stream changes the byte value only at the run’s boundaries; every interior position still holds the same value as before the shift. A long run of identical bytes — the zero-valued horizontal blanking interval, or a monochromatic image region — therefore contributes almost identical XORed values whether or not the data is shifted, and so are less discriminative between candidate alignments.

If the grid search fails to find a confident alignment for a line, i.e. no alignment produces

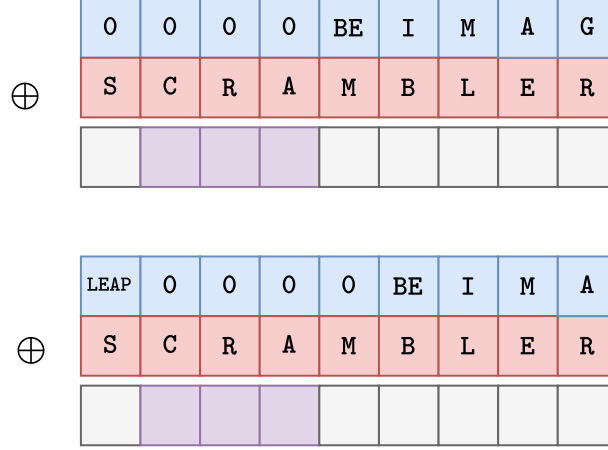


Figure 4.6: A leap byte shifts the data stream by one position, within a run of n identical bytes, only the two boundary positions change value; the $n - 1$ interior positions still hold the same byte, and so XOR with the scrambler to the same output as before the shift. XORed values that are unchanged by the leap are shown in purple.

a peak in the correlation scores, the line is marked as unaligned and excluded from leap pattern estimation.

Once a refined alignment has been found for a run of consecutive lines, the leap pattern is recovered from how the scrambler state advances between line starts. Let $\sigma_\ell^* = \sigma_\ell + \Delta\sigma_\ell^*$ be the refined scrambler state at the start of line ℓ . Between consecutive line starts the scrambler is clocked once per transmitted byte. Each line carries at least n_b bytes, where n_b is the number of bytes expected in the absence of leap bytes, any advance of the scrambler beyond n_b is attributable to leap bytes, modulo the scrambler period P .

$$\text{leaps}_\ell = [(\sigma_\ell^* - \sigma_{\ell-1}^*) \bmod P] - n_b \quad (4.6)$$

4.1.4 Bresenham sequence

Inspecting runs of detected leaps reveals a *Bresenham sequence*: a binary sequence describing the output of Bresenham’s line drawing algorithm [4]. The algorithm draws a straight line across a discrete pixel grid, and is equivalently a natural way to distribute a given number of leaps across a given number of lines so as to achieve a target average leap rate, and hence a target average line length.

A Bresenham sequence is the most even distribution of leap bytes across lines possible: it tracks a target leap rate as closely as a binary sequence can. Consequently, the number of leaps in any window of n consecutive lines is either $\lfloor n\alpha \rfloor$ or $\lceil n\alpha \rceil$, where α is the leap rate.

On the targeted laptop with Intel Skylake GT2 graphics controller in the 800×600 @

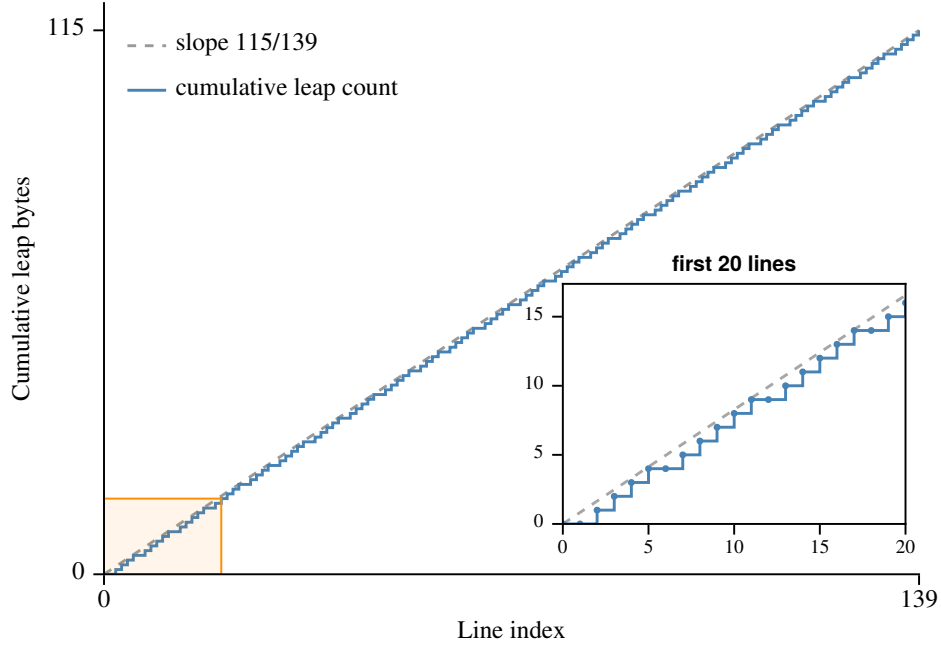


Figure 4.7: One period of a Bresenham sequence with 115 leaps across 139 lines.

60.3 fps video mode, the detected leap pattern was consistent with a Bresenham sequence of 115 leaps across 139 line period, targeting a rate of $\alpha = 115/139 \approx 0.8273$. We denote this sequence $\text{Bresenham}(139, 115, \phi)$, where ϕ is the initial phase — the starting position within the period.

Listing 4.1: Bresenham sequence generation algorithm.

```

1 function bresenham_sequence(period::Int, leaps::Int, phase::Int=0)
2     seq = zeros(Int, period)
3     acc = phase
4     for k in 1:period
5         acc += leaps
6         if acc >= period
7             acc -= period
8             seq[k] = 1
9         end
10    end
11    return seq
12 end

```

This pattern is verified operationally in section 4.2 by using it to predict the per-line leap bytes: the seed alignment it provides is sufficient to recover RGB content across every frame of the recording, at least $628 \times 120 = 75,360$ consecutive lines.

4.2 Colour reconstruction

The Bresenham(139,115) leap pattern hypothesis can be tested by attempting to reconstruct an RGB image using the predicted pattern. If the pattern is incorrect, the data bytes of the template will be misaligned with the scrambler and colours will decode as their cyclical rotations.

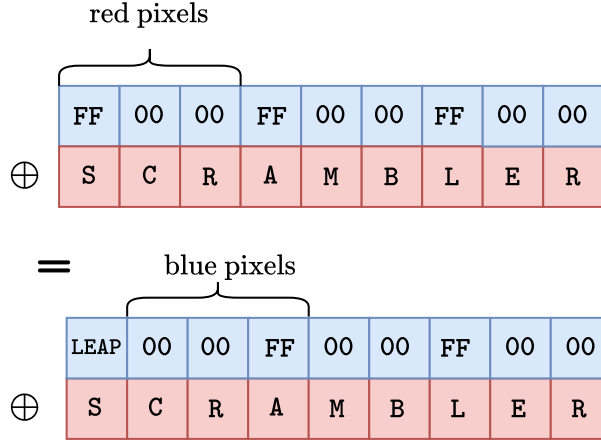


Figure 4.8: After a leap byte, the data bytes are shifted by one position relative to the signal. A template packed with red pixels (0xFF0000) that does not account for a leap (top) would actually detect blue pixels (0x0000FF), as the channel values are cyclically rotated by the shift.

For a single line, RGB-sensitive decoding is possible using the refined alignment from the grid search, which already accounts for any leap on that line. For each candidate colour of a target palette, a pair of templates is constructed, one assuming the initial running disparity is positive and the other assuming it is negative. The data bytes of each template are packed with the RGB values corresponding to the target colour, and it is aligned to the signal using the offsets found by the grid search.

Where a template matches the signal, the elementwise product has large magnitude; applying the windowed average of Equation 3.7 yields a per-colour score that can be checked against the known image content, as in Figure 4.9.

$$u_c[n] = \max_{\rho \in \{+1, -1\}} \left\{ \left| \sum_{k=0}^{w-1} s[n+k] \overline{z(\sigma, \mathbf{b}_c, \rho)[k]} \right| \right\} \quad (4.7)$$

4.2.1 Leap-aware line tracker

To obtain a seed alignment for every line without the grid search, I constructed a leap-aware line iterator. It extends Erdeljan's code phase tracker with knowledge of the Bre-

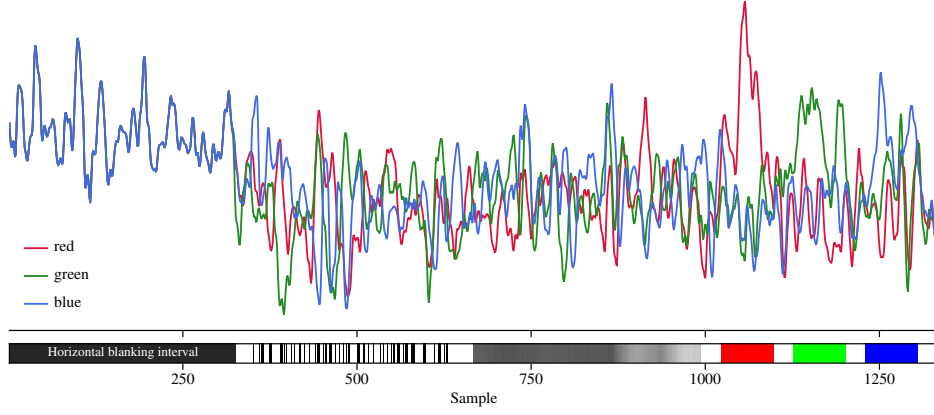


Figure 4.9: Template match score $u_c[n]$ of red, green and blue templates aligned against a single line and the corresponding image content of the target. The template scores show clear peaks at the positions of their corresponding colours in the image, demonstrating that the fine alignment has been successful.

senham parameters p and q and a starting leap phase, allowing it to predict line crossings precisely.

The initial state is found as before, by detecting a scrambler reset: at that point in the signal the scrambler is known to be in its initial state. Since the reset signalling occupies the first four bytes of a line, the tracker begins from data byte five.

The tracker then progresses much as the original model, measuring code phase error against blanking templates and feeding it into the PID controller that estimates the bitrate. The key difference from Equation 3.4 is that the scrambler state, rather than the signal index, is treated as the primary quantity: each step advances the scrambler state by one byte and the signal index by the current estimate of samples per byte. This makes data-level boundaries, in particular line crossings, straightforward to track.

Real-valued sample indices are handled at template construction time, by rounding to the nearest sample and compensating for the residual offset in the template construction, as in Equation 4.3.

With an alignment available for every line, the incoherent decoding of Equation 3.7 reliably recovers RGB content across multiple frames.

The window w over which correlations are summed is important. Since the magnitude is not taken until after the summation, the *complex* correlations are summed horizontally across the window. This improves signal power by constructively combining contributions from across neighbouring pixels, but also results in a horizontal smearing effect. A larger window would be able to decode large sections of colour with fewer frames, but would never resolve fine details such as text. A smaller window provides better resolution, but is more sensitive to noise. A comparison of the effect of the window size on the decoded image quality is shown in Figure 4.11.



Figure 4.10: RGB reconstruction of the target image using the leap-aware tracker. The reconstruction uses Equation 3.7 with a window size $w = 6$ and colour palette containing the 64 most common colours of the target image. The fill regions have been removed using the method described in subsection 4.2.3. The image is aligned vertically using the vertical blanking interval method of subsection 4.1.2.

4.2.2 Coherent RGB colour decoding

Coherent averaging across frames is enabled by the carrier phase tracking of section 3.4. Rather than summing correlation magnitudes across frames, the complex correlations are summed directly. The tracking estimates the carrier phase and de-rotates each correlation into a common phase reference. Once aligned, the signal contributions combine constructively, their summed amplitude growing as N over N frames, while the noise, having independent random phase, grows only as $\sqrt{N}$. The signal-to-noise amplitude ratio therefore improves as $\sqrt{N}$.

The improved SNR yields substantially better resolution, rendering text legible.

4.2.3 Removing fill regions

Decoding the template correlations directly leaves the fill regions visible in the reconstructed image.

Since the template is well aligned with the signal, these can largely be removed by mapping the colour score vector $u_c[n]$ from sample space into byte space: for each transmitted byte, a score is obtained by interpolating the sample-space scores around that byte's position.

Iterating through the byte-space score vector, bytes belonging to fill regions are skipped. Some residual contribution leak from the fill regions into the decoded image as the receiver



Figure 4.11: Effect of the window size w on windowed incoherent-averaging decode quality, at a fixed frame count, for three content types.

sample rate is lower than the 10-bit symbol rate, but this diminishes as scores are averaged across frames.

		3b/4b sub-block		Symbolic	
		byte b	$\rho = +$	$\rho = -$	
Balanced	0xc9	0110	0110	x	x
	0x36	1001	1001	$-x$	$-x$
Unbalanced	0xff	0111	1000	y	$-y$
	0x00	1101	0010	z	$-z$

Table 4.1: 3b/4b sub-block encodings of four example bytes under positive and negative running disparity, with a symbolic representation. $0x36 = \neg 0xc9$ and $0x00 = \neg 0xff$ are complementary byte pairs. For balanced bytes the encoding is disparity-invariant ($x = x$) and complementation negates it ($x \leftrightarrow -x$); for unbalanced bytes a disparity flip negates the encoding ($y \leftrightarrow -y$) while complementation gives an unrelated value (y vs z).

For an *unbalanced* sub-block, opposite disparities give complementary encodings: $\beta_{b,+} = \neg\beta_{b,-}$ (row y vs $-y$ in Table 4.1). While the complementary data byte $\neg b$ produces an unrelated unbalanced sub-block $\beta_{\neg b,\rho} \neq \neg\beta_{b,\rho}$ (column y vs z in Table 4.1).

In coherent averaging, unbalanced sub-blocks cancel with their opposite disparity encoding, as $\beta_{b,\rho} + \beta_{b,-\rho} = 0$, so coherent averaging can only see contributions from balanced sub-blocks.

Incoherent averaging takes the magnitude of the correlations: $|\beta_{b,\rho}| + |\beta_{b,-\rho}|$, so unbalanced sub-blocks do not cancel out, and balanced templates are not required.

Since the complementary byte $\neg b$ produces an unrelated unbalanced sub-block, incoherent averaging can distinguish the unbalanced sub-blocks of encoded complementary bytes.

In summary: coherent averaging sees only balanced sub-blocks, requiring balanced templates, but correctly handles all complement pairs. Incoherent averaging sees all sub-blocks but cannot distinguish fully balanced blocks from their complements. Each sample integrates contributions from several bytes, which will generically include a mix of balanced and unbalanced sub-blocks, and the unbalanced contributions restore complement sensitivity.

Balanced templates have the additional advantage of an efficient sub-template decomposition, which allows for 24-bit colour reconstruction with only 61 sub-templates. Full templates cannot be decomposed in this way; each byte propagates running disparity to the next byte, so the contributions per channel are not independent.

4.2.5 Full colour enumeration

Coherent averaging requires balanced templates, which also lets us exploit the sub-template colour enumeration of section 3.5: a template for any RGB colour is constructed as a sum of 61 precomputed sub-templates, so the full 2^{24} -colour space can be searched without enumerating colour templates explicitly.

This has two main advantages, the first is that it is a memory and time efficient way to represent the full colour space: lending itself well to continuous-tone target images such as the parrot in Figure 4.14. The second advantage is that it makes no prior assumptions about the image content: every representable colour competes on equal terms, making it well suited to an attack against unknown screen content.

The cost is convergence speed. With 2^{24} competing colours, many will produce non-negligible correlation scores by chance, so more frames of signal are required before the true colour accumulates a reliably dominant score. A fixed target palette, by contrast, encodes prior knowledge about the likely content: restricting the competition to K colours. Another approach could be to start with the full enumeration to identify a candidate palette, then rerun against only the candidate palette to converge more quickly.



Figure 4.14: Full colour enumeration results after 119 frames of coherent averaging using the sub-template method. This makes no prior assumptions about the target colour palette.

4.3 Refined leap byte detection

Leap byte detection can be made more robust and efficient by starting from the known scrambler state at the scrambler reset. On the first line after the reset there are only two possible scrambler states relative to the data, so only two templates need be compared: one assuming a leap byte in the line, and one assuming none. Each template is scored by the maximum of its sub-sample cross-correlation, using the same $\delta = m + p$ parameterisation as the grid search.

At the start of the j -th line after the scrambler reset the scrambler state is $j \cdot n_b + c(j)$

where $c(j)$ is the cumulative number of leap bytes since the reset. Since each line may or may not contain a leap byte, $c(j)$ may lie anywhere in $\{0, 1, \dots, j\}$, so the line has $j + 1$ possible scrambler start states. However assuming an underlying Bresenham sequence, the leap rate observed across preceding lines predicts the cumulative leap count, and the search can be narrowed to a window of k candidate states centred on that prediction. The parameter k trades search cost against robustness to an inaccurate rate estimate – too small a window risks excluding the true state when the running estimate is poor, while a larger window is more tolerant at proportionally greater cost.

A line is deemed to have discriminated the scrambler state if its best template scores some proportional margin better than the runner-up; lines failing this test are discarded and do not contribute to the fit.

The surviving lines are combined under the global constraint that the leap pattern is a Bresenham sequence parameterised by a rate p/q and a phase ϕ . A candidate Bresenham sequence $B = \text{Bresenham}(q, p, \phi)$ dictates a specific cumulative leap count on each line $c_B(j)$; each is scored by summing the scores predicted by the corresponding candidate scrambler states: $n \cdot n_b + c_B(j)$.

4.3.1 Leap byte detection without known image content

The eavesdropper can detect the leap pattern without known image content by applying the same approach, using only the fill region structure to discriminate between candidate leap patterns. Transfer units start and end at fixed offsets relative to the first data byte of a line. The effect of a leap byte on the fill region will be to change the scrambled bytes at each of the fill region boundaries. For the 800x600@60.3fps video mode, there are 50 transfer units per line, each with three bytes whose scrambled values are changed by a leap byte, so about 150 discriminating bytes per line. In order to discount noise from correlation with the image content, the template has all non-discriminating bytes zeroed out, so the correlation is only against the target bytes.

This enables an eavesdropper to characterise the leap byte rate $\alpha = p/q$ and phase ϕ to conduct an RGB attack without needing to have ever previously measured the leap pattern on that hardware using known image content.

4.4 Required precision of Bresenham parameters

The scrambler reset serves as an anchor for the scrambler state every 512 lines. Hence, we only need to be able to predict the cumulative number of leap bytes since that anchor.

Assume there is a true, underlying Bresenham sequence described by parameters q^* , p^* and ϕ^* . For each line n lines after the reset, the Bresenham sequence predicts a cumulative leap count $c^*(n)$ that defines the scrambler state at the start of each line as $n \cdot n_b + c^*(n)$.

The cumulative leap count predicted by the true sequence is

$$c^*(n) = \lfloor \frac{np^* + \phi^*}{q^*} \rfloor \quad (4.8)$$

We want q , p and ϕ such that the predicted cumulative leap count $c(n) = \lfloor \frac{np + \phi}{q} \rfloor$ is the same as the true cumulative leap count $c^*(n)$ for n in $\{1, 2, \dots, 512\}$.

It is convenient to work with the rate and normalised phase

$$\alpha = \frac{p}{q}, \quad \beta = \frac{\phi}{q}, \quad \alpha^* = \frac{p^*}{q^*}, \quad \beta^* = \frac{\phi^*}{q^*}, \quad (4.9)$$

so that $c(n) = \lfloor \alpha n + \beta \rfloor$ and $c^*(n) = \lfloor \alpha^* n + \beta^* \rfloor$. The two arguments differ by

$$e(n) = (\alpha - \alpha^*)n + (\beta - \beta^*), \quad (4.10)$$

and their floors coincide exactly when no integer separates them. Adding $e(n)$ to the true argument leaves its floor unchanged if and only if $e(n)$ does not push the fractional part $\{\alpha^* n + \beta^*\}$ past a unit boundary, that is, when

$$-\{\alpha^* n + \beta^*\} \leq e(n) < 1 - \{\alpha^* n + \beta^*\}. \quad (4.11)$$

A line violating (4.11) has its scrambler state mispredicted by at least one byte.

As n ranges over a reset block the iterates $\{\alpha^* n + \beta^*\}$ are spread approximately uniformly across $[0, 1)$, so the fractional part may be treated as uniform. Line n then fails with probability equal to the width of the excluded set, which for $|e(n)| < 1$ is $|e(n)|$. The expected number of mispredicted lines over $N = 512$ lines is

$$\mathbb{E}[\#\text{mispredicted}] \approx \sum_{n=1}^N |e(n)| \quad (4.12)$$

$$= \sum_{n=1}^N |(\alpha - \alpha^*)n + (\beta - \beta^*)| \quad (4.13)$$

$$\leq \frac{|\alpha - \alpha^*|N(N+1)}{2} + |\beta - \beta^*|N \quad (4.14)$$

So across an $N = 512$ line block, the expected number of mispredicted lines by α and β is approximately bounded by

$$|\alpha - \alpha^*|131328 + |\beta - \beta^*|512. \quad (4.15)$$

Assuming the rate is exact, $\alpha = \alpha^*$, we have

$$\mathbb{E}[\#\text{corrupted}] \approx N |\beta - \beta^*|. \quad (4.16)$$

Requiring fewer than k corrupted lines requires that the absolute error in the phase is less than k/N :

$$|\beta - \beta^*| < \frac{k}{N}. \quad (4.17)$$

Assuming instead that the phase is exact, $\beta = \beta^*$, we have

$$\mathbb{E}[\#\text{corrupted}] \approx \frac{1}{2} |\alpha - \alpha^*| N^2. \quad (4.18)$$

Requiring fewer than k corrupted lines requires that the absolute error in the rate is less than $2k/N^2$:

$$|\alpha - \alpha^*| < \frac{2k}{N^2}. \quad (4.19)$$

For $N = 512$ and $k = 1$ mispredictions, the estimated rate α should be precise up to $|\alpha - \alpha^*| < 7.6 \times 10^{-6}$ whereas the phase need only satisfy $|\beta - \beta^*| < 2.0 \times 10^{-3}$. The rate must be recovered to roughly one part in 10^5 – 10^6 ; the phase tolerance is much more lenient. Geometrically, α shifts the slope of the line drawn by the Bresenham algorithm, whereas β translates the line horizontally.

If α is precise enough, the error due to phase is always bounded: the constant term $\beta - \beta^*$ in $e(n)$ shifts the predicted leap count by at most one byte on lines scattered uniformly through the frame rather than clustered. A rate error instead grows without bound and eventually mispredicts whole contiguous bands by several bytes.

Therefore, although it can be estimated using the method described in subsection 4.3.1, the phase may not need to be estimated at all. A random β differs from the true phase β^* by, on average, a quarter of a cell: the mean circular distance between two independent uniform phases is $1/4$. It therefore corrupts on the order of $1/4$ of lines, but each misprediction is isolated and one byte off. The corrupted set is different across frames: so on average a random leap phase will still decode a line correctly more often than not.

This leaves only the rate α to be estimated with high precision. The average length of a line should be $n_b + \alpha^*$, so α^* can be read off from any accurate estimate of the mean line length. Such an estimate is provided by the scrambler reset model of Equation 3.3, which fits R as the average length of a 512-line period and P as the average length of the scrambler overflow period spanning $2^{16} - 1$ bytes. The empirical mean line length is thus

$$\hat{n}_b = \frac{R/512}{P/2^{16}-1} \quad (4.20)$$

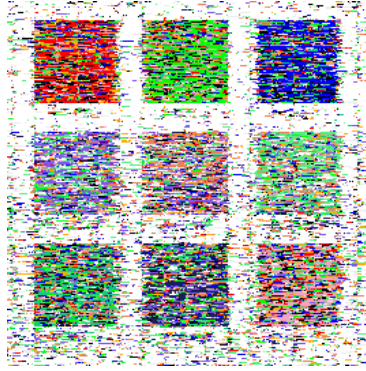


Figure 4.15: Incoherent RGB reconstruction using wrong initial leap phase across 12 frames.

and the leap rate α can be taken to be the fractional part of $\hat{n}_b$:

$$\alpha = \hat{n}_b - \lfloor \hat{n}_b \rfloor. \quad (4.21)$$

When this is accurate enough to decode RGB, it not only reduces the processing cost of the attack by eliminating the leap byte characterisation stage, but also makes no assumptions about the transfer unit structure.

Chapter 5

Evaluation

In the previous chapter I described how I measured the leap-byte pattern for a particular hardware target, and methods to automate the leap pattern and phase characterisation. Once the leap pattern is characterised for a given piece of hardware in some video mode, full RGB decoding is possible.

In this chapter I present the results of applying the RGB attack pipeline.

5.1 Experimental setup

The algorithms described in the previous chapter were developed using recordings inherited from Erdeljan [6, 7], from which the leap pattern parameters of $p = 115$ leaps per $q = 139$ lines were identified. That dataset was captured from a laptop with Intel Skylake GT2 graphics controller, driving an 800×600 display at 60.3 fps over a single lane at 1.62 Gbit/s with 64-byte transfer units in enhanced framing mode, using a directional antenna at a range of up to 8 m. This evaluation focuses on a dataset I captured, demonstrating the RGB attack pipeline on distinct hardware, and at an extended colour depth of 10 bits per channel.

5.1.1 Capture hardware

The receiver was a USRP X300 software-defined radio fitted with a UBX-160 daughter-board. Captures were taken at a centre frequency of 350 MHz with a sample rate of 50 Msps, with the daughterboard clock reduced to 20 MHz as recommended for the UBX-160 below 1 GHz. At this sample rate and a link rate of 1.62 Gbit/s, each captured sample spans roughly 3.2 bytes on the wire (≈ 0.31 samples per byte), so individual bits are not resolved and recovery relies on template correlation. The resulting data rate of approximately 400 MB/s is carried to the receiving desktop’s NIC over a 10 Gbit/s fibre-optic Ethernet connection.



Figure 5.1: Near-field H-probe held near the DisplayPort cable connector

The emissions were coupled into the receiver with a near-field probe held close to the cable connector, capturing the field directly at the cable. The probe incorporates a powered amplifier ahead of the receiver; its output was checked on an oscilloscope to confirm it remained within the receiver’s safe input range. A near-field probe was used in preference to an antenna because the goal was to demonstrate the RGB attack concept and study how leaps occur across video modes, rather than to demonstrate a long-range attack — for which the Erdeljan dataset already provides evidence.

5.1.2 Source hardware

I captured the dataset from an Intel HD Graphics 630 (Kaby Lake GT2) integrated GPU,¹ driving an LG32UD59 monitor. The displayed content during capture was the test image of Figure 4.1.

The Intel HD Graphics 630 was one device I attempted to capture; scrambler synchronisation succeeded reliably on only this one. The other devices and setup configurations are detailed in Appendix A.

5.2 10-bit colour depth

When running initial leap byte detection on the desktop recordings, I came across a strange issue in comparison to the Erdeljan dataset: the templates based on known image content were not confident in their detections, in particular, no initial scrambler state performed much better than its competitors, even when using a whole image line of discriminative

¹Desktop computer; ASRock Q270M-C motherboard, Intel Core i7-7700 processor.

data. Decoding the image using the original pipeline yielded the correct displayed image (of course, without correct colour), so the correct image had been displayed at the time of recording. The image even showed strong signal to noise ratio, presumably due to the near-field probe. When I tried the fill region boundary leap detection (subsection 4.3.1), the templates were similarly unhelpful – implying that the positions of the fill region boundaries in my template were not matching the positions in the signal.

I ran the decoding pipeline against a recording of the 1440x480 video mode. This mode has the feature that it should have exactly 100 per cent utilisation at 1.62 Gbit/s, so there should be no transfer unit stuffing with fill bytes. However, when I decoded the image, the fill regions were clearly visible ².

I discovered that the monitor was enforcing a colour depth of 10 bits: yielding a different data encoding with higher data rate. This meant that the M parameter of the TU stuffing algorithm (subsection 2.2.1) was different, and the positions of fill regions would be shifted. In the case of the 1440x480 video mode, the increase in data rate had pushed the link over the threshold of a single lane, to using two lanes instead. After accounting for 10-bit colour depth and potential multiple lanes in my templates, both leap detections based on known image content and based on only fill region boundaries worked. Colour reconstruction similarly required templates to account for the different data encoding and TU stuffing pattern: once that was done, I was able to decode RGB on the desktop recordings as well. As a result of this, another contribution of this work is a demonstration of the RGB attack on 10-bit colour depth.

5.3 RGB attack pipeline

The RGB attack pipeline extends the original DisplayPort eavesdropping attack (chapter 3) to include leap rate estimation before colour decoding. Once the leap pattern has been estimated, the scrambler tracker additionally models the effect of leap bytes in the data stream, to maintain byte-accurate synchronisation and enable RGB decoding.

The most efficient way to estimate the leap pattern is to use the average line length as estimated from the scrambler reset model (Equation 4.20). The reliability of this depends on how well the scrambler reset model fits the data. This method does not give an estimate for the leap pattern phase, for which incoherent averaging suffers more than coherent averaging.

Hence, if the eavesdropper is targeting RGB decoding using incoherent averaging, or more compute time is available, the leap pattern can be measured directly using the fill region boundary method described in subsection 4.3.1, which also gives an estimate of the phase.

²A consequence of this analysis was a data-based TU size discovery mechanism: transfer unit size and utilisation ρ can be approximated based on the decoding results.

5.4 Attack results

In these results, an RGB reconstruction is judged successful if the three primary colour blocks in the test image — red, green, and blue — are each recovered in a colour that a viewer would unambiguously categorise as that primary, i.e. the red block reads as red, the green as green, and the blue as blue, distinct from one another and from grey.

This criterion is a deliberately coarse proxy for leap synchronisation rather than a measure of reconstruction fidelity. The primary colour blocks are the most demanding content for colour decoding: under incorrect leap alignment, a one-byte shift cyclically permutes the R, G, and B channel bytes, so a misaligned red block decodes as green, blue, and greyscale (R=G=B) regions — which are invariant to such shifts — dominate the argmax. Correct, distinct recovery of the three primaries therefore implies the channel-to-byte alignment is sufficiently accurate. Conversely, if leap synchronisation has failed, the primaries collapse toward grey or a mixture of each other.

5.4.1 Intel HD Graphics 630 and 10-bit colour depth

Across the Intel HD Graphics 630 recordings captured with a near-field probe, the leap rate was estimated directly from the synchronisation parameters P and R , by taking the fractional part of the estimated average line length and approximating p and q by continued-fraction expansion. The resulting rate was in each case close — within approximately 6×10^{-5} — to the $p/q = 115/139$ rate that had separately been measured for the same video mode on different hardware. Decoding with 115/139 was indeed also sufficient to recover RGB on this hardware.

Using this estimate with a random initial leap phase, both coherent and incoherent averaging recovered RGB successfully. In the incoherent case the 10-bit colour depth introduced a black-and-white smearing artefact, the cause of which has not been investigated. In general, more work on 10-bit colour depth is needed, as the 3.75-bytes-per-pixel packing breaks the clean channel-per-byte alignment that the 8-bit pipeline relies on. A representative reconstruction is shown in Figure 5.2.

5.4.2 Leap characterisation across distance

To evaluate leap characterisation under realistic capture conditions, the pipeline was applied to Erdeljan’s 800×600 recordings, captured with a directional antenna at ranges from 3 to 8 m. For each recording the leap rate was both estimated directly from the synchronisation parameters P and R , and measured independently from the signal using the fill-region boundary method.

The fill-region boundary method recovered a leap rate close to the $115/139 \approx 0.82734$ fingerprint identified for this hardware at every distance, with a deviation of at most



(a) Decoded as 8-bit colour depth: vertical strips are missing where the 8-bit assumption mispredicts the fill-region placement. (b) Coherent decode using the leap rate estimated from P and R . (c) Incoherent decode using the P/R rate estimate; note the black-and-white smearing artefact.

Figure 5.2: RGB reconstruction of the 10-bit Intel HD Graphics 630 recording. (a) assuming 8-bit colour depth recovers monochrome features but misplaces the fill regions; (b) and (c) use the leap rate estimated directly from the synchronisation parameters P and R , decoded coherently and incoherently.

6×10^{-4} and, for the 3m recording, an exact match. The recovered rate did not drift systematically with range: the leap pattern is a fixed property of the source hardware, and is reproducible from independent captures even as the signal-to-noise ratio degrades with distance. This is consistent with the interpretation of the leap period as a fingerprint of the source’s clock-generation hardware rather than an artefact of any particular recording.

On this dataset, the estimate obtained directly from the reset-model parameters P and R was systematically higher than the true rate, by between 2×10^{-3} and 5×10^{-3} across the recordings. The reset-model estimate is attractive because it requires no separate leap measurement, however on these recordings it is too coarse to be relied upon alone; the fill-region boundary measurement, although more expensive, recovers the rate to sufficient precision.

5.5 Remaining limitations of the RGB colour attack

More work needs to be done to understand the required precision in the leap rate α and the initial leap pattern phase β for successful RGB decoding. From practical experience, as the error in the leap rate estimate increases the results get worse: an error of 0.005 can completely lose scrambler synchronisation and fail to decode any image structure, however the exact relationship between leap rate error and decoding performance is yet to be fully understood.

Running the leap detection method across the whole signal is computationally expensive, so it would be useful to define bounds that identify convergence on leap pattern detection.

It would be interesting to investigate whether the leap rate $\alpha \approx 115/139$ that worked across both hardware targets is coincidental, or it is derived from a shared hardware property or driver implementation. More work also needs to be done to understand

whether and how the identified leap pattern can be generalised for use across video modes.

Chapter 6

Summary and conclusions

In this dissertation, I have presented an extension to the DisplayPort eavesdropping attack that enables a colour reconstruction that distinguishes between the red, green and blue colour channels of a pixel. I first described the existence of leap bytes as an underspecified feature of the DisplayPort standard: every line must have a discrete number of bytes, and yet a non-integer average line length must be satisfied. This presented itself as a gap in the original attack pipeline — scrambler synchronisation did not know how to account for leap bytes, and so absorbed the uncertainty in its position, rendering RGB decoding impossible. My work describes how I first measured the pattern of leap bytes, and was able to use this to reliably distinguish colour channels. After understanding the structure of the leap pattern as a Bresenham sequence, I developed methods to automate the leap pattern characterisation that extend the existing attack pipeline to RGB in general.

6.1 Future work

The remaining limitations of the attack can be broadly categorised into limitations of the attack concept and limitations of the implementation.

Limitations of the attack concept are features or modes that the attack has not yet been designed to work with, for example, video modes with a higher lane count, 10-bit colour depth, or sources with spread-spectrum clocking enabled. Limitations of the implementation include efficiency improvements, improvements in convergence rate, a palette discovery mechanism, and more practical demonstrations across different hardware at varying distances.

6.1.1 Multi-lane video modes

Video modes with a higher lane count split pixels across multiple lanes, transmitting bytes in parallel. From practical experience, at least some structure in the transmission is already recoverable using the existing attack pipeline, that assumes a single lane. For

example, the transfer unit fill regions, which occur at the same byte offset across lanes, are visible. The leap byte detection method based on fill region boundaries is already compatible with two lanes: after packing data across multiple lanes, the bitstreams of each lane are independently generated, their emissions are generated, with inter-lane skew applied, and the resulting templates are summed additively. This is relatively straightforward as the leap detection method relies on known image content (or the known presence of fill regions) – the challenge with a multi-lane image reconstruction attack would be how to enumerate candidate templates against multiple lanes efficiently. A multi-lane attack would enable a wider range of potential targets and would be a significant step towards a practical attack, as the high data rate required to drive modern displays is likely to require multiple lanes.

6.1.2 Spread-spectrum clocking

Spread-spectrum clocking (SSC) is a technique DisplayPort supports [11] to reduce electromagnetic interference of a device by spreading the clock signal over a range of frequencies. Instead of the link rate aiming to hold a strict, single frequency (such as exactly 1.62 GHz) SSC intentionally shifts the clock frequency back and forth around that target, in a triangular wave pattern, in order to widen the signal’s spectrum. The current bitrate tracking mechanism does not model this, and the magnitude and frequency of the SSC shift would be significantly out of range of the currently permitted variation. This could be accounted for by additionally modelling the bitrate with an SSC triangular wave and fitting the parameters of that.

6.1.3 Efficiency and convergence improvements

There is significant scope to improve the efficiency of the attack pipeline, in particular, colour decoding is highly parallelisable, as each line and colour template can be processed independently. The attack should therefore benefit substantially from an implementation that takes advantage of this parallelism, for example, by writing the attack pipeline for GPU.

Convergence improvements would allow the attack to reduce the required signal duration (i.e. fewer frames) or compute time. Defining some probabilistic bounds on the confidence of a given pixel decoding could allow an eavesdropper to reduce that pixel’s target colour palette, or stop processing a given image region once a certain confidence threshold in its confidence is reached. Both of these approaches aim to reduce the total compute time by avoiding redundant computation. Reducing the number of colour templates that need to be tested is already being done implicitly by providing an initial target colour palette (except in the case of full 24-bit colour enumeration). Against a target with no known colour palette, the 24-bit colour enumeration could be used as a palette discovery mechanism, where the most commonly decoded colours are added to the target palette

and the attack is rerun with that refined palette, which should require fewer frames and compute time to converge against noise.

Bibliography

- [1] Tempest-time. Cryptome, <https://cryptome.wikileaks.org/tempest-time.htm>, accessed 2026-05-26
- [2] Barnes, J.: drm/i915: prefer wide & slow to fast & narrow in DP configs. Linux kernel commit 2514bc510d0c (Jun 2012), <https://github.com/torvalds/linux/commit/2514bc510d0c3aadcc5204056bb440fa36845147>, accessed 2026-05-26
- [3] Boak, D.G.: A history of U.S. communications security. National Security Agency Technical Report (1973), declassified 2008. Available at https://www.nsa.gov/portals/75/documents/news-features/declassified-documents/cryptologic-histories/history_us_comms.pdf
- [4] Bresenham, J.E.: Algorithm for computer control of a digital plotter. In: Seminal graphics: pioneering efforts that shaped the field, pp. 1–6 (1998)
- [5] Demoulin, B., Kone, L., Poudroux, C., et al.: Electromagnetic radiation of shielded data transmission lines. In: Symposium on Electromagnetic Security for Information Protection. vol. 173. Rome, Italy (1991)
- [6] Erdeljan, D.: Eavesdropping risks of the displayport video interface (2023), <https://www.repository.cam.ac.uk/handle/1810/364583>
- [7] Erdeljan, D., Kuhn, M.G.: Electromagnetic Eavesdropping on DisplayPort. IEEE Transactions on Dependable and Secure Computing **23**(03), 5404–5421 (May 2026). <https://doi.org/10.1109/TDSC.2026.3653840>, <https://doi.ieeecomputersociety.org/10.1109/TDSC.2026.3653840>
- [8] Gonggrijp, R., Hengeveld, W.J.: Studying the nedap/groenendaal es3b voting computer: A computer security perspective. In: Proceedings of the USENIX workshop on accurate electronic voting technology. pp. 1–1 (2007)
- [9] Kreml, V.S.: Lauschangriff am Geldautomaten (Jan 1999), <https://web.archive.org/web/20001031024042/http://www.spiegel.de/netzwelt/technologie/0,1518,13731,00.html>
- [10] Kuhn, M.G.: Ballot printer – protection against eavesdropping attacks – guidance for system designers. Kamerstuk, bijlage 2016d17984, Tweede Kamer der

- Staten-Generaal (House of Representatives, Netherlands) (Apr 2016), <https://www.tweedekamer.nl/kamerstukken/detail?id=2016D17984&did=2016D17984>, richtlijnen ter voorkoming van compromitterende straling
- [11] Lee, H.C., Lee, S.W., Kang, J.K.: Spread spectrum clock generator for displayport. In: 2008 International SoC Design Conference. vol. 02, pp. II-5–II-8 (2008). <https://doi.org/10.1109/SOCDC.2008.4815670>
 - [12] National Security Agency: Tempest: A signal problem (1972), <https://www.nsa.gov/portals/75/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf>, originally published in Cryptologic Spectrum, Summer 1972, Vol. 2, No. 3; declassified and released via FOIA in 2007
 - [13] Smulders, P.: The threat of information theft by reception of electromagnetic radiation from rs-232 cables. *Computers & Security* **9**(1), 53–58 (1990)
 - [14] Van Eck, W.: Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security* **4**(4), 269–286 (1985)
 - [15] Video Electronics Standards Association (VESA): Displayport™ standard version 1.2. Tech. Rep. Ver.1.2, Video Electronics Standards Association, Milpitas, CA (2010), <https://glenwing.github.io/docs/DP-1.2.pdf>
 - [16] Wikipedia contributors: Matched filter — Wikipedia, the free encyclopedia (2026), https://en.wikipedia.org/wiki/Matched_filter

Appendix A

Device screening

I attempted to capture the DisplayPort emissions of several devices. Synchronisation — the first stage of the pipeline — succeeded reliably on only the Intel HD Graphics 630 desktop GPU used for the evaluation in chapter 5; the others failed to synchronise. The devices and their capture arrangements are listed in Table A.1.

Device	GPU chipset	/	Form	Monitor	Setup notes	Sync.
Desktop	Intel Graphics 630	HD	Desktop	LG 32UD59	DP-DP; monitor is 10-bit depth	✓
Mac Mini	Intel Graphics 4000	HD	Desktop	iiyama ProLite XUB2495WSU LCD	FireWire	×
Dell XPS	Intel Iris Xe Graphics (Alder Lake GT2)		Laptop	As above	DP over USB-C to dock, then DP- DP; probed at DP connector	×
MacBook Pro 2017	Intel Iris Plus Graphics 640		Laptop	As above	As above	×

Table A.1: Device and setups attempted. Devices and setups attempted, captured with a near-field probe (the desktop was additionally tried with Yagi Sinclair and Schwarzbeck log-periodic antennas). Synchronisation succeeded on only the desktop GPU, using any antenna