tempest-helper: Calm avant la Tempest

Valentin Buffa, Louis Distel et Léo Iagoridcov

Université de Rennes, Cyberschool
{valentin.buffa, louis.distel, leo.iagoridcov}@etudiant.univ-rennes1.fr

Sous la supervision de:
Christophe MOY, Pierre GRANIER, François SARRAZIN et Jordane LORANDEL

Abstract. Les attaques TEMPEST exploitent les émissions électromagnétiques involontaires des appareils électroniques pour en extirper des informations sensibles. Les supports transportant un flux vidéo ne font pas exception et certains logiciels permettent d'espionner le contenu d'un écran. L'objectif est de développer un outil automatisé pour découvrir la fréquence d'horloge des pixels (pixel clock) d'un écran à partir des ondes électromagnétiques émises par le câble HDMI. Cette information est cruciale pour les attaques TEMPEST, car il est rare de connaître les paramètres d'un écran que nous souhaitons prendre pour cible. Ce projet a pour but de simplifier la réalisation de ce type d'attaque et d'ouvrir la voie à de nouvelles recherches pour l'amélioration de la distance d'exploitation.

Keywords: Tempest · Software Defined Radio · HDMI leakage.

1 Introduction

TEMPEST est un nom de code attribué par la NSA et une certification de l'OTAN désignant l'espionnage des systèmes d'information par le biais de fuites non intentionnelles, notamment des signaux radio ou électriques et mêmes sonores. Le programme TEMPEST a été initié dans les années 1950 suite à la découverte par les gouvernements de l'espionnage soviétique interceptant des communications classifiées en captant les émissions électromagnétiques des machines à écrire électriques.

La découverte de la vulnérabilité des machines à écrire électriques a conduit à des recherches approfondies sur les émissions électromagnétiques des autres appareils électroniques. C'est dans ce contexte que Willem van Eck, un ingénieur néerlandais, a réalisé une avancée majeure en 1985. Il a démontré qu'il était possible de récupérer des informations sensibles à partir des émissions électromagnétiques des écrans d'ordinateur cathodiques [2]. Cette découverte, connue sous le nom de van Eck Phreaking, a eu un impact considérable sur le domaine de la sécurité TEMPEST. Son travail a permis de sensibiliser à la menace des fuites d'informations par émissions électromagnétiques et a conduit à un développement accru des technologies et des procédures TEMPEST pour protéger les informations sensibles (cf. SDIP-27 [1]).

Notre projet se focalise sur l'attaque contre les écrans d'ordinateur et plus précisément sur le câble HDMI transitant l'image de l'unité centrale à l'écran. Le câble HDMI émet des ondes électromagnétiques que nous captons, enregistrons et exploitons pour reconstituer l'image envoyée par le câble.

Le dispositif d'attaque, illustré par la figure 2, repose sur deux éléments: une antenne et une plateforme SDR (Software Defined Radio). L'antenne, pointée vers le câble HDMI à quelques centimètres de distance,

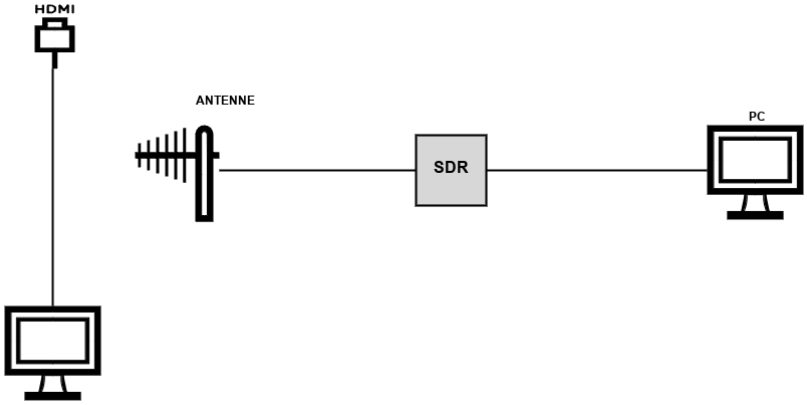


Fig. 2: schéma d'attaque sur cable HDMI

capte les ondes électromagnétiques émises involontairement par le câble. Elle est ensuite connectée à la plateforme SDR, qui elle-même est reliée à l'ordinateur. Deux aspects distincts composent cette attaque: la partie matérielle et la partie logicielle.

La partie matérielle comprend l'antenne et la plateforme SDR. Le choix de l'antenne est crucial pour augmenter la portée de l'attaque. Une antenne directive permet de focaliser la réception dans la direction du câble HDMI, autorisant une distance d'écoute supérieure à celle d'une antenne fouet. Cependant, nos expérimentations n'ont pas permis d'obtenir des résultats concluants à plus d'un mètre du câble.

La partie logicielle repose sur GNU-Radio, un logiciel de traitement du signal. Il permet de créer des "flowgraphs", composés de blocs connectés entre eux et assumant chacun une fonction spécifique (définition de la source radio, application de filtres, enregistrement des données...). L'exécution du flowgraph se traduit par sa conversion en Python, puis son exécution automatique. GNU-Radio propose une interface graphique intuitive pour la création des flowgraphs, sans nécessité de codage. Néanmoins, le code Python généré lors de l'exécution est conservé dans les fichiers de GNU-Radio, un aspect crucial détaillé ultérieurement.

La reconstruction de l'image à partir des ondes électromagnétiques captées est possible grâce à des logiciels spécifiques, exploitant le principe du raster vidéo (vidéo matricielle). Deux logiciels principaux existent : gr-tempest [4] et TempestSDR [5]. Le premier est basé sur GNU-Radio, tandis que le second s'articule autour d'une bibliothèque C, de plugins pour différents SDR et d'une interface utilisateur Java. Ces logiciels nécessitent de renseigner de plusieurs informations très précisément pour fonctionner:

- La résolution de l'écran (Horizontale et Verticale)
- Le nombre d'images par seconde affiché par l'écran (FPS)
- La fréquence d'écoute appelée pixel clock (Freq)

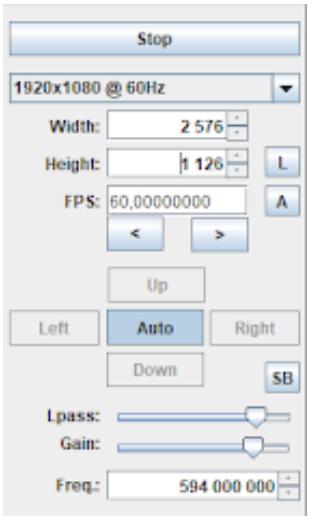


Fig. 3: Paramètrage de TempestSDR

Si le paramétrage du logiciel n'est pas bon, aucune reconstruction d'image n'est possible. Le paramètre le plus important dans l'attaque est d'avoir la bonne fréquence à laquel écouter, c'est-à-dire, la bonne pixel clock. Définissons ce qu'est-ce que la pixel clock : "Il s'agit d'un signal électrique qui contrôle le minutage et la synchronisation des pixels d'un écran. La pixel clock est exprimée en hertz (Hz) et correspond au *nombre de pixels envoyés par seconde*."

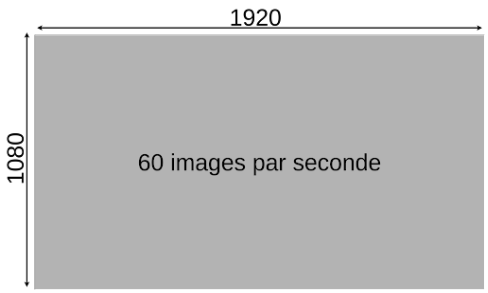


Fig. 4: Calcul de la pixel clock

Selon la définition et ce que l'on observe sur la figure 4, le calcul naïf de la pixel clock serait $pixel_clock = H * V * FPS$ cependant nous allons voir qu'il n'est en fait pas si simple.

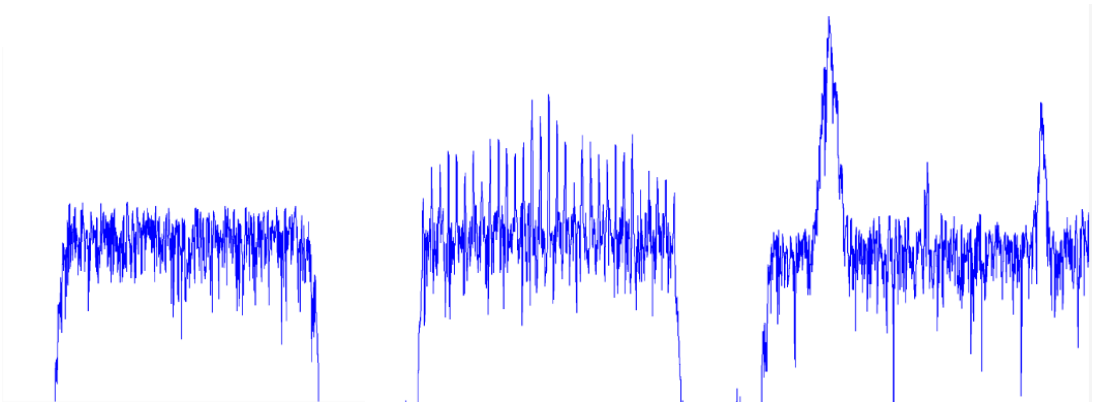
La problématique que l'on observe est que pour réaliser l'attaque il faut connaître beaucoup de paramètres au préalable. Il existe deux logiciels permettant de réaliser une la reconstruction d'image lors d'une attaque TEMPEST, mais il n'existe aucun outil permettant de trouver le pixel clock à partir des ondes électromagnétiques émises par le câble HDMI.

Le sujet de notre projet est de développer un outil permettant de trouver automatiquement la pixel clock d'un écran sans aucune information préalable. Notre outil constitue donc une étape préliminaire à l'attaque, le Calm qui vient avant la Tempest.

2 Idées de fonctionnement

Le sujet de notre projet s'est déssiné lors de nos expérimentations des différents logiciels de raster. En effet, lors de nos premières expérimentations nous avons passé un temps considérable à tenter de faire fonctionner gr-tempest sans grand succès, et nous sommes alors dirigé vers TempestSDR. Nous avions à l'esprit de modifier ce premier raster, ayant des briques de python, donc bien plus modulable que le second. Cependant l'impossibilité de le faire fonctionner nous a dirigé vers son homologue en C# pour la partie raster. C'est alors que nous avons compris l'empleur de la tâche après à nouveau un temps considérable pour réussir à le paramétrer correctement. Par celà nous entendons particulièrement quatre paramètres: la fréquence (pixel clock), largeur d'écran, hauteur d'écran et taux de rafraîchissement d'écran. Le paramètre le plus crucial dans cette liste est le premier, que nous appellerons la pixel clock. La pixel clock comme son nom l'indique définie à quelle fréquence chaque pixel apparaîtra. De celui-ci il est possible de modifier les autres paramètres du raster afin de retrouver un écran, cependant sans la pixel clock l'inverse est impossible. Nous avons décidé alors de reprendre le fonctionnement de gr-tempest (même si inutilisable empiriquement, fonctionne en théorie).

Notre première approche fût de créer des flowgraphs en utilisant GNU Radio Companion (ou GRC). Cependant malgré la simplicité d'utilisation et la grande quantité de blocks, l'absence de contrôle du flux d'exécution nous a rapidement posé problème. En effet nous souhaitons itérer sur une liste prédéfinie de différents candidats et aucun block ne semble proposer cette fonctionnalité. Il est possible de développer des block customisés sous form de plug-in pour GRC mais comme ce n'était pas l'objectif principal du projet nous n'avons pas creusé cette piste. Le fichier généré par GNURadio lors de la compilation d'un flowgraph (.grc) contient du code Python faisant usage de divers modules gnuradio. Nous avons décidé de créer un "squelette" grâce aux flowgraphs puis de modifier le code Python généré. Notre objectif étant de retrouver un écran, nous avons dans un premier temps utilisé ce squelette modifié pour faire des mesures de la quantité de fuites électromagnétiques, la figure 5 illustre les trois cas que nous avons rencontré.



(a) Aucune fuite (b) Fuite d'un écran (c) Autre fuite ("bruit")

Fig. 5: Différents graphes de fuites électromagnétiques

Deux cas se distinguent parmi les trois observés : les cas b et c, qui présentent tous deux un pic de fuite électromagnétique. Cette observation a permis d'affiner notre sujet d'étude : identifier les fuites spécifiques à un écran et les différencier du "bruit" ambiant. Deux approches étaient envisageables, la première est un balayage d'une plage complète de fréquences pour identifier le candidat recherché. La seconde est l'utilisation d'une liste de pixels clocks probables basée sur la popularité des résolutions d'écran. Le calcul de cette liste n'est pas aussi simple qu'il n'y paraît ($hauteur \times largeur \times fps$), car il existe en réalité un "padding" transmis à l'écran qui ajoute de la largeur et de la hauteur à l'écran envoyé, et modifie inévitablement la pixel clock. Ce padding est composé de plusieurs éléments tels d'un

back et front porch, deux parties qui ont d'importants rôles pour la transmission du signal telle que la synchronisation ou les informations de début de ligne, tout ceci est fait de manière transparente pour l'utilisateur. Voici comment se compose une image en réalité en terme d'information transmise par le câble HDMI:

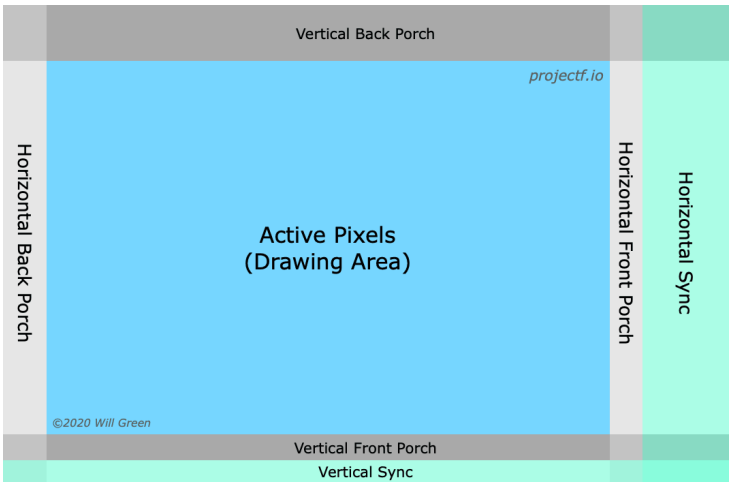


Fig. 6: Informations transmises de plus que l'écran visible (active pixels) lors de la transmission d'une image par un câble HDMI.

Comme nous pouvons le constater, l'écran actif ne représente qu'une partie des informations transmises. Cet aspect jouera un rôle majeur dans notre recherche du pixel clock, puis dans la tâche plus approfondie que nous nous sommes fixée : fournir également les informations relatives aux pixels verticaux et horizontaux, ainsi que le nombre d'images par seconde. Nous détaillerons cet aspect dans la partie dédiée à l'implémentation.

Malgré ces informations, le problème du bruit subsiste. Conscients qu'une seule mesure de fuite ne suffirait pas à éliminer les candidats composés uniquement de bruit, nous avons développé une approche multicritère. Notre implémentation s'appuie sur plusieurs outils : une liste de fréquences candidates, la mesure des niveaux de fuite de ces fréquences, et enfin la corrélation, un élément crucial pour discriminer les écrans du bruit ambiant, que nous allons maintenant détailler. Cette liste de fréquences candidates prend la forme d'une liste de **modes** d'affichage, chacun définit par sa résolution horizontale, verticale et le nombre d'images par seconde. Ce concept est expliqué en détail par la suite dans la partie 3.1 où il devient important.

3 Implémentation

Pour des raisons de clarté et de modularité nous avons décidé de découper le script en plusieurs fichiers .py dont un fichier "maître" qui va utiliser les autres comme des modules, comme décrit dans le schéma de la figure 7. Nous avons utilisé l'approche décrite précédemment uniquement dans les fichiers qui repose sur des modules gnuradio (c'est-à-dire qui interagissent avec la plateforme SDR) : `energy_det.py` et `save_traces.py`. Le fichiers `helpers.py` définit des fonctions et des constantes utiles dans les autres fichiers, et le contenu du dossier `traces/` va être remplacé par les traces enregistrées par `save_traces.py`, qui serviront pour `frame_correlate.py`.

Nous avons donc créé un outil de découverte de paramètres d'écran dans le but d'ensuite effectuer une attaque Tempest sur cet écran. Malgré le fichier transpilé à partir de Javascript (décrit dans la partie 3.1), l'entièreté du code est écrite en Python, avec des tests effectués via l'interpréteur ¨Python 3.11 et

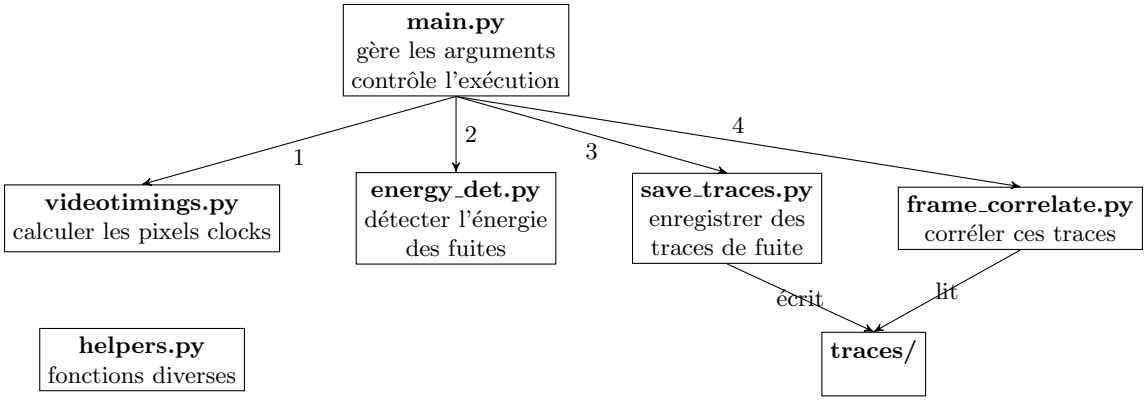


Fig. 7: Flux d'exécution de Calm

GNURadio 3.10.9.2-1 sur ArchLinux et Windows 11. Cependant grâce à l'utilisation d'un langage cross-platform et de l'attention portée pour rendre le code compatible, il ne devrait pas y avoir de problème sous MacOS.

Les dépendances requises pour l'exécution de Calm sont les suivantes :

- une installation fonctionnelle de GNURadio,
- `osmosdr`, `uhd` ou autre selon la SDR utilisée,
- `PyQt5` pour l'affichage graphique lors de la détection d'énergie,
- `numpy` pour effectuer des opérations sur des arrays lors de la corrélation,
- `js2py` pour le bon fonctionnement de `videotimings.py`.

Voici les arguments supportés par notre outil en ligne de commande :

```
usage: main.py [-h] [-v] [--folder FOLDER] [--skip-record] [--sdr SDR] [--step 1-4]
               [--custom-list CUSTOM_LIST]

Automatic resolution and refresh rate finder for TEMPEST

options:
  -h, --help            show this help message and exit
  -v, --verbose
  --folder FOLDER       path to where traces will be saved/searched (default: traces/)
  --skip-record         skip steps 1 and 2 and look in FOLDER for traces to process
  --sdr SDR             choose between "usrp" or "hackrf" (default: hackrf)
  --step 1-4            explore until the desired step is reached (default: 3)
                        1 - energy detection
                        2 - save traces
                        3 - correlation
                        4 - autocorrelation
  --custom-list CUSTOM_LIST
                        path to a file the list of modes to test (default: modes.txt)
```

3.1 Obtenir la pixel clock

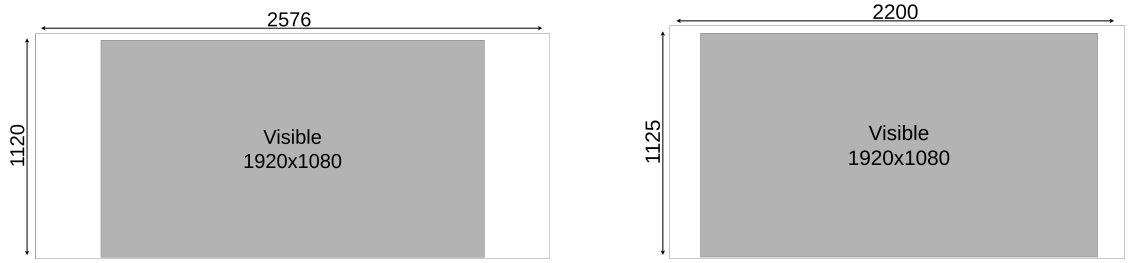
Afin de pouvoir isoler les pixel clocks les plus pertinentes à tester sur un logiciel de raster comme TempestSDR, il est d'abord nécessaire de construire une liste de pixel clocks habituelles. L'idée est ici de trouver un moyen de calculer la (ou les) pixel clock(s) associée(s) à chaque **mode**. On rappelle la définition d'un mode comme `HxV@fps` où :

- H est la résolution horizontale (ou largeur)
- V est la résolution verticale (ou hauteur)
- fps est le nombre d'images par secondes (ou taux de rafraichissement de l'écran)

Cependant comme vu dans la partie précédente, à cause du padding (causé par les front et back porch ainsi que la sync), le calcul de la pixel clock n'est pas simplement $H \times V \times fps$, mais $Htot \times Vtot \times fps$ où $Htot > H$ et $Vtot > V$. De plus la taille du padding n'est pas constante et peut différer d'un écran à l'autre même à mode identique. En effet elle est définie selon le **standard** supporté par l'écran de destination, on compte 5 standards :

1. *CVT (Coordinated Video Timings)* Surtout pour les moniteurs d'ordinateurs et les cartes vidéo, bon équilibre entre la qualité d'image et la compatibilité
2. *CVT-RB (Reduced Blanking)* Version modifiée de CVT avec des intervalles de suppression réduits, augmente la zone d'image active et peut améliorer la qualité d'image, moins compatible que CVT standard
3. *CVT-RBv2 (Reduced Blanking version 2)* Offre une meilleure qualité d'image et une meilleure compatibilité que CVT-RB mais nécessite un support matériel plus récent
4. *CEA-861 (Consumer Electronics Association)* Similaire à CVT, plus largement utilisé dans les appareils électroniques grand public
5. *DMT (Display Monitor Timings)* Offre une qualité d'image comparable à CVT, moins largement pris en charge que CVT

Bien qu'entièrement transparent pour l'utilisateur, l'utilisation d'un standard ou d'un autre va complétement changer la pixel clock de l'écran, c'est à dire la fréquence à laquelle nous devons chercher les fuites du câble. Par exemple pour un écran à $1920 \times 1080@60$, on va avoir une pixel clock de $2576 \times 1120 \times 60 = 173\text{MHz}$ pour le standard CVT et $2200 \times 1125 \times 60 = 148.5\text{MHz}$ pour DMT.



(a) Padding CVT (b) Padding DMT

Fig. 8: Différences entre deux standards

Dans la suite nous allons écrire une association d'un mode à un certain standard de la manière suivante : `HxV@fps#standard`.
Il est donc crucial de prendre en compte tout les standards disponibles pour chaque mode (à savoir que certains standards n'existent pas pour certains modes, par exemple pas de CEA pour $1024 \times 768@60$, ce qui équivaut aussi à dire que $1024 \times 768@60\#cea$ n'existe pas).
Le site Video Timings Calculator[7] permet de calculer la pixel clock pour chaque standard à partir d'un mode fournit par l'utilisateur. De plus l'entièreté des calculs sont fait en Javascript directement par le navigateur, nous avons donc accés au code source complet. Notre intention première était de lire, comprendre et réécrire ce code source de 7000 lignes de Javascript en Python afin de l'incorporer dans notre outil. Par faute de temps et comme ce n'est pas le sujet principal de recherche nous avons rapidement abandonné la réécriture. Enfin il s'est avéré assez difficile de trouver d'autres sources d'informations précises et complètes concernant le calcul du padding selon ces standards.

Nous avons donc choisi d'épurer et d'adapter le code source Javascript à nos besoins pour ensuite le transpiler en Python à l'aide du paquet *js2py*. Voici le processus de transpilation et un exemple d'utilisation:

```
import js2py
js2py.translate_file('videotimings.js', 'videotimings.py')

from videotimings import *
print('["cvt", "cvt_rb", "cvt_rb2", "dmt", "cea"]')
pxclocks, htotals, vtotals = videotimings.compute_pxclock(1920, 1080, 60)
print(pxclocks) #[173000000, 138500000, 133320000, 148500000, 148500000]
print(htotals)  #[2576, 2080, 2000, 2200, 2200]
print(vtotals)  #[1120, 1111, 1111, 1125, 1125]
```

Après vérification les résultats sont identiques à ceux du site ce qui confirme que cette solution est fonctionnelle bien qu'évidemment sous-optimale, car le code Python produit est incompréhensible et que les modification de code source doivent être apportées à *videotimings.js*. Malgré cela nous pouvons désormais utiliser la fonction `compute_pxclock(H, V, fps)` du module `videotimings` pour obtenir dynamiquement la pixel clock, Htot et Vtot pour tout les standards disponibles du mode passé en argument.

Cela permet de créer une fonction lisant une liste de mode depuis un fichier et donnant une liste de mode et standards ainsi que les pixel clocks correpondantes:

```
def parse_mode_and_clock(path):
    outfile = open(path, "r")
    data = outfile.readlines()

    standards = ["cvt", "cvt_rb", "cvt_rb2", "dmt", "cea"]
    mode = []
    clock = []
    for line in data:
        # compute pixel clocks for supported standards
        _, htotals, vtotals = videotimings.compute_pxclock(
            get_x(line), get_y(line), get_fps(line)
        )
        # Pixel clocks computed by the "js" script are rounded
        # Let's compute them again
        pxclocks = []
        fps = get_fps(line)
        for htot, vtot in zip(htotals, vtotals):
            pxclocks.append(htot * vtot * fps)
        # DONE! Now clocks should be more precise
        for pxclock, standard in zip(pxclocks, standards):
            # if existant, append to the list to check
            if pxclock != 0:
                mode.append(line.replace("\n", "") + "#" + standard)
                clock.append(int(pxclock))

    return clock, mode
```

À partir de notre liste de 20 modes, en Annexe A.1, cette fonction nous retourne une liste de 85 modes et standards et autant de pixel clocks, le détail de cette liste est disponible en Annexe A.2.

Il est important de noter que notre fonction recalcule la valeur de la pixel clock, car le code initial l'arrondissait au dixième de MHz près. Bien que cela ne semblait pas spécialement poser de problèmes

nous trouvions plus juste le fait de la recalculer avec l'équation $Htot \times Vtot \times fps$ pour avoir un résultat précis.

Cette fonction est définie dans `helpers.py` utilise donc `videotimings.py` et est appelée au début de l'exécution de `energy_det.py`, il serait judicieux de l'appeler directement dans `main.py` pour coller au mieux à la figure 7.

3.2 Détection du niveau d'énergie des fuites

L'objectif de cette partie de l'exécution est de trouver les fréquences de pixel clock ayant le plus de fuites électromagnétiques. Notre objectif étant de trouver ces dernières, qui représenteraient en théorie un écran, un bon premier tri est de mesurer si une information est transmise, avant de mesurer si c'est celle que nous cherchons. Cette partie se concentre alors sur l'élimination du plus grand nombre de candidats initiaux possible. Avec une liste restreinte de candidats, nous pourrons par la suite faire des calculs plus précis, même si plus coûteux en ressources de calcul et de temps.

Les pixel clocks calculées précédemment se situent pour la majeure partie entre 25 MHz et 150 MHz. Malheureusement, cette gamme de fréquences est utilisée pour d'autres communications comme la radio FM, ce qui va créer un pic de détection électromagnétique à ces fréquences sans pour autant qu'un écran ne s'y trouve. Pour une majeure partie des fréquences de pixel clocks, les détections d'énergie ne peuvent être concluantes et montrer la présence d'un écran du fait de ce que nous appellerons désormais le "bruit" des autres émissions. Ce problème nous a menés à utiliser un des principes des signaux radio : les harmoniques. En effet, le signal numérique du câble étant composé d'une série d'impulsions, les informations seront répétées à chaque multiple de la fréquence de la pixel clock. Ceci est contraire à la plus grande source de bruit, à savoir la radio FM, qui est un signal continu avec une modulation de fréquence et qui, par conséquent, ne se répète pas aux harmoniques. Nous avons utilisé ce principe pour multiplier les fréquences de pixel clock assez de fois pour atteindre une bande de fréquences contenant moins de bruit. Par une phase de test, nous avons choisi empiriquement de multiplier toutes les fréquences jusqu'au-dessus de 500 MHz, où nous avons pu observer qu'il y avait un bruit négligeable. Par conséquent, pour la suite de l'exécution, nous traiterons les fréquences avec un harmonique au-dessus de ce seuil que nous avons choisi.

Une fois les fréquences correctement adaptées, les mesures sont réalisées. Nous avons pris la décision de faire ces différents calculs sur un thread différent de celui créé par ce top block de calcul afin de conserver un affichage graphique des mesures électromagnétiques tout le long de la première phase de ce top block. Le calcul est simple : sur 1 MHz de largeur de bande avec au centre la fréquence de la pixel clock, nous calculons la moyenne de la puissance de ce signal. Cette opération est faite quatre fois à la suite pour chaque pixel clock. Quatre mesures suffisent pour avoir un échantillon représentatif même si, lors d'une mesure, une baisse de niveau d'énergie a eu lieu. Un bruit intempestif, en revanche, ne pose pas de problème, il ajoute simplement un candidat éliminé par les prochains traitements. Pour chacune des fréquences de pixel clock, nous faisons alors ces quatre moyennes, en en faisant aussi une moyenne, et nous sauvegardons sa valeur dans un tableau. Le tableau entièrement constitué comporte alors les moyennes des quatre moyennes de mesures pour toutes les pixel clocks.

Enfin, nous pouvons passer à l'élimination des candidats. Nous avons choisi d'éliminer les candidats en ne choisissant que ceux au-dessus d'un seuil étant une dernière moyenne : celle de tous nos éléments du tableau. Aussi, nous avons fait le choix de réduire encore le nombre de candidats en le calculant ainsi :

$$\text{seuil} = \mu(\textbf{tableau\_de\_mesures}) + \frac{5}{100}\mu(\textbf{tableau\_de\_mesures}) \tag{1}$$

Par ce biais, nous estimons que nous éliminons entre 80% et 90% des candidats fournis par notre liste de modes d'origine. C'est ainsi que pour le mode $1920 \times 1080@60$, nous arrivons à ce résultat :

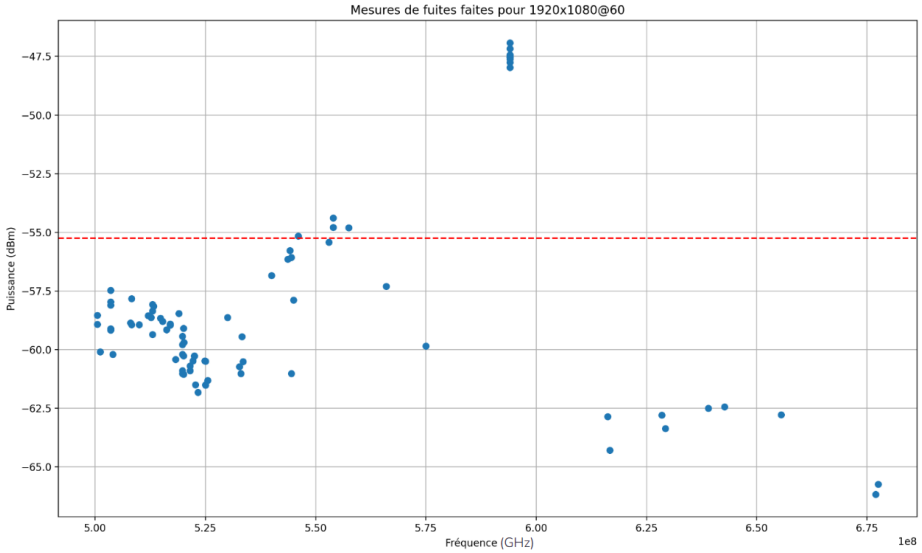


Fig. 9: Mesures de fuites électromagnétiques de toutes les pixel clocks lors d'une transmissions d'un écran en 1920 × 1080@60. Le seuil en rouge indique que les seuls candidats conservés sont ceux au dessus de ce dernier.

Dans notre exemple, nous conservons une douzaine de candidats pour les calculs de corrélation et éliminons par conséquent 73 candidats. En cas d'ajout de fréquences à la liste initiale fournie, la seule raison qui pourrait amener d'autres candidats sélectionnés serait qu'ils soient un multiple de la fréquence mesurée. Nous en avons répertorié une majorité, par conséquent nous estimons à une vingtaine le nombre de candidats maximaux sélectionnés. En fonction de la fréquence, nous estimons en moyenne le nombre de candidats à une dizaine. Une fois les candidats sélectionnés, la partie décisive du traitement peut débuter.

3.3 Enregistrement des traces

Afin de réaliser un calcul de corrélation sur les candidats sélectionnés précedemment nous avons fait le choix d'enregistrer des traces d'une demi seconde pour chacun d'entre eux. Une demi seconde suffit à pouvoir faire un traitement adéquat par la suite sans pour autant saturer la mémoire de traces trop volumineuses. Chaque enregistrement aura une taille d'approximativement 12Mo. Ces enregistrements sont faits grâce au bloc natif GNURadio file sink. Nous créons successivement plusieurs top blocks prenant en argument la fréquence de pixel clock à laquelle l'enregistrement doit avoir lieu ainsi qu'une chaîne de caractères représentant le mode, qui sera utilisée comme nom de fichier. Chaque top block créé un fichier ayant pour nom le mode de la pixel clock, et enregistre une demi seconde, voici l'appel du dit top block depuis la fonction main :

```
def save_traces(sdr, folder_path="traces/"):
  for mode, frq in zip(candidate_modes, candidate_freqs):
    tb = trace_to_file(os.path.join(folder_path, mode + ".trace"), frq, sdr=sdr)
    tb.start()
    time.sleep(0.5)
    tb.stop()
    tb.wait()
```

La figure 10 est le flowgraph GNURadio appelé pour chaque fichier par le code précédent. Il calcule simplement l'amplitude de chaque complexe et l'écrit en binaire dans un fichier.

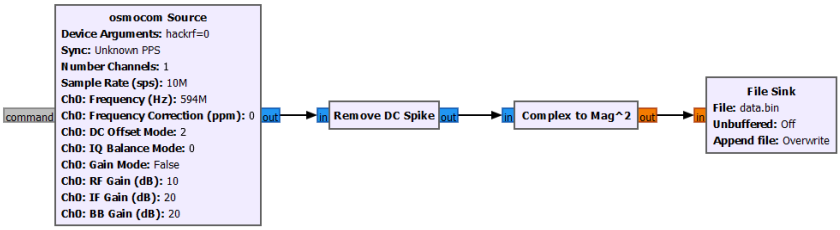


Fig. 10: Flowgraph GNURadio pour enregistrer une trace

Cette solution est la plus simple et intuitive que nous ayons trouvé. La problématique principale est qu'il est impossible dans un même top block de modifier le fichier de destination d'un File Sink, or nous voulions un fichier par enregistrement de fréquence. Il aurait été possible de tout enregistrer dans le même fichier et ensuite savoir dans quelle partie était l'enregistrement de quelle fréquence, mais cette solution est bien moins lisible et modulable que notre première approche. Nous avons donc conservé cette volonté de plusieurs fichiers. Il reste d'autres approches pour faire cette dernière, et ont été explorées. La première était de créer un seul top block que nous modifierons au fil de l'exécution, cependant les fonctions permettant de faire ceci ont été écartées des dernières versions de GNURadio. Ces fonctions sont respectivement lock et unlock. Ou encore il était possible de faire usage de "callbacks" afin de modifier le top block, mais cet utilisation s'est avérée moins aisée et lisible que des créations et suppressions de top blocks successifs. Nous avons conservé alors cette implémentation avec des appels de start et stops successifs. Le nom du fichier dans lequel est enregistrée une trace sera sous cette forme : `HxV@fps#standard.trace`, où H, V, fps et standard sont remplacés par les données correspondantes. Ainsi le nom du fichier contient toute les informations nécessaires au traitement de la trace lors de la prochaine étape. La fin de notre étape étant complétée, plusieurs fichiers enregistrés, la dernière partie de notre traitement peut débuter. De plus à partir de maintenant, la SDR n'est plus requise car nous avons déjà toutes les données nécessaires pour la suite.

3.4 Corrélation sur les images

Dans la partie 3.2, nous avons réussi à isoler une poignée de modes et standards qui sont des candidats potentiels pour être des fuites électromagnétiques issues du flux vidéo envoyé sur un écran. Dans cette partie nous allons essayer de déterminer quelles fuites proviennent bel et bien d'un écran et quelles fuites sont simplement dûes au bruit ambiant ou à une source inattendue d'ondes.
Pour ce faire nous allons exploiter deux particularités du contexte :

- hors cas peu pertinents dans le cadre d'une attaque Tempest, comme les vidéos et les jeux vidéos, le contenu d'un écran d'ordinateur exposant des informations sensibles est presque exclusivement du texte. Lors de l'affichage de texte sur un écran le contenu d'une image à la suivante est quasiment toujours identique, ainsi les fuites résultant du passage de ces images par le câble se ressembleront
- de plus les images sont envoyées à la suite du processeur graphique vers l'écran, il n'y a donc aucune superposition et il est facile d'isoler les fuites correspondants à une image.

Voici par exemple une trace enregistrée à 10000000 de samples par seconde à l'étape précédente d'un écran affichant 60 images par seconde :

Les données d'une image (en prenant en compte le padding) est envoyée en $1/60 \approx 0.0166666$ seconde, ce qui correspond à $10000000 * 0.0166666 = 166666$ samples, on note cette durée de frame `frame_t`,

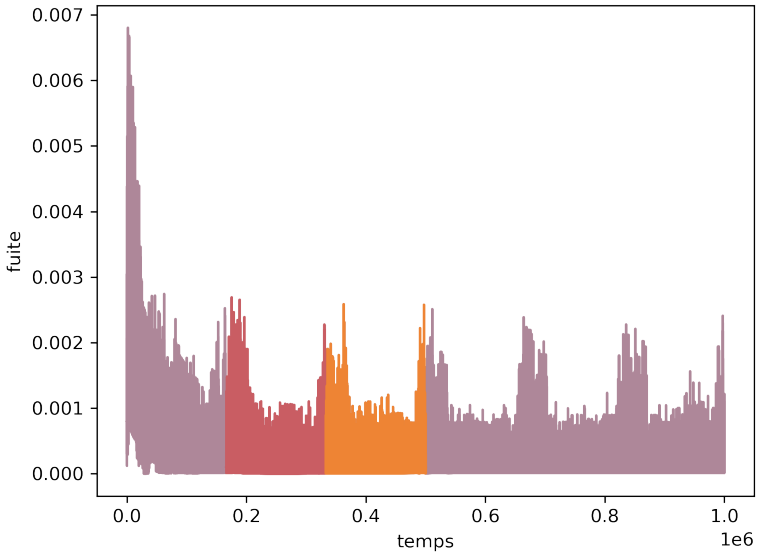


Fig. 11: Visualisation d'une trace de fuites à 60 images par seconde

exprimée en samples. La figure 11 montre bien qu'une répétition a lieu tout les 166666 samples. **Notes** Bien que la pixel clock soit importante pour réaliser les traces à l'étape précédente, on ne s'en sert pas ici puisque qu'elle caractérise déjà le contenu des fuites. On remarque sur la figure 11 un pic au début de la trace, il est causé par la SDR utilisée (HackRF dans ce cas) qui doit se calibrer lors d'un changement de fréquence. Il est donc présent sur chaque trace mais n'est pas dérangeant lors des calculs car nous pouvons l'ignorer en sautant les $5\times$`frame_t` premiers samples enregistrées.

Afin d'avoir une métrique représentant la ressemblance, c'est à dire une corrélation, entre 2 ensembles successifs X et Y de `frame_t` samples chacun, nous avons utilisé le coefficient de Pearson qui se calcule de la manière suivante :

$$r_{XY} = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y}$$

On obtient une valeur de r comprise entre -1 et 1 :

- plus cette valeur est proche de 1, plus les ensembles X et Y sont semblables,
- si elle vaut 0 alors il n'y a aucune corrélation entre ces ensembles,
- si elle se rapproche de -1 alors les ensembles corrèlent de manière opposée.

Nous calculons en tout 5 coefficients de Pearson, si on note f_i la trace résultant de la frame i (on rappelle qu'on saute les 5 premières), on calcule donc :

$$(r_{f_5 f_6}, r_{f_6 f_7}, r_{f_7 f_8}, r_{f_8 f_9}, r_{f_9 f_{10}})$$

Enfin on en calcule le maximum, la moyenne et le minimum. Le maximum n'est pas très indicatif puisqu'on suppose qu'une corrélation pourrait rarement apparaître même dans du bruit. Si le minimum est faible, cela n'exclu pas la possibilité d'un écran, puisqu'il peut être le résultat d'une mouvement brusque sur l'écran, comme un changement de fenêtre. Ainsi on regarde surtout la moyenne qui prend en compte ces différents cas, si les traces correspondent bien aux fuites d'un écran on aura une moyenne relativement proche de 1. De manière empirique nous avons déterminé qu'une moyenne supérieure à 0.1 est intéressante,

surtout à une distance modérée. L'exemple de la figure 11 donne une corrélation de 0.69, ce qui est largement satisfaisant.

Enfin on trie les candidats initiaux par leurs valeur moyenne de corrélation et on affiche les paramètres à tester dans un logiciel de raster. Voici la partie importante du code pour effectuer la corrélation à l'aide du coefficient de Pearson :

```
max_correlations = []
print(f"Correlation over {N_FRAMES} frame(s), skipping {START_FRAME} frame(s)")
for path in paths:
    data = np.fromfile(os.path.join(folder_path, path + ".trace"), dtype=np.float32)
    # time that a whole (inc padding) frame takes in the trace
    frame_t = int(samp_rate / get_fps(path))
    # compute pearson coefficients for each successive pair of frames
    correlations = []
    for i in range(START_FRAME, START_FRAME + N_FRAMES):
        # prevent going out of bounds for the sample
        if frame_t * (i + 2) >= len(data):
            print("error: sample too small/target frame too high!")
            break

        samp1 = data[frame_t * i : frame_t * (i + 1)]
        samp2 = data[frame_t * (i + 1) : frame_t * (i + 2)]
        r = pearson(samp1, samp2)
        correlations.append(r)

    max_correlations.append(max(correlations))
```

La figure 12 montre un exemple de sortie une fois l'étape de corrélation terminée. On y voit les modes et standards avec une corrélation supérieure à 0.1, triés par ordre décroissant, ainsi que les paramètres à rentrer dans un logiciel de raster : la fréquence, Htot, Vtot et le nombre d'images par seconde.

```
STEP 3 - Correlation on successive frames

Correlation over 5 frame(s), skipping 5 frame(s)

Modes to test, in order:
1. 3840x2160@30.0#cea: 0.863     total resolution 4400x2250@30.0        freq 594000000.0
2. 1920x1080@60.0#cea: 0.858     total resolution 2200x1125@60.0        freq 594000000.0
3. 1920x1080@60.0#dmt: 0.855     total resolution 2200x1125@60.0        freq 594000000.0
4. 1920x1080@50.0#cea: 0.768     total resolution 2640x1125@50.0        freq 594000000.0
5. 3840x2160@25.0#cea: 0.729     total resolution 5280x2250@25.0        freq 594000000.0
6. 1920x1080@60.0#cvt: 0.237     total resolution 2576x1120@60.0        freq 519321600.0
```

Fig. 12: Exemple de sortie après la corrélation

Approfondissements Admettons que lors de l'enregistrement, le contenu visible d'un écran `1920x1080@60#cvt` change totalement d'une frame à la suivante, on pourrait supposer que la corrélation sera très faible. Cependant un trace comprend aussi toutes les données de padding, et ces données sont toujours constantes. Si on se réfere à la figure 8a, on peut calculer quelle proportion p de l'écran corrélera toujours d'une frame à l'autre :

$$p = \frac{2576 \times 1120 - 1920 \times 1080}{2576 \times 1120} \approx 0.28$$

Bien sûr, cette valeur peut varier selon le mode et le standard mais cela confirme que même si l'écran affiche un contenu qui change à chaque frame, notre programme n'éliminera par la fréquence candidate.

Enfin dans le cas réel, nous n'avons aucune certitude que la trace des fuites d'une frame commence vraiment au début d'une frame, cela est même très peu probable, avec une chance sur 166666 pour écran à 60 fps et un enregistrement à 10000000 de samples par seconde. La figure 13 montre avec un exemple réaliste que cette trace de fuite tombe au milieu d'une frame et va aller jusqu'au milieu de la suivante.

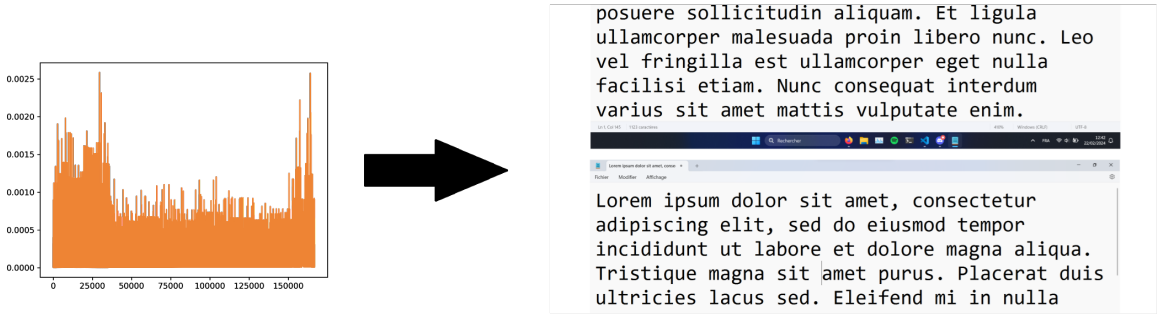


Fig. 13: Exemple réaliste de la représentation d'une fuite (avec padding)

Mais il est apparaît assez rapidement que ceci n'est pas un problème pour la corrélation car tant que `frame_t` est constant nous allons quand même calculer le coefficient de Pearson sur des traces successives relativement identique : il n'est pas nécesssaire d'identifier le début ou la fin d'une frame pour calculer la corrélation.

Le calcul de la corrélation tel que présenté précédemment est quasiment instantané (jamais plus d'une seconde même avec 20 traces) car nous utilisons les arrays du paquet Python Numpy, dont l'implémentation est compilée à partir de code C. Pour pousser l'exploitation plus loin et peut être obtenir de meilleurs résultats avec des distances plus élevées il est envisageable de calculer le coefficient de Pearson sur un plus grand nombre de frames.

4 Problèmes rencontrés et futur travaux

Durant la réalisation de ce projet, nous avons rencontré divers obstacles qui ont ralenti notre progression, mais que nous avons finalement identifiés et surmontés.

Le premier problème auquel nous avons été confrontés concernait l'installation des logiciels de reconstruction vidéo. Le premier, gr-tempest, ne fonctionnait que sur une version antérieure de GNU-Radio et nécessitait une distribution Linux spécifique. L'installation de TempestSDR s'est avérée plus simple sur Windows grâce à une version portable contenant tous les pilotes requis.

De plus, nous avons été confrontés à des perturbations externes lors de l'utilisation de notre outil, qui, dans un certain contexte, nous a fourni des résultats totalement incohérents. Nous avons réalisé que cela provenait du fait que l'antenne du SDR était trop proche de l'alimentation de l'ordinateur et faussait complètement les données mesurées.

Comme indiqué dans la section "Enregistrement des traces", il était impossible d'enregistrer des traces à des fréquences différentes dans la même instance (appelée top block). La solution que nous avons adoptée consiste à créer et à exécuter des top blocks distincts, chacun ayant comme argument la fréquence à laquelle nous devons enregistrer les traces.

Nous avons des pistes d'amélioration pour notre outil et pour aller plus loin dans l'attaque. En plus d'utiliser la corrélation image par image, nous pourrions implémenter la corrélation ligne par ligne. Cette approche nous permettrait de confirmer avec une précision accrue que les données comparées dans la corrélation proviennent effectivement d'un écran et non d'une source de bruit parasite. La corrélation ligne par ligne est plus précise que la corrélation image par image car, dans le cas d'une image, deux lignes de pixels consécutives seront très similaires. Comme le montre la figure 14, les données à afficher à l'écran sont envoyées ligne par ligne. Il suffirait donc de calculer la corrélation comme précédemment en divisant `frame_t` par Vtot pour obtenir le temps requis pour envoyer une ligne, ce qui correspond au nombre de samples associé à cette ligne. Cependant, l'utilité d'implémenter cette méthode reste à discuter car les résultats actuels sont déjà plutôt concluants.

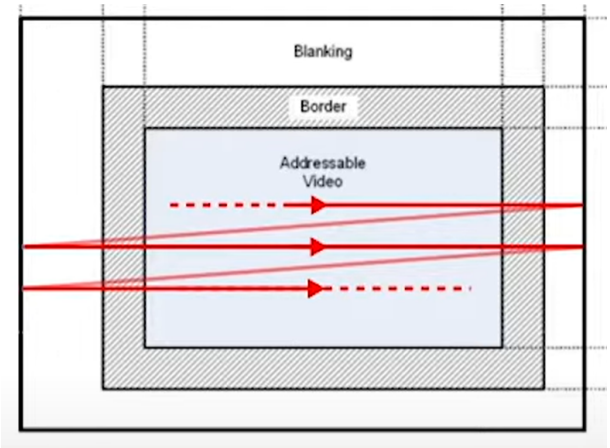


Fig. 14: Affichage d'un écran ligne par ligne

L'amélioration de la distance à lequel notre outil fonctionne, pour cela nous pensons que l'utilisation d'antennes directive, plus précise couplé à un amplificateur à faible bruit pourrait grandement améliorer la distance. Par exemple, une équipe de chercheurs [6] en 2020 ont réalisés avec succès une attaque TEMPEST à une distance de 80 mètres.

On a observé que la pixel clock de certains modes était les mêmes, par exemple la px_clock de `3840x2160@30` est égale à 2 fois la px_clock de `1920x1080@60` pour certains standards comme le CEA-861:

$$4400 \times 2250 \times 30 = 297000000 = 2 \times (2200 \times 1125 \times 60)$$

Ainsi chaque harmonique paire de `1920x1080@60#cea` sera aussi une harmonique de `3840x2160@30#cea`. Donc à une telle harmonique il y aura une corrélation similaire pour ces deux candidats. Il semble faisable de déterminer quel candidat est le bon en prenant celui qui a le plus petite taille d'écran: en effet une corrélation sur `1920\times 1080` impliquera une corrélation sur `3840\times 2160`, mais une corrélation sur `3840\times 2160` n'implique pas une corrélation sur `1920\times 1080`.

5 Conclusion

Rappelant notre objectif de rendre l'attaque TEMPEST sur les écrans plus accessible et, si possible, automatisée, nous avons surmonté les obstacles initiaux et expérimenté avec TempestSDR et gr-tempest. Cette démarche nous a permis d'identifier les paramètres clés pour la reconstruction du contenu d'un écran, ainsi que la précision requise pour chacun d'eux : d'abord le pixel clock (le plus important), suivi par Htot, Vtot et fps. Cette avancée représente une étape importante dans le développement d'une attaque TEMPEST efficace contre les écrans d'ordinateur.

Des recherches approfondies ont permis d'exploiter les données électromagnétiques pour éliminer les pixels clocks les moins probables. Deux approches ont été explorées : la détection de l'énergie des fuites aux harmoniques dans une bande à faible bruit, et la quantification de la périodicité de ces fuites par rapport au taux de rafraîchissement de l'écran. Les tests effectués démontrent que Calm identifie systématiquement les modes supportés. L'ajout de nouveaux modes à tester s'effectue simplement en modifiant la liste préexistante. La création de plusieurs listes permet un contrôle fin sur le niveau d'exploitation de l'outil.

Cependant, l'identification du mode correct n'est pas toujours immédiate, ce qui limite l'automatisation complète du processus jusqu'à l'ouverture du logiciel de rasterisation. De plus, l'exécution de TempestSDR avec des paramètres passés en ligne de commande semble impossible sans modification du code source. Deux alternatives existent : le développement d'un outil de reconstruction d'image à partir de zéro ou l'utilisation d'un outil en cours de développement comme celui de Pierre Granier [3].

Bibliographie

1. Otan sdip-27, `https://www.ia.nato.int/niapc/tempest/certification-scheme`
2. van Eck, W.: Electromagnetic radiation from video display units: An eavesdropping risk? Elsevier Science Publishers B.V. (North-Holland) (1985)
3. Granier, P.: Cybel tp étudiant (2024), `https://gitlab.istic.univ-rennes1.fr/pigranier/CYBEL_TP_etudiant`
4. Larroca, F., Bertrand, P., Carrau, F., Severi, V.: gr-tempest: an open-source gnu radio implementation of tempest. In: 2022 Asian Hardware Oriented Security and Trust Symposium (AsianHOST). pp. 1–6. IEEE (2022)
5. Marinov, M.: Tempestsdr (2014), `https://github.com/martinmarinov/TempestSDR`
6. Pieterjan De Meulemeester, Bart Scheers, G.A.V.: Eavesdropping a (ultra-)high-definition video display from an 80 meter distance under realistic circumstances. 2020 IEEE International Symposium on Electromagnetic Compatibility & Signal/Power Integrity (EMCSI) (2020)
7. Verbeure, T.: Video timings calculator (2019), `https://tomverbeure.github.io/video_timings_calculator`

A Annexes

A.1 Liste arbitraire des modes les plus courants

Cette liste a été contruite depuis nos observations tout au long du projet et avec l'utilitaire en ligne de commande linux **xrandr**. Ce dernier permet de connaître les modes d'affichage supportés par l'écran auquel l'ordinateur hôte est connecté.

1920x1080@60.0
1920x1080@50.0
1920x1080@59.94
1280x720@60.0
1280x720@50.0
1280x720@59.94
1280x1024@60.02
1024x768@60.0
800x600@60.32
720x576@50.0
720x480@60.0
720x480@59.94
640x480@60.0
640x480@59.94
3840x2160@30.0
3840x2160@25.0
3840x2160@24.0
3840x2160@29.97
3840x2160@23.98
2400x1350@59.93

A.2 Liste des modes les plus courants et leurs standards

Voici la liste précédente complétée avec tout les standards supportés pour chaque mode.

1920x1080@60.0#cvt
1920x1080@60.0#cvt_rb
1920x1080@60.0#cvt_rb2
1920x1080@60.0#dmt
1920x1080@60.0#cea
1920x1080@50.0#cvt
1920x1080@50.0#cvt_rb
1920x1080@50.0#cvt_rb2
1920x1080@50.0#cea
1920x1080@59.94#cvt
1920x1080@59.94#cvt_rb
1920x1080@59.94#cvt_rb2
1920x1080@59.94#dmt
1920x1080@59.94#cea
1280x720@60.0#cvt
1280x720@60.0#cvt_rb
1280x720@60.0#cvt_rb2
1280x720@60.0#dmt
1280x720@60.0#cea
1280x720@50.0#cvt
1280x720@50.0#cvt_rb
1280x720@50.0#cvt_rb2
1280x720@50.0#cea
1280x720@59.94#cvt
1280x720@59.94#cvt_rb
1280x720@59.94#cvt_rb2
1280x720@59.94#dmt
1280x720@59.94#cea
1280x1024@60.02#cvt
1280x1024@60.02#cvt_rb
1280x1024@60.02#cvt_rb2
1280x1024@60.02#dmt
1024x768@60.0#cvt
1024x768@60.0#cvt_rb
1024x768@60.0#cvt_rb2
1024x768@60.0#dmt
800x600@60.32#cvt
800x600@60.32#cvt_rb
800x600@60.32#cvt_rb2
800x600@60.32#dmt
720x576@50.0#cvt
720x576@50.0#cvt_rb
720x576@50.0#cvt_rb2
720x576@50.0#cea
720x480@60.0#cvt

720x480@60.0#cvt_rb
720x480@60.0#cvt_rb2
720x480@60.0#cea
720x480@59.94#cvt
720x480@59.94#cvt_rb
720x480@59.94#cvt_rb2
720x480@59.94#cea
640x480@60.0#cvt
640x480@60.0#cvt_rb
640x480@60.0#cvt_rb2
640x480@60.0#dmt
640x480@60.0#cea
640x480@59.94#cvt
640x480@59.94#cvt_rb
640x480@59.94#cvt_rb2
640x480@59.94#dmt
640x480@59.94#cea
3840x2160@30.0#cvt
3840x2160@30.0#cvt_rb
3840x2160@30.0#cvt_rb2
3840x2160@30.0#cea
3840x2160@25.0#cvt
3840x2160@25.0#cvt_rb
3840x2160@25.0#cvt_rb2
3840x2160@25.0#cea
3840x2160@24.0#cvt
3840x2160@24.0#cvt_rb
3840x2160@24.0#cvt_rb2
3840x2160@24.0#cea
3840x2160@29.97#cvt
3840x2160@29.97#cvt_rb
3840x2160@29.97#cvt_rb2
3840x2160@29.97#cea
3840x2160@23.98#cvt
3840x2160@23.98#cvt_rb
3840x2160@23.98#cvt_rb2
3840x2160@23.98#cea
2400x1350@59.93#cvt
2400x1350@59.93#cvt_rb
2400x1350@59.93#cvt_rb2