



E.T.S.I. INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Diseño de una herramienta para monitorización y test de intrusión en el
protocolo CAN bus
Design of a tool to audit and pentest the CAN bus protocol

Realizado por
Jesús Javier Quirante Pérez
Tutorizado por
David Santo Orcero
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Fecha

Fecha defensa:
El Secretario del Tribunal

Agradecimientos

El éxito no siempre tiene que ver con el talento. Se trata de constancia. El trabajo duro y constante conduce al éxito. Ha sido un camino largo, lleno de obstáculos que hacían muy difícil ver el final, pero todo llega para aquellos que siguen luchando por lo que quieren. Tengo mucho que agradecer a las personas que han hecho todo lo posible para que pueda sacar la mejor versión de mí haciendo que, con este proyecto, pueda cerrar una de las etapas de mi vida.

En primer lugar, quiero agradecer a mi familia por darme la oportunidad durante estos años de poder seguir en esto y de confiar en mí desde el principio. Además quiero agradecer a mi pareja por estar conmigo a pie de cañón para que todo salga bien, has sido y eres un pilar fundamental para poder llegar a esto.

También quiero agradecer a mis compañeros y amigos de la facultad. Han sido muchas las horas que hemos estado estudiando para un examen, una práctica o ayudándonos entre nosotros desinteresadamente para poder llegar al final de esto. No pensábamos que llegaría pero sí, existe un final y se puede llegar a ser ingeniero informático.

Finalmente, quiero agradecer a mi tutor y amigo David Santo Orcero, tanto por sus enseñanzas como por su labor en este proyecto. Elegimos un tema para este proyecto que me ha hecho disfrutar y aprender muchísimo durante estos meses.

Gracias a todos, los que están y los que ya no están, por haber formado parte de este proceso y haberme aportado tanto para llegar a ser ingeniero informático.

Resumen:

El protocolo CAN Bus utilizado en los automóviles es el que hace posible que todos los componentes electrónicos funcionen correctamente. Este protocolo no es del todo seguro, por lo que se ha diseñado una herramienta capaz de realizar ciertas funciones dentro de la red, tales como la monitorización y/o almacenaje del tráfico que circula en el vehículo en tiempo real, así como la modificación e inyección de ese tráfico en la red.

Para esto, primero es necesario entender como funciona el protocolo CAN Bus, para qué se usa, qué tipos de tramas circulan por la red y por qué se utiliza. Una vez sepamos como funciona y queramos usar la herramienta para modificar el comportamiento del coche, es importante hacer un análisis de las vulnerabilidades y los tipos de ataques que se han realizado para ello.

Palabras claves: Tramas, CAN Bus, seguridad, coche, ECU

Abstract:

The CAN bus protocol used in automobiles allows all electronic components to function properly. This protocol is not completely secure, so a tool has been developed to perform certain functions within the network, such as monitoring and/or storing the traffic circulating in the vehicle in real time, as well as modifying and injecting this traffic into the network.

The first step is to understand how the CAN bus protocol works, what it is used for, what types of frames circulate on the network and why they are used. Once we know how it works and we want to use the tool to modify the behaviour of the car, it is important to carry out an analysis of the vulnerabilities and the types of attacks that have been carried out to do this.

Keywords: frames, CAN Bus, security, car, ECU

Índice de contenidos

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Estructura de la memoria	3
2. Estado del arte	5
2.1. Introducción	5
2.2. Seguridad en CAN Bus	12
2.2.1. Ataques de acceso físico	14
2.2.2. Ataques de acceso remoto	15
2.2.3. Posibles soluciones	17
2.3. Can-utils	24
2.4. SavvyCAN	25
2.5. Kayak	26
2.6. PCAN-View	27
3. Herramientas empleadas	29
3.1. Python	29
3.2. Python curses	31
3.3. Instrument Cluster Simulator - ICSim	32
3.3.1. Configuración del entorno	33
3.4. SocketCAN	35
3.5. SQLite	42
4. Análisis	45
4.1. Ingeniería de requisitos	45
4.1.1. Requisitos funcionales	45
4.1.2. Requisitos no funcionales	47
4.1.3. Casos de uso	48
5. Diseño e implementación	49
5.1. Monitorización y almacenamiento	49
5.2. Modificación	52
5.3. Inyección	53
6. Conclusiones y líneas futuras	55
6.1. Conclusiones	55
6.2. Futuras líneas de trabajo	56

6.2.1. Mejora visual	56
6.2.2. Opción para mostrar diferencias	56
6.2.3. Compatibilidad con Windows	57
Bibliografía	59
Apéndice A. Manual de usuario	63
A.1. Monitorización	64
A.2. Modificar	68
A.2.1. Modificar archivo	68
A.2.2. Modificar trama	69
A.3. Inyectar	70
A.3.1. Inyectar archivo	70
A.3.2. Inyectar trama	71

CAPÍTULO 1

Introducción

1.1. Motivación

La ciberseguridad ha cobrado una especial importancia en las últimas décadas y en concreto en los últimos años. Asimismo, con el desarrollo de la industria del automóvil y en concreto el auge de los coches eléctricos, la necesidad de crear entornos seguros se ha hecho más presente que nunca.

Uno de los protocolos de comunicación más famosos que utilizan los automóviles, que fueron fabricados aproximadamente a partir del año 2000, es el protocolo CAN Bus. Este protocolo utiliza paquetes para comunicarse con los componentes electrónicos de un coche y realizar cada una de sus funcionalidades, pero, ¿cómo de seguro es este protocolo? ¿Es posible interceptar y leer los datos de esos paquetes? ¿Es posible crear o modificar un paquete y distribuirlo dentro de la red del automóvil para controlar sus funcionalidades? Estas y muchas más preguntas surgieron a raíz de pensar en el funcionamiento de este protocolo y en cómo penetrar en el sistema del vehículo para tener acceso a toda la información.

Para hacer un análisis de esto, surge la necesidad de crear una herramienta que sea capaz de monitorizar toda la información recibida del protocolo CAN Bus, capaz de procesar los datos para interpretarlos, modificarlos o inyectarlos.

1.2. Objetivos

El objetivo principal de este proyecto ha sido hacer una herramienta que sea capaz de interactuar con el protocolo CAN Bus, es decir, se han desarrollado diferentes funciones para tratar con los paquetes recibidos de un automóvil.

Una de las funciones principales que permite esta herramienta era la monitorización del tráfico de la red CAN Bus. La aplicación es capaz de recibir y mostrar en tiempo real los datos de los paquetes que están circulando por la red. El objetivo de esta función era poder monitorizar todo lo que está pasando y ver mucha información sobre las tramas. Además, otro de los objetivos de esta función consistía en poder almacenar todos esos datos en una base de datos para, posteriormente, analizar los datos. Además, la función de monitorizar tenía como objetivo poder aplicar un filtro para mostrar y/o almacenar únicamente las tramas que el usuario desea.

Otro de los objetivos de este proyecto ha sido hacer una función que permita al usuario modificar el contenido de las tramas que están almacenadas en un fichero. El objetivo de esta función era poder cambiar el campo de los datos de una trama por otro elegido por el usuario.

La función modificar carece de sentido si no viene acompañada de otra función para inyectar el contenido modificado en la red, la cual ha sido otro de los objetivos de la herramienta. La función inyectar es capaz de enviar de vuelta a la red una o varias tramas, con el objetivo de modificar el comportamiento en la red según los datos introducidos por el usuario. Es decir, el usuario podrá crear paquetes personalizados para enviarlos a la red, engañando al sistema y haciendo que el vehículo ejecute acciones pensando que provienen de él mismo. Por ejemplo, si el usuario detecta el tipo de trama que hace que el coche arranque, podría enviarla al sistema y provocar esa acción sin necesidad de arrancar manualmente el coche, ya que el sistema no verifica de donde viene el paquete.

Al final, el objetivo del proyecto ha sido hacer una herramienta que permita realizar todas estas funciones de manera eficaz, con una interfaz agradable para el usuario y funcional.

1.3. Estructura de la memoria

La memoria está estructurada de la siguiente forma:

1. **Introducción:** informa sobre los objetivos y la motivación del proyecto.
2. **Estado del arte:** se realiza un análisis del estado del arte.
3. **Herramientas empleadas:** se hace un análisis sobre las herramientas empleadas y la configuración del entorno.
4. **Análisis:** se realiza un análisis de la herramienta desarrollada.
5. **Conclusiones y líneas futuras:** informa sobre el futuro de la herramienta y añade una conclusión.

Estado del arte

2.1. Introducción

La red **CAN Bus** (*Controller Area Network*) hace posible la comunicación entre los componentes electrónicos (ECUs) del automóvil. El funcionamiento es similar al del cuerpo humano donde todos los elementos electrónicos están interconectados entre sí vía CAN Bus y la información de cada componente se comparte por la red al resto de nodos.

The diagram illustrates the intricate wiring of a modern vehicle, highlighting the integration of various electronic systems. A legend in the top right corner identifies the four main communication protocols used:

- CAN bus** (Blue line)
- LIN bus** (Yellow line)
- ASRB bus** (Red line)
- FlexRay bus** (Green line)

Key components and their connections are labeled throughout the vehicle's chassis:

- Front End:** Headlight, Turn Signal, Engine, Brake, Millimeter-Wave Radar, Front Sensor, Wiper, Air conditioner-Main.
- Interior/Chassis:** Pedestrians Protecting Unit, Airbag, Curtain Airbag, EPS, Power Window, Dashboard, Side Airbag, Rear Sensor, Wiper, Lear Light, Suspension, Brake, Star Coupler, Seat, Pretensioner, Seatbelt, SBW-Center ECU, Door-Main, Door Mirror, Accelerator, Passenger Seat Sensor, Pressure Sensor, Body Computer (BCM), Door-Main.
- Wheels/Chassis:** Rain Sensor, Brake, Accelerator, Door-Main.

The diagram demonstrates how these diverse systems are interconnected through a multi-protocol bus architecture, ensuring efficient data exchange and coordinated vehicle operation.

Dentro del automóvil existe un conector para poder acceder a la red, normalmente situado bajo el volante del conductor, llamado conector OBD-II o también conocido como

diagnostic link connector (DLC). Este conector cuenta con 16 pines, cada uno con diferentes funciones o protocolos ya que existen multitud de ellos. Cada fabricante decide cuales encajan mejor para su vehículo, por lo que la configuración de algunos pines puede ser diferente entre modelos. Uno de los protocolos más famosos y más usado es el protocolo CAN Bus, comúnmente situado en los pines 6 y 14 del vehículo. Otros protocolos que también encontramos en este conector son:

- **Protocolo SAE J1850.** Este protocolo de bus es más lento que el protocolo CAN pero es mucho más barato de implementar. Este protocolo se utiliza para la comunicación de datos entre diferentes módulos del automóvil.
- **Protocolo PWM.** Protocolo utilizado en los sistemas de control de motores y luces, a través de pulsos eléctricos. Este protocolo es principalmente usado por *Ford* en los pines 2 y 10.
- **Protocolo VPW.** El protocolo trabaja a un voltaje de 7V y tiene algunas diferencias con CAN a la hora de interpretar los datos, ya que la señal es dependiente del tiempo.
- **Protocolo Keyword y ISO 9141-2.** También conocido como KWP2000, usa el pin 7 y, opcionalmente el 15, es común encontrarlo en los vehículos europeos fabricados después del año 2003. Al contrario que CAN, los paquetes de este protocolo tienen un nodo transmisor y un nodo destino.
- **Protocolo *Local Interconnect Network (LIN)*.** Es el protocolo más barato de los protocolos de vehículos y fue diseñado para complementar el protocolo CAN. No tiene código de prioridad por lo que existe un nodo maestro para realizar todas las transmisiones de la red.
- **Protocolo MOST.** *Media Oriented Systems Transport (MOST)* está diseñado para conectar los dispositivos multimedia, soportando hasta un máximo de 64 dispositivos MOST en topología anillo o estrella.

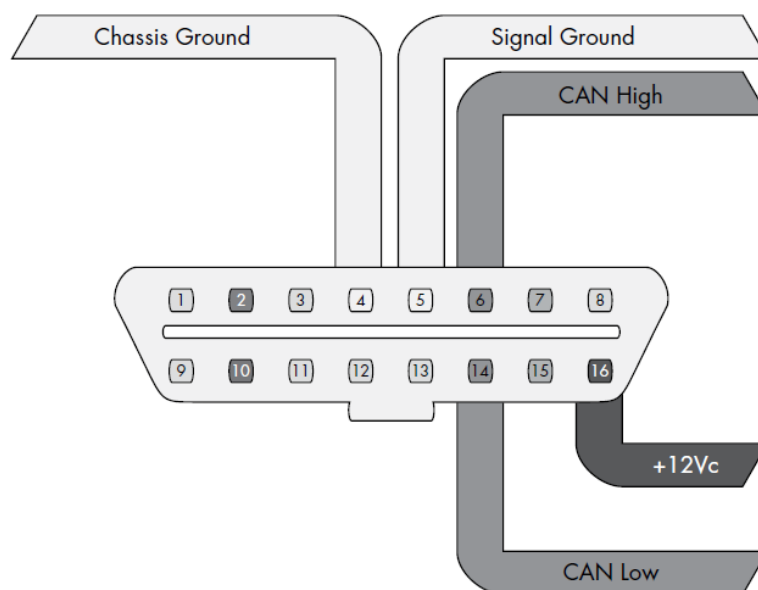


Figura 2.2: Pines del conector OBD-II. Fuente: Libro

El protocolo CAN Bus trabaja a 2.5V sin señal, añadiendo o restando 1V al recibir una señal. La red tiene dos conexiones diferentes *CAN High* y *CAN Low*, como se muestra en la Figura 2.2. Los paquetes que tienen información crítica sobre el vehículo, como el frenado, se transmiten a través del bus de alta velocidad (**CAN High**) mientras que los paquetes con información con menor relevancia, como el control de las puertas, ocurre en el bus de menor velocidad (**CAN Low**).

Los paquetes que viajan sobre CAN Bus no están estandarizados, es decir, el paquete que abre las puertas de un coche no es el mismo para todos los modelos, cada uno representa los datos de manera distinta. No obstante, podemos identificar la estructura de una trama de datos ya que, en el protocolo CAN Bus, existen dos tipos: estándar y extendido. Los paquetes extendidos son similares a los estándar, solo que utilizan un mayor espacio para almacenar los identificadores.

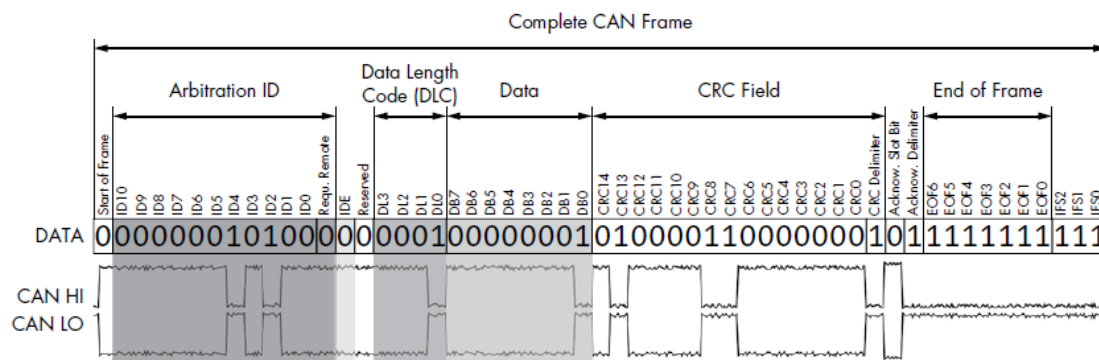


Figura 2.3: Formato estándar de los paquetes CAN. Fuente: Libro

Los paquetes estándar, representados en la Figura 2.3, contienen cuatro elementos clave:

- **Arbitration ID.** Estos bits identifican el dispositivo que se está comunicando, aunque cualquier dispositivo podría enviar múltiples *Arbitration IDs*. Si se envían al mismo tiempo dos paquetes CAN, el que tenga menor identificador tendrá prioridad.
- **Identifier extension (IDE).** Este bit siempre está a 0 en los paquetes estándar o a 1 para los paquetes extendidos.
- **Data length code (DLC).** Este campo indica el tamaño de datos, que puede variar entre 0 y 8 bytes.
- **Data.** Contiene los datos del paquete, pudiendo tener un tamaño máximo de 8 bytes.

Otros de los elementos que también encontramos en la trama son:

- **Start of frame.** Es el primer bit de la trama y demarca el comienzo de la transmisión.
- **Petición de transmisión remota (RTR).** Es el último bit dentro de *Arbitration ID* e indica una petición de transmisión remota tomando el valor dominante (0) para tramas de datos y recesivo (1) para peticiones de tramas remotas.
- **Reserved.** Bit reservado, puede tener el valor 0 o 1.

- **CRC.** Tiene una longitud de 15 bits, verifica que los datos fueron transmitidos correctamente. El último bit de este campo corresponde con el delimitador CRC que debe tener el valor 1.
- **Hueco de acuse de recibo - ACK.** Bit que tomará el valor recesivo (1) cuando el transmisor emite y el receptor emite dominante (0). El siguiente bit a continuación de este corresponde con el delimitador ACK que debe tomar el valor 1.
- **Fin de la trama EOF.** Este campo indica el final de la trama, con una longitud de 7 bits que deben tomar el valor 1.

Sin embargo, en el formato extendido encontramos algunas diferencias ya que el identificador pasa a tener 29 bits. El formato de la trama es el siguiente:

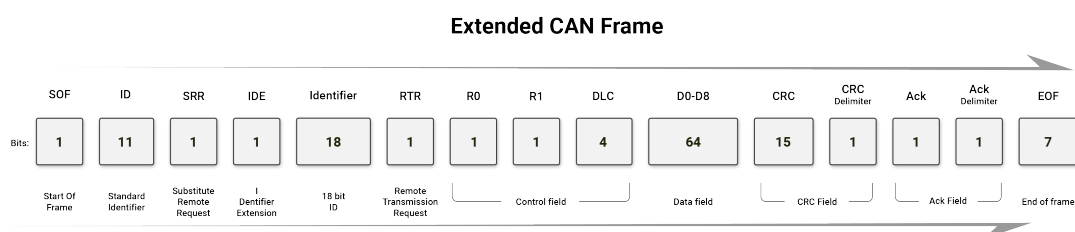


Figura 2.4: Formato extendido de la trama CAN. Fuente autopi.io

- **Start of frame.** Es el primer bit de la trama y demarca el comienzo de la transmisión.
- **Identificador A (ID_A).** Corresponde a la primera parte del identificador, con un tamaño de 11 bits, donde se representa la prioridad de la trama.
- **Sustituto de transmisión remota (SRR).** Este bit siempre toma el valor 1.
- **Identifier extension (IDE).** Este bit siempre está a 1 en los paquetes extendidos o a 0 para los paquetes estándar, ya que indica si el identificador contiene 29 bits.
- **Identificador B (ID_B).** Corresponde a la segunda parte del identificador donde también representa la prioridad de la trama. El tamaño es de 18 bits.
- **Petición de transmisión remota (RTR).** Es el último bit dentro de *Arbitration ID* e indica una petición de transmisión remota tomando el valor dominante (0) para tramas de datos y recesivo (1) para peticiones de tramas remotas.
- **Bits reservados (r1, r0).** En este caso contamos con dos bits reservados que debe de ser dominante (0) pero debe de aceptar tanto dominante como recesivo (1).
- **Data length code (DLC).** Este campo indica el tamaño de datos, que puede variar entre 0 y 8 bytes.
- **Data.** Contiene los datos del paquete, pudiendo tener un tamaño máximo de 8 bytes.
- **CRC.** Tiene una longitud de 15 bits, verifica que los datos fueron transmitidos correctamente.
- **Delimitador CRC.** Debe tomar el valor 1.

- **Hueco de acuse de recibo - ACK.** Bit que tomará el valor recesivo (1) cuando el transmisor emite y el receptor emite dominante (0).
- **Delimitador ACK.** Debe tomar el valor 1.
- **Fin de la trama EOF.** Este campo indica el final de la trama, con una longitud de 7 bits que deben tomar el valor 1.

Existen cuatro tipos de tramas, las tramas de datos (*data frames*) que acabamos de definir, las tramas remotas (*remote frames*), tramas de error (*remote frames*) y las tramas de sobrecarga (*overload frame*).

Usualmente en las **tramas remotas** la información se transmite en pequeños paquetes llamados tramas. Sin embargo, a veces un dispositivo necesita información específica de otro dispositivo. En este caso, el primer dispositivo puede enviar una solicitud de trama remota pidiendo la información que necesita. El dispositivo que tiene la información envía una trama en respuesta a la solicitud (RTR = 1; recesivo). Las tramas remotas o de petición de transmisión remota se diferencian de las tramas de datos en que las tramas remotas no tienen campo de datos.

Las **tramas de error** son un tipo especial que no cumple con el formato de las tramas CAN. Se transmiten cuando un nodo detecta un mensaje erróneo, provocando que el resto de nodos transmitan a su vez una trama de error. Para evitar que un nodo bloquee el bus con mensajes de error existe un mecanismo de contadores.

La **trama de sobrecarga** se parece a la trama de error, ya que viola el formato estándar de las tramas CAN. Se envía cuando un nodo está muy ocupado y no puede enviar más tramas, por lo que el bus de comunicación debe proporcionar un retardo adicional entre tramas para evitar la pérdida de datos.

En el protocolo CAN, las tramas de datos y remotas están separadas por al menos tres bits recesivos (1). Si se detecta un bit dominante (0) después de estos tres bits recesivos, se considera como el inicio de una nueva trama. Las tramas de error y sobrecarga no respetan este espacio entre tramas.

Para garantizar la sincronización en CAN, se utiliza un método llamado "bit stuffing."° relleno de bits. Este método consiste en insertar un bit de polaridad opuesta después de cinco bits consecutivos de la misma polaridad. Los bits insertados son eliminados por el receptor.

En una trama CAN, todos los campos se rellenan con bits de relleno, excepto el delimitador CRC, el acuse de recibo ACK y el fin de la trama. Si un nodo detecta seis bits consecutivos iguales en un campo que debe ser rellenado, se considera un error y se emite un error activo, que consiste en seis bits consecutivos dominantes y viola la regla de relleno de bits. Esta regla puede hacer que una trama sea más larga de lo esperado si se suman los bits teóricos de cada campo de la trama.

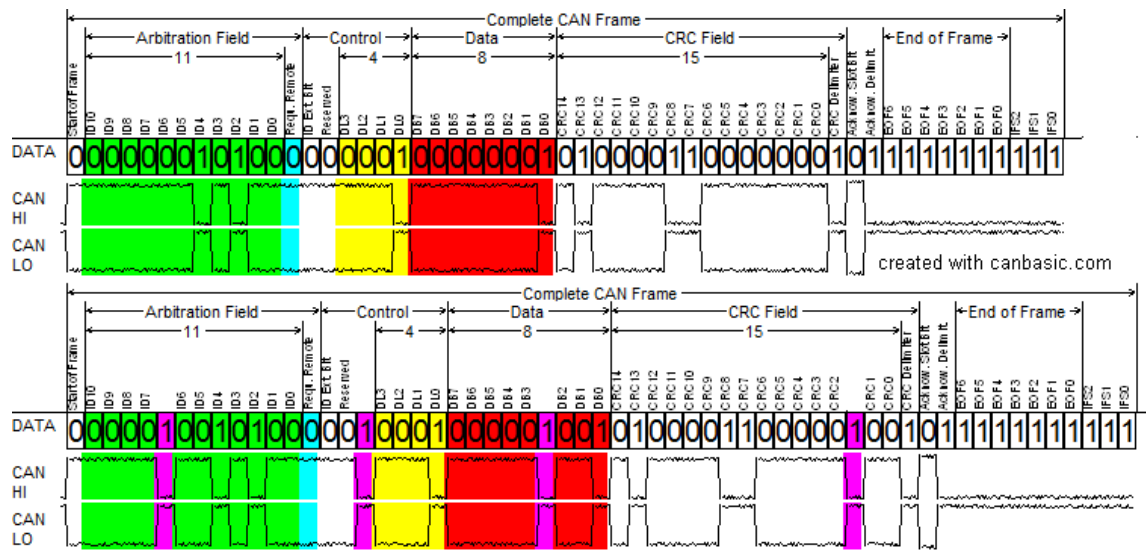


Figura 2.5: Trama CAN antes y después de la adición de bits de relleno (en morado). Fuente: Wikipedia

En las redes de bus CAN, los paquetes se transmiten en modo broadcast, lo que significa que todos los controladores en la red reciben cada paquete, al igual que ocurre con el protocolo UDP en redes Ethernet. Los paquetes no contienen información sobre qué controlador envió qué información, lo que hace difícil identificar a los dispositivos responsables de la transmisión de los datos. Además, dado que cualquier dispositivo en el bus puede ver y transmitir paquetes, resulta sencillo para cualquier dispositivo simular ser cualquier otro dispositivo, lo que puede dar lugar a problemas de seguridad en la red.

El protocolo CAN Bus fue desarrollado por la compañía alemana Bosch en la década de los 80 ante la necesidad de un protocolo robusto y confiable para el sistema de control de los vehículos. Hasta ese momento los sistemas de comunicación en la industria del automóvil tenían grandes carencias de eficiencia, velocidad, seguridad y no cumplían muchos de los requisitos indispensables necesarios para la transmisión de datos en tiempo real.

Como se ha mencionado anteriormente, este protocolo usa una topología de bus para transmitir los mensajes entre las diferentes ECUs. Esto ofrece una gran cantidad de ventajas:

- Inmunidad a las interferencias. Al utilizar dos cables con señales opuestas evita las posibles interferencias, junto con el mecanismo de prioridad de mensajes y la detección y corrección de errores.
- Reducción del número de cables en la red. Esto es una gran ventaja ya que reduce el número de averías, simplifica la implementación del sistema y reduce los costes. Figura 2.6.
- Eficiencia en la transmisión de mensajes. Esto implica mayor facilidad para transmitir y gestionar la prioridad de los mensajes en la red, según su importancia, así como la obtención de los datos como la velocidad del vehículo, el nivel del depósito, las revoluciones del motor, etc.
- Velocidad de transmisión. Los mensajes en las redes CAN pueden llegar a transmitirse a una velocidad de hasta 1Mbps.

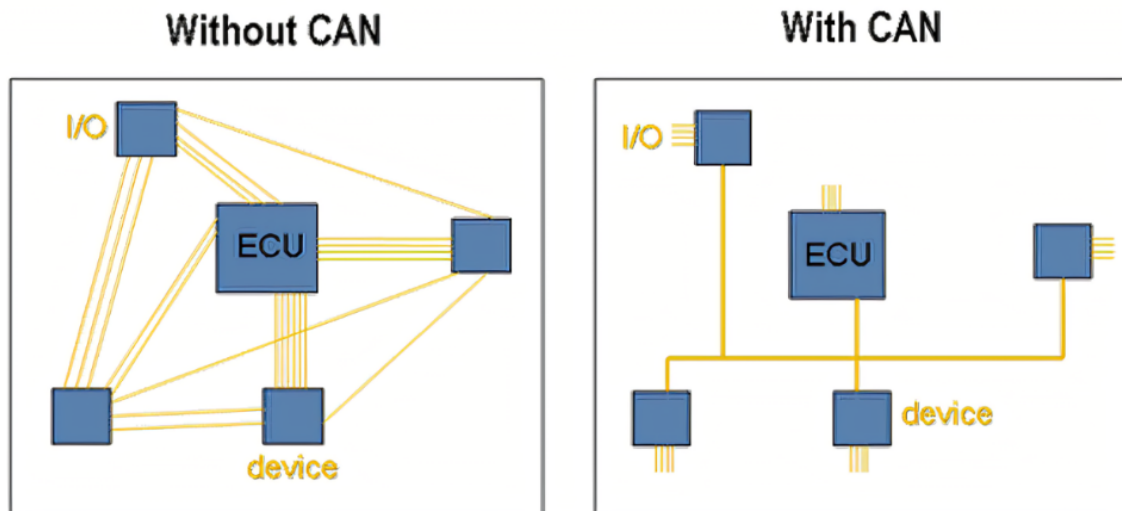


Figura 2.6: Ejemplo de las conexiones necesarias al utilizar o no el protocolo CAN Bus. Fuente: blog.reparacion-vehiculos.es

Sin embargo, el protocolo CAN Bus también presenta algunas desventajas:

- Vulnerabilidad a ataques. Como ya hemos hablado previamente, es relativamente sencillo interceptar los paquetes de la red ya que no cuenta con ningún tipo de autenticación lo que también implica poder ver el contenido de estos ya que los paquetes no están encriptados. Además, también es posible inyectar paquetes dentro de la red ya que lo único que identifica el origen es el identificador, por lo que un atacante podría enviar paquetes modificados y se ejecutarían como si fuesen emitidos por una ECU.
- Si el nodo central falla y deja de funcionar provocará fallos en la conexión con todos los dispositivos de la red.
- Debido al mecanismo de prioridad de los mensajes, es posible que algunos de ellos tarden mucho o no sean enviados si existe una alta demanda de mensajes de alta prioridad.

El protocolo CAN Bus es utilizado principalmente en la automoción (tanto en coches como en camiones, autobuses u otro tipo de vehículos) para establecer una comunicación entre los distintos dispositivos electrónicos, como se ha explicado anteriormente. Sin embargo, este protocolo tiene otras aplicaciones donde también es comúnmente utilizado, por ejemplo:

- Maquinaria industrial. Esto facilita la comunicación entre los dispositivos de la maquinaria o los equipos en la industria, incluyendo maquinaria agrícola, maquinaria de construcción, equipos de minería, maquinaria de fabricación, sistemas de automatización, etc.
- Aviación. Se emplea para conectar los sistemas y sensores individuales de las aeronaves, como una alternativa a los cables múltiples convencionales.
- Industria marina. El protocolo CAN bus es empleado para el control de la navegación, la monitorización del sistema y la comunicación del resto de dispositivos electrónicos.

- Industria médica. En este ámbito es empleado para conectar diferentes dispositivos y equipos que requieren una comunicación rápida y fiable, como por ejemplo en el quirófano, la radiología o la ecografía.

2.2. Seguridad en CAN Bus

Las características de seguridad integradas ya existentes en el protocolo CAN Bus están principalmente diseñadas para garantizar una comunicación confiable y no para una red segura. Como resultado, es inevitable la existencia de ciberataques a esta red, atentando contra la seguridad del vehículo y de los pasajeros, por ejemplo la manipulación de las ECUs. Eventualmente, puede afectar a la reputación del fabricante de automóviles con implicaciones financieras sustanciales como retiros del mercado.

Una de las características más alarmantes es la ausencia de encriptación en CAN, teniendo una gran influencia en la privacidad de los datos individuales. Como se ha comentado anteriormente, por diseño, CAN es una red de difusión que permite a los nodos capturar mensajes que pasan por la red. Dado que los datos se transmiten sin cifrar, el adversario puede obtener los datos deseados. Esto puede conducir a una invasión de la privacidad, especialmente porque los automóviles modernos tienen la capacidad de recopilar información personal del conductor.

Según una encuesta de la industria de 2019, la seguridad y la protección son las principales preocupaciones a corto y mediano plazo para la industria automotriz. Por lo tanto, se ha llevado a cabo una amplia investigación para encontrar posibles soluciones a las vulnerabilidades CAN. Algunos de estos estudios han sido testigos de ataques experimentales exitosos contra automóviles y vehículos pesados. Al mismo tiempo, los investigadores también han propuesto formas de prevenir estos conocidos ataques. Esto incluye la segmentación de la red, el cifrado, la autenticación y los sistemas de detección de intrusos (IDS).

Es imprescindible contar con una evaluación de las vulnerabilidades de la red para identificar los problemas de seguridad de la misma. Por lo tanto, el análisis de vulnerabilidades del protocolo CAN Bus se puede realizar en base a la confidencialidad, integridad y disponibilidad. La confidencialidad consiste en proporcionar los datos únicamente a personas autorizadas. Estos datos que recibe el receptor deben de ser exactamente los mismos que el emisor envió, sin ninguna modificación (integridad). Disponibilidad significa que el sistema está disponible para los usuarios autorizados en cualquier momento. Sin embargo, el protocolo CAN no tiene métodos criptográficos integrados para garantizar la confidencialidad. Esto permite a los atacantes acceder a datos confidenciales de los usuarios y comprometer su privacidad. Además, CAN Bus tiene un CRC para verificar la integridad de los errores de transmisión, pero no evita que los piratas informáticos ingresen datos y violen su integridad. El protocolo no tiene comprobaciones de integridad exhaustivas y no mantiene la integridad. Debido a la naturaleza de la mensajería basada en prioridades, cuando se reenvía/inserta un mensaje con la prioridad más alta, los nodos con prioridades más bajas pierden el acceso a la red y su disponibilidad se ve comprometida. El protocolo CAN Bus no cumple con tres criterios básicos de seguridad, esto indica claramente que el protocolo no tiene protección contra ataques.

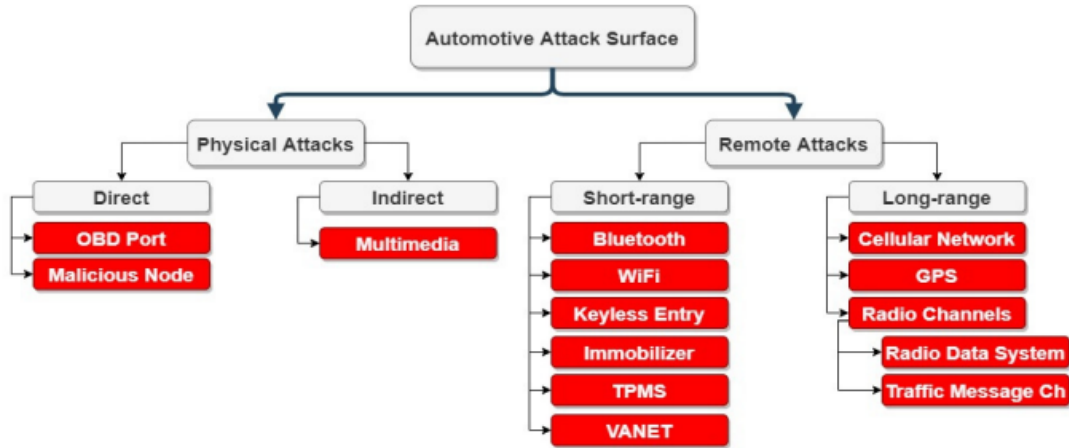


Figura 2.7: Vectores de ataque en los automóviles.

A lo largo de la historia se han realizado una gran variedad de ataques, cada vez existen nuevos vectores de ataque ya que los automóviles tienen muchas más tecnologías (Fig 2.7). En la década de 1950, la electrónica automotriz tenía un costo de un 1 % del total del automóvil, mientras que en la actualidad es del 35 % y se espera un aumento de hasta el 50 % en 2030. Ante cualquiera de estos ataques, el protocolo CAN Bus se muestra vulnerable. En la tabla 2.1 se muestra un resumen de los ataques más conocidos.

DDoS	Modificación	Tipo de acceso	Notas
Si	No	OBD-II	No requiere mensajes CAN completos
No	Si	OBD-II, CD, Bluetooth, GSM	Ataques experimentales sistemáticos. Acceso indirecto a través del sistema central del vehículo
No	Si	OBD-II indirecto	Ataque por medio de una aplicación móvil
Si	Si	Múltiples entradas remotas	Análisis de ataques remotos realizados a 21 coches comerciales
No	Si	Wi-fi, 2G, OBD-II	Acceso a la red CAN a través de un exploit de navegador
Si	No	OBD-II, ECU Comprometida	Exploits de la capa de enlace de datos de SAE J1939
No	Si	Wi-fi, 2G	Ataques ransomware a través del aire
No	Si	TPMS	Envío remoto de datos TPMS (sistema de monitorización de la presión de los neumáticos) falsos

Tabla 2.1: Resumen de los ataques al protocolo CAN Bus. La modificación incluye ataques de repetición, suplantación de identidad y de información falsa.

El primero de los ataques que se realizó fue en 2007 por Hoppe y Dittman en el elevador eléctrico. Este tipo de ataques se clasifica como ataque de acceso físico, ya que requieren de acceder físicamente al vehículo por parte del atacante. También existe otro tipo de ataques donde no es necesario un acceso físico si no que se realizan a través de interfaces de comunicación inalámbrica, llamados ataques de acceso remoto. La mayoría de los ataques que se han realizado a lo largo del tiempo han sido ataques de acceso físico, aunque algunos expertos mencionan que el acceso físico a la red CAN Bus no es práctico, por lo que es más interesante centrarse en los ataques de acceso remoto.

2.2.1. Ataques de acceso físico

Un ataque de acceso físico requieren acceso directo o indirecto a red CAN Bus. Se puede acceder directamente a través del puerto de diagnóstico a bordo (OBD) o un nodo malicioso. Los puertos OBD son una superficie de ataque importante ya que se puede acceder a todos los nodos, incluso si se utiliza la segmentación de red. Uno de los ejemplos fue cuando koscher et al. (uno de los primeros en realizar prácticas de ataque al protocolo CAN Bus) manipuló y controló varios módulos del CAN, incluyendo algunos de gran importancia como el control de los frenos y el control del motor, a través del puerto OBD-II. De esta forma fue capaz de controlar el freno para inutilizarlo mientras que el vehículo estaba en movimiento, pudiendo manipular el cuadro de instrumentos con datos falsos, parámetros del motor e incluso la desactivación. Por otra parte, si existe un nodo malicioso en la red CAN es capaz de escuchar y transmitir mensajes para interrumpir la red. Utilizando el puerto de diagnóstico OBD es posible replicar ataques gracias al nodo malicioso. Palanca et al. lanzó un ataque de denegación de servicio (DDoS) dirigido contra un Alfa Romeo Giulietta 2012 sin modificar. Este estudio demostró que cualquier persona con acceso físico a una red podría romperla con herramientas simples. Este ataque no requiere una transmisión completa del mensaje, en cambio, sobrescribe los bits recesivos y genera un error de transmisión. La contribución de este estudio fue que explotó una vulnerabilidad en el estándar CAN donde, después de esta investigación, el gobierno de Estados Unidos declaró una alerta (ICS-ALERT-17-209-01). Murvay y Groza llevaron a cabo un análisis de investigación similar para mostrar las limitaciones del ataque en diferentes tasas de bits y para violar los métodos de autenticación. Mukherjee et al. implementaron ataques DDoS en el estándar SAE J1939, utilizado en automóviles pesados. Realizaron tres ataques DDoS por separado:

1. Envío de múltiples mensajes de solicitud para un *Program Group Numer* (PGN) compatible con el fin de sobrecargar la ECU objetivo. PGN trata de un concepto que forma parte del estándar SAE J1939, que define cómo se comunican los dispositivos electrónicos a través del bus CAN en vehículos pesados. . Un PGN es un número de 18 bits que identifica de forma única un grupo de parámetros dentro del estándar J1939.
2. Enviar una solicitud de envío falsa manipulada (RTS) provocando un desbordamiento de buffer en el objetivo. Las tramas *Request To Send* (RTS) se utilizan en las comunicaciones para indicar que un dispositivo está listo para transmitir datos.
3. Mantener abiertas las conexiones a través de mensajes *Clear to Send* (CTS) y ocupar toda la red. Los mensajes CTS se utilizan en las comunicaciones para indicar que un dispositivo está listo para recibir datos.

Este trabajo fue uno de los primeros estudios que utilizó la especificación SAE J1939. Murvay y Groza implementaron ataques de suplantación de identidad y DDoS en SAE J1939. Estos trabajos demostraron que SAE J1939 es vulnerable a ataques específicos de protocolo además de todos los ataques en CAN Bus.

Existe la posibilidad de que un ataque de acceso físico sea indirecto. Estos ataques requieren la inserción de un objeto físico en el vehículo, pero no requieren que el atacante tenga acceso directo a la red. Chekovey et al. desarrollaron un modelo de ataque de acceso indirecto que piratea el sistema de TI de un taller de reparación de automóviles para acceder a CAN a través de un ordenador. Los modelos de ataque también incluyeron ataques a través de dispositivos multimedia (reproductores de CD, USB o MP3). Hoppe et al implementaron un ataque de disco multimedia. Aunque el ataque no obtuvo acceso al CAN, pudo asustar al conductor tras mostrar una advertencia en la pantalla y reproducir una alarma.

2.2.2. Ataques de acceso remoto

Los vehículos modernos de hoy en día incluyen muchos tipos de interfaces inalámbricas necesarias para comunicarse con sistemas, como el antirrobo pasivo, los sistemas de control de presión de neumáticos (TPMS), Bluetooth, datos inalámbricos, etc. Estas interfaces inalámbricas generalmente necesitan comunicarse con CAN a través de una ECU puerta de enlace para proteger la red. Sin embargo, los estudios han demostrado que las ECU de puerta de enlace han sido pirateadas para acceder al CAN Bus de forma aislada.

Checkoway y otros han logrado comprometer el TPMS, Bluetooth, el canal FM y la red de un vehículo mediante ingeniería inversa, afirmando que un ciberdelincuente podría secuestrar automóviles a través de los mensajes CAN, siendo posible abrir las puertas del coche y facilitando el robo del mismo. Wu et al., propusieron un ataque remoto a través de una aplicación maliciosa de autodiagnóstico. Si alguien usa una aplicación maliciosa para monitorear/diagnosticar la condición del vehículo, el atacante controlará el vehículo de forma remota y llevará a cabo un ataque a distancia.

Valasek y Miller realizaron un estudio de ataques remotos en 12 marcas de automóviles y 21 vehículos comerciales, identificando la superficie de ataque remoto y la dificultad de entrar en cada vehículo. El ataque fue en tres etapas. La primera de ellas consistía en comprometer una ECU que formase parte de una interfaz inalámbrica. El segundo paso fue inyectar mensajes para comunicarse con la ECU vulnerable. El paso final fue modificar la ECU para que funcionase maliciosamente. Los investigadores creían que aumentar la cantidad de sistemas ciberfísicos en los automóviles aumentaría sus vulnerabilidades, pero debido a la amplia gama de aplicaciones en los automóviles, no pudieron probar esto en la práctica. También hackearon de forma remota un Jeep Cherokee y desactivaron el motor en 2014. Después de este ataque, se hizo una declaración pública de que el automóvil era vulnerable a un ataque remoto. En 2016, Savage y su equipo controlaron los frenos y los limpiaparabrisas del Chevrolet Corvette con una unidad de control telemático comercial. Este ataque indica que las vulnerabilidades de CAN pueden ser pirateadas por hardware del mercado y los fabricantes no pueden abordar el problema por completo. En 2016, Nie et al. implementó un ataque remoto a un Tesla Model S mediante interfaces inalámbricas y móviles. El laboratorio de seguridad de Tencent Keen ha identificado varias superficies de ataque para vehículos BMW, lo que demuestra que incluso los vehículos de alta gama disponibles comercialmente pueden ser atacados y vulnerados por ciberataques.

Otro método de ataque inalámbrico son las actualizaciones de software por aire (OTA). OTA es una solución rentable y escalable para que los fabricantes entreguen actualizaciones de software de forma remota. Sin embargo, este es otro vector de ataque que los ciberdelincuentes pueden usar para ganar acceso en la red de comunicaciones del automóvil. Beek y Samani implementaron un ataque de ransomware mediante una actualización *Over The Air* (OTA). Los vectores de ataque remotos de los automóviles modernos tienen cada vez más importancia que los ataques físicos y, a medida que aumenta la cantidad de conexiones en un automóvil, la cantidad de ataques inalámbricos también aumenta cada día. En un futuro cercano, los vehículos pueden estar equipados con comunicaciones de vehículo a vehículo (V2V) y de vehículo a infraestructura (V2I) para crear redes ad-hoc vehiculares (VANET). VANET es un tipo de red de comunicación en la que los vehículos se utilizan como nodos de red. Cuando se reduce el alcance del canal de comunicación (hasta 1km), esta se establece de forma esporádica (ad-hoc). Por este motivo, estas redes se consideran un tipo especial de red móvil. Sin embargo, VANET está diseñado para optimizar el tráfico y evitar colisiones. Para brindar estos beneficios, las VANET usan sensores automotrices y son inalámbricas. Se pueden recibir o transmitir mensajes falsos en la red VANET, lo que puede interferir con la red de comunicación del vehículo.

Como se ha observado, en la obtención de datos en una red CAN Bus no solo vulnera la seguridad del vehículo si no que también puede invadir la privacidad del conductor, ya que los vehículos modernos recopilan información de los datos personales. Una investigación reveló que era posible obtener el historial de ubicación preciso del automóvil y otros datos personales (registro de llamadas telefónicas, lista de contactos, direcciones de correo electrónico y fotos) del teléfono conectado. De esta forma, un atacante puede obtener toda esta información estando en escucha en CAN Bus.

2.2.3. Posibles soluciones

El protocolo CAN Bus es claramente vulnerable, como ya hemos visto, existen multitud de vectores de ataque tanto físicos como remotos que vulneran este protocolo haciéndolo inseguro. Por lo tanto, requiere de mecanismos de defensa para tener una conducción segura. Las investigaciones que se han realizado al respecto para solucionar este problema han concluido en cuatro mecanismos de defensa: segmentación de red, encriptación, autenticación y detección de intrusos. A continuación se muestra un resumen de las ventajas y desventajas de implementar estos mecanismos de seguridad, en la tabla 2.2.

Método propuesto	Ventajas	Desventajas
Segmentación de red	Límite de acceso al usuario final.	Costo incrementado, dificultad en el mantenimiento.
Cifrado	Confidencialidad en la transmisión de datos.	Incrementa el costo computacional, incrementa el tráfico en la red, cifrado débil debido al tamaño de trama.
Autenticación	Transmisión de datos más segura.	Incremento del coste computacional, incremento del tráfico en la red.
Detección de intrusos	Detección de anomalías y ataques en la red.	Diseño complejo del algoritmo, no puede garantizar la seguridad en la red.

Tabla 2.2: Resumen de las soluciones de seguridad propuestas.

La segmentación de red consiste en dividir la red en subredes más pequeñas, de esta forma permite controlar el tráfico de red en cada segmento, mejorando la seguridad y el rendimiento ya que reduce la superficie de ataque y evita el desplazamiento lateral de los atacantes. En el caso de las redes CAN, la interconexión entre subredes se controla a través de una ECU de puerta de enlace. Además, es un mecanismo de protección sencillo de implementar en la red CAN, pero no es efectivo si la ECU de puerta de enlace es comprometida por el atacante. Para encontrar una solución a este problema, Kammerer y otros propusieron un enrutador de acoplamiento en estrella con algunas configuraciones de seguridad, ignorando la seguridad dentro de la subred, siendo posible realizar un ataque de repetición en una subred y a atacar a otras subredes ignorando la verificación de seguridad del enrutador. TU München et al. propusieron una arquitectura de bus de servicio automotriz diseñada como una arquitectura de dos niveles para evitar ataques externos, separando el sistema de información y entretenimiento y todas las funciones esenciales entre sí. Todos los componentes pueden enviar y recibir mensajes pero, no pueden enviar datos porque cualquiera puede escribir en el bus de servicio del automóvil a través de la ECU central.

Por lo tanto, la segmentación de red mejora la seguridad pero no es suficiente para proteger el CAN Bus. Además, dificulta el mantenimiento del sistema y aumenta los costos.

El protocolo CAN Bus utiliza una red de comunicación sin ningún mecanismo criptográfico integrado, lo que permite q aun atacante escuchar y comprender todas las co-

municaciones entre nodos de la red. Para evitar dicha exfiltración de datos, se debe implementar un sistema de cifrado ligero. Aunque existen métodos comerciales de encriptación basados en software (Trillium, CANcrypt) y los fabricantes están implementando métodos de encriptación patentados en los vehículos, hay informes que afirman que los mecanismos de encriptación en los automóviles disponibles comercialmente pueden romperse.

Uno de los problemas que presenta CAN a la hora de implementar un cifrado seguro es la limitación del campo de datos. Aunque este problema podría solucionarse enviando varias tramas CAN para un solo mensaje resolviendo el problema en redes de poco tráfico, no es una solución para la gran cantidad de tráfico que existe actualmente en los automóviles. Otro inconveniente al que se enfrenta el cifrado es el poder computacional limitado de las ECUs. Teniendo en cuenta la vida útil del automóvil, es posible descifrar una clave de cifrado estática, por lo que se requiere un intercambio dinámico de claves. Sin embargo, esto es más difícil de implementar y requiere de más recursos computacionales. Las claves dinámicas también pueden causar latencia en las ECUs con recursos limitados y son inaceptables para los sistemas en tiempo real. Doan y Ganesan implementaron el cifrado de hardware AES-128 en chips FPGA para sistemas CAN. La implementación de hardware de este método reduce la latencia y aumenta el rendimiento. Sin embargo, este método modifica la ECU y no es compatible con versiones anteriores. Otro estudio utilizó funciones físicas no clonables (PUF), es decir, entidades que aprovechan las características únicas de un material físico para generar un valor único. PUF se basa en las características inherentes de los componentes electrónicos, como cambios en las características eléctricas de los transistores o fluctuaciones en las señales analógicas. Estas propiedades se utilizan para generar respuestas únicas que no se pueden reproducir incluso si se creara un dispositivo idéntico. De esta forma se puede adquirir la clave privada a partir de las características físicas de la ECU. Este método resuelve el problema de generación de claves criptográficas, pero también requiere la modificación de la ECU.

El cifrado protege contra ataques y proporciona privacidad, pero esto no es suficiente para proteger el CAN Bus. Incluso los mecanismos criptográficos irrompibles no pueden evitar los ataques de repetición. A continuación se muestra la tabla 2.3 los sistemas de cifrado propuestos.

Método de cifrado	Tráfico de red	Clave
AES-128 y SHA-1	Aumenta	Simétrica estática
XOR	Sin cambio	Sincronizada dinámicamente
XOR	Sin cambio	Simétrica estática
Diffie-Hellman de curva elíptica y AES-256	Aumenta	Simétrica
<i>Tiny Encryption Algorithm</i>	Aumenta	Simétrica estática
Triple DES	Aumenta	Sincronizada dinámicamente

Tabla 2.3: Resumen de las soluciones de seguridad propuestas.

Según el diseño de CAN, no es posible identificar el remitente de un mensaje. Si un atacante tiene acceso a la red, puede enviar mensajes maliciosos y todos los nodos tomarán ese mensaje como legítimo. Una de las soluciones propuestas a este problema es la autenticación VeCure, basada en grupos de confianza donde los grupos de alta confianza reparten una clave simétrica y tiene una velocidad de procesamiento aceptable. No obstante, se envía un mensaje de autenticación después de cada mensaje transmitido en la red, lo que duplicaría el tráfico de red. Además, si algún nodo de la red está comprometido, no puede proteger el sistema ya que un nodo autorizado sería el que envía los mensajes. Otra de las opciones es LiBrA-CAN, propuesto por Groza y otros, cuya función es dividir las claves de autenticación entre grupos de múltiples nodos para mejorar la eficiencia. Aunque este método es bastante exitoso, requiere un gran ancho de banda y no es compatible con el CAN tradicional. Nowdehi y otros identificaron cinco criterios para la implementación comercial de métodos de autenticación: rentabilidad, compatibilidad con versiones anteriores, reparación de vehículos y soporte de mantenimiento, detalles de implementación suficientes y gastos generales razonables. Analizaron diez métodos de autenticación y, naturalmente, ningún método pudo pasar los cinco criterios. También existen productos comerciales que brindan autenticación basada en hardware, como la familia S32K de NXP. La familia S32K tiene un motor de servicio criptográfico (CSE) con código de autenticación de mensajes basado en criptografía (CMAC) que proporciona autenticación segura y es un sistema basado en hardware que acelera significativamente el proceso. Por ejemplo, la autenticación de clave pública se puede lograr en menos de 100 microsegundos con aceleración de hardware, mientras que la autenticación de software tarda 10 milisegundos o más, según el tamaño de la clave. Sin embargo, la industria está actualmente preocupada por el coste de las ECUs. Es posible ver mas métodos basados en hardware para proteger el CAN Bus gracias a la mejora de la tecnología hardware.

La última de las propuestas para la mejora de la seguridad en CAN Bus es el sistema de detección de intrusos (IDS). Una de las grandes ventajas que presenta es que generalmente no necesita hacer una modificación al controlador CAN actual y, además, el tráfico de red no aumenta. Los métodos de detección de intrusos se pueden dividir en detección basada en firmas (uso indebido) y detección basada en anomalías. Las comprobaciones de detección basada en firmas se encargan de comprobar ataques conocidos en las bases de datos, por lo que requieren de actualizaciones periódicas para estar actualizadas por los nuevos ataques. Tiene bastante éxito en la detección de ataques conocidos, pero no detecta los desconocidos. El sistema de detección de intrusos basado en anomalías analiza el comportamiento de la red y reconoce las desviaciones del comportamiento normal, aunque la precisión suele ser inferior al sistema de detección basado en firmas. A diferencia de este, el IDS basado en anomalías puede detectar ataques desconocidos.

Hay varios parámetros que el sistema IDS puede evaluar a través del CAN Bus. Muther et al. definen ocho sensores de detección de anomalías para identificarlas de forma estructurada. En la siguiente tabla 2.4 se muestran los 8 sensores de detección de anomalías. Todos estos sensores de detección están inspirados en el comportamiento común de CAN Bus. Las desviaciones del comportamiento normal de estos sensores son indicativos de intrusión, y varias soluciones IDS utilizan estos sensores para la detección de intrusiones. Estas soluciones se pueden clasificar como basadas en tiempo/frecuencia, basadas en características del sistema físico, basadas en especificaciones y basadas en características.

Sensor	Descripción
Formalidad	Corregir el tamaño del mensaje, la cabecera, el tamaño del campo, delimitadores, etc.
Ubicación	El mensaje está permitido respecto al sistema de bus dedicado.
Rango	Cumplimiento de la carga útil en términos de rango de datos.
Frecuencia	El comportamiento temporal de los mensajes es el correcto.
Correlación	La correlación de mensajes en diferentes sistemas de bus se adhiere a la especificación.
Protocolo	El orden correcto, la hora de inicio, etc. de los protocolos internos de desafío-respuesta.
Plausibilidad	El contenido de la carga útil del mensaje es plausible, no hay una correlación inviable con los valores anteriores.
Consistencia	Los datos de fuentes redundantes son consistentes.

Tabla 2.4: Sensores de detección de anomalías

En los automóviles la mayoría de las ECUs transmiten señales periódicas, es decir, cualquier cambio en la frecuencia de estas señales se podría identificar como una anomalía. Esta anomalía es altamente probable de que sea un ataque. El IDS basado en el intervalo de tiempo/frecuencia propuesto por Lee et al., analiza el tiempo de respuesta de la trama remota transmitida. En este caso, el ataque y el tipo de ataque se pueden detectar con un algoritmo simple y eficiente, pero este método aumenta el tráfico del bus al inyectar tramas remotas para el análisis. El análisis de tiempo/frecuencia proporciona información útil sobre el CAN, sin embargo, las condiciones del vehículo (por ejemplo inactivo, conduciendo) y los esquemas de prioridad de CAN Bus pueden alterar significativamente la información del tiempo y afectar los resultados de IDS basados en tiempo/frecuencia. Este método tampoco puede detectar ataques cuya frecuencia no cambia.

No obstante, se puede hacer uso de las características físicas de la red CAN para detectar ataques, ya que cada transceptor tiene un tipo de señal distinto aunque transmitan los mismos datos. Esto puede producirse debido al envejecimiento, cableado o a variaciones aleatorias producidas por la fabricación. Choi y otros propusieron VoltageIDS usando las características eléctricas únicas de las señales CAN como huellas dactilares. Las distintas localizaciones de las ECUs con sus variaciones en la longitud del cable resultan en una resistencia diferente, cambiando las características de la señal. Analizaron ocho características de la señal, como valores de pendiente positiva y negativa y valores de voltaje dominante. Este método tiene cero falsos positivos y puede distinguir entre un ataque y un error. Sin embargo, necesita un osciloscopio para adquirir señales de una red, y el procesamiento de señales requiere mucho trabajo.

En la red CAN no existe un reloj maestro compartido con la red, cada ECU usa el suyo. Cho y Shin sugirieron utilizar la desviación del reloj para la detección de intrusos. Las ECUs funcionan a la misma frecuencia, pero pueden tener una deriva aleatoria de más de 2400 ms por día. Tomaron los datos del transmisor de la ECU a través del sesgo del reloj y detectaron las intrusiones. Pudieron lograr una detección de anomalías del 97 % con una tasa de falsos positivos del 0,55 %, pero este método solo funcionó para mensajes periódicos. Sin embargo, este método se puede engañar simulando un desfase del reloj.

Las características físicas de CAN Bus proporcionan información esencial sobre las ECUs. Sin embargo, los factores ambientales como la temperatura, la humedad y el enve-

jecimiento de los componentes pueden cambiar las propiedades físicas, resultando en un fallo de la IDS. Además, los ataques a nivel de software no se pueden detectar porque las ECUs autorizadas envían mensajes maliciosos y el IDS no encuentra cambios en las características de la señal. De manera similar, el IDS basado en características físicas requiere un procesamiento de señal intenso, resultando en demoras o requiriendo un equipo costoso.

Larson y otros propusieron la detección de ataques basada en especificaciones e implementaron reglas de especificación basadas en el protocolo CAN Open. Este método tiene capacidades limitadas de detección de intrusos y requiere un detector en cada ECU. Este método tampoco es lo suficientemente robusto para prevenir ataques, ya que existen algunos que son compatibles con el protocolo. Studnia y otros propusieron la detección de intrusiones basada en el lenguaje, derivando esta característica de las especificaciones de las ECUs, generando secuencias prohibidas. Estas secuencias prohibidas son las que indicarían que se está aconteciendo una intrusión en la red.

El análisis de detección de intrusos basado en características se centra en examinar todos los parámetros de la red, como la frecuencia, el número de tramas perdidas, la carga del bus o incluso otras características como los mensajes mal formados. Normalmente todo esto está basado en técnicas de inteligencia artificial. Seo et al., utilizando un modelo de aprendizaje profundo, propusieron un IDS basado en *Generative Adversarial Networks* (GAN). Dado que este método es fácil de escalar y difícil de manipular por un atacante, el mecanismo de detección tiene una característica de caja negra. El filtrado *Bloom* propuesto por Groza y Mervy analiza la periodicidad y la carga útil de los mensajes CAN, proporcionando un análisis de datos en memoria de manera eficiente. Ambos métodos son computacionalmente costosos, pero parecen prometedores en términos de resolución de problemas de seguridad en CAN Bus.

Como hemos visto, cada método tiene unas características únicas que hacen que pueda o no funcionar de manera eficiente, pero también tienen desventajas y un coste. Por lo tanto, la mejor solución encontrada una vez se han analizado todos los IDS sería un sistema híbrido, que aprovechase varios métodos. Aunque IDS podría mitigar el problema de seguridad, necesitaríamos criptografía para poder asegurar la confidencialidad en el sistema. A continuación se realiza una comparación en la tabla 2.5 de los distintos IDS.

Análisis de algoritmos	Parámetros	Ventajas	Desventajas
<i>Generative Adversarial Networks</i> (GAN)	Patrón del ID de CAN	Puede entrenarse para ataques desconocidos	Hardware costoso.
Sistema de inferencia difusa basado en red adaptativa	Carga del bus, frecuencia de mensajes	Solución simple, detecta el tipo de ataque	Funciona para ataques simples que están actualizados ya que necesita una base de datos.
Basado en la entropía	Entropía de los IDs, <i>payload</i>	No requiere mucha información del tráfico de datos	Es muy vulnerable a ataques que incluyen bits aleatorios.
Redes de memoria a largo corto plazo	<i>Payload</i>	No requiere conocimiento previo	No entiende el cambio natural.
Basados en especificación	Política de protocolo	Menor dependencia	El IDS se debe colocar en cada ECU.
<i>Hamming Distance</i>	<i>Payload</i>	Bajo coste computacional	Baja detección
Relación de compensación e intervalo de tiempo	Trama de tiempo remota	Algoritmo simple y eficiente con hardware de bajo costo	Aumento del tráfico.
Análisis de la secuencia de identificación	Secuencia de identificación	Bajo requerimiento de memoria y computación, detección de pocos mensajes maliciosos inyectados	Muy vulnerable a ataques que tienen una secuencia similar de tráfico normal.
Máquina de vectores de soporte y árbol de decisión potenciado	Señal eléctrica	Robusto contra algunos tipos de ataque, primer IDS en diferenciar entre ataque y error	Alto coste y vulnerable a cambios medioambientales.
Mínimos cuadrados recursivos	Sesgo del reloj	Robusto a algunos tipos de ataque	Solo funciona con señales periódicas.
<i>Bloom Filtering</i>	ID, <i>payload</i>	Bajo uso de memoria	Algoritmo complejo.
Función de densidad de probabilidad	Período del ciclo de recepción (análisis de frecuencia)	Aprendizaje <i>online</i>	Difícil de autenticar un mensaje no periódico.
Basado en flujo	Frecuencia de mensajes	Algoritmo sencillo	Solo funciona con señales periódicas.

Tabla 2.5: Comparación de los distintos IDS.

La seguridad en CAN Bus está recibiendo cada vez más atención y las estandarizaciones están cada vez más cerca para poder abordar los problemas de ciberseguridad. En 2016 fue publicada por SAE la guía de ciberseguridad para sistemas de vehículos ciberfísicos y los principios fundamentales de la especificación de ciberseguridad automotriz (PAS 1885:2018) por *British Standards Institute* en 2018. El estándar *ISO/SAE 21434: Road Vehicles – Cybersecurity Engineering*, fue publicado en septiembre de 2021 por SAE International y la Organización Internacional de Normalización (ISO). Debido a esto, las empresas están fabricando ECUs seguras de gama alta gracias al impacto de las vulnerabilidades en CAN Bus. La especificación *Secure Hardware Expansion* (SHE) desarrollada por *Hersteller Initiative* (HIS) se ha convertido en un estándar abierto y es utilizada por muchas empresas en sus ECUs, como el microcontrolador NXP MPC5646C. Además, algunas ECUs comerciales tienen un IDS integrado, la serie NXP TJA115x puede evitar ataques de suplantación de identidad y utilizarse como IDS. También existen sistemas de detección de intrusos patentados. Si bien es cierto que se han tomado muchas medidas para proteger el CAN Bus, todavía necesita mucho por desarrollar. La industria no comparte parte de su investigación, no hay suficientes datos de ataques ni puntos de referencia, por lo que es necesario mucha más investigación para poder desarrollar buenas defensas frente a los diversos ataques.

2.3. Can-utils

Can-utils es un conjunto de herramientas para trabajar con el bus CAN en Linux usando el protocolo SocketCAN. El protocolo SocketCAN es una implementación del nivel de enlace de datos de CAN Bus que utiliza el modelo de sockets de Linux para proporcionar una interfaz uniforme y transparente a las aplicaciones de usuario. El paquete *can-utils* proporciona un conjunto de herramientas utilizadas para tratar los datos dentro de una red *CAN Bus*. Estas herramientas proporcionan muchas funciones diferentes, algunas de ellas son:

- **Candump.** Herramienta capaz de capturar el tráfico de la red y mostrar los paquetes que circulan por la red CAN. Se puede usar para monitorizar el tráfico CAN en tiempo real o para guardar los datos en un formato legible por humanos o binario para su posterior análisis. También se puede usar para aplicar filtros basados en el identificador de arbitraje o la máscara de bits a los datos recibidos.
- **Cansend.** Herramienta que permite transmitir mensajes a través de la red CAN. Se puede usar para enviar comandos o solicitudes a los dispositivos conectados al CAN Bus.
- **Cangen.** Herramienta capaz de generar tráfico de red para, posteriormente, probar o depurar sistemas CAN. Se puede usar para probar el rendimiento o la robustez de CAN Bus o de los dispositivos conectados. También se puede usar para especificar el rango, la frecuencia, la longitud y el tipo de los marcos CAN generados.
- **Canplayer.** Herramienta que permite la reproducción de paquetes CAN previamente capturados. Se puede usar para simular el tráfico CAN a partir de los datos grabados con *candump* u otras fuentes. También se puede usar para modificar los tiempos, las velocidades o los identificadores de los marcos CAN durante la reproducción.
- **Cansequence.** Herramienta para analizar y enviar secuencias de tramas CAN. Se puede usar para verificar la integridad o la pérdida de datos en el bus CAN. También se puede usar para especificar el identificador, la longitud, el tipo y el incremento de los marcos CAN enviados.
- **Cansniffer.** Herramienta de monitorización en tiempo real de los paquetes CAN que circulan por la red. Además ofrece la posibilidad de aplicar filtros, marcar bits, leer y escribir archivos de configuración, cambiar el modo de salida (binario o hexadecimal) y el color.

Aunque existen muchas más herramientas estas son las más básicas a la hora de capturar, mostrar, generar o enviar tráfico CAN. Podemos encontrar el proyecto en el repositorio de GitHub.

2.4. SavvyCAN

SavvyCAN es una herramienta de código abierto que ayuda a monitorizar, filtrar, analizar, capturar y decodificar tráfico CAN en tiempo real. Este software está programado en C++ basado en QT, originalmente escrito para utilizar hardware EVTV como el hardware EVTVDue y CANDue. Desde entonces, se ha ampliado para poder utilizar cualquier dispositivo compatible con socketCAN, puede capturar y enviar a múltiples buses y dispositivos de captura CAN a la vez. Esta herramienta es comúnmente utilizada en la industria del automóvil y en otros campos que hagan uso de las redes CAN Bus, como la automatización industrial o la robótica.

Una de las particularidades de esta herramienta es que ofrece una interfaz gráfica que permite visualizar los datos de la red *CAN Bus* (Figura 2.8). Además, este software es compatible tanto para sistemas Linux como sistemas Windows, lo que la hace más accesible al usuario.

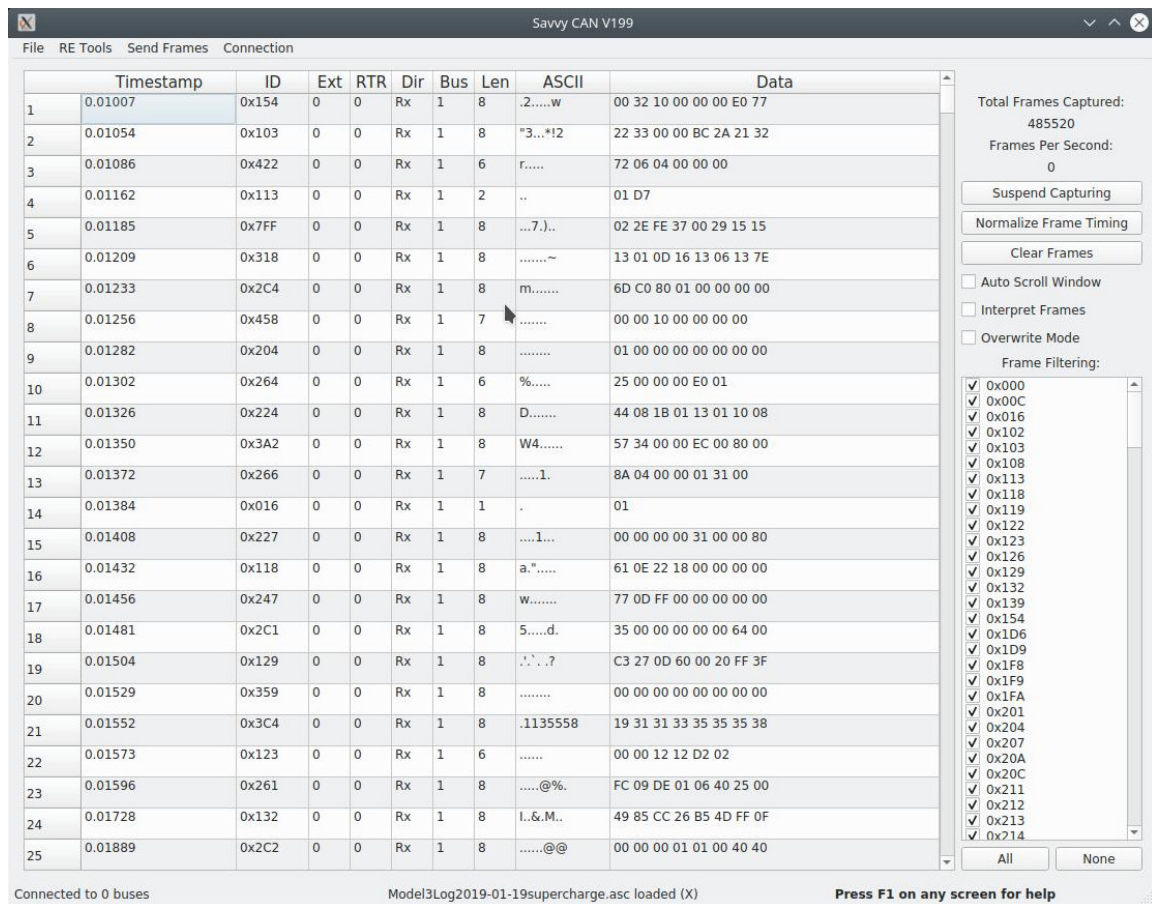


Figura 2.8: Interfaz gráfica SavvyCAN. Fuente: SavvyCAN

2.5. Kayak

Kayak es una herramienta de código abierto implementada en Java cuyo objetivo principal es el diagnóstico y la monitorización del protocolo CAN Bus. La aplicación cuenta con una interfaz gráfica de usuario simple que facilita su uso, principalmente para capturar y analizar el tráfico de mensajes CAN en tiempo real. Además, la herramienta es compatible con diferentes interfaces *hardware* gracias a *socket* CAN y TCP/IP, ya que se usan como una capa de abstracción sobre el controlador *hardware* CAN. Se establece una conexión entre el dispositivo *Socket* CAN y *socketcand* sobre una máquina Linux y el *socket* TCP/IP de Kayak.

Socketcand es un demonio que forma parte de la implementación de *SocketCAN* y proporciona acceso a las interfaces CAN de una máquina por medio de la interfaz de red. El protocolo de comunicación utiliza una conexión TCP/IP y un protocolo específico para transmitir los mensajes CAN y los comandos de control. Esto permite que las aplicaciones que utilizan el protocolo CAN (en este caso Kayak) se comuniquen con otros sistemas CAN.

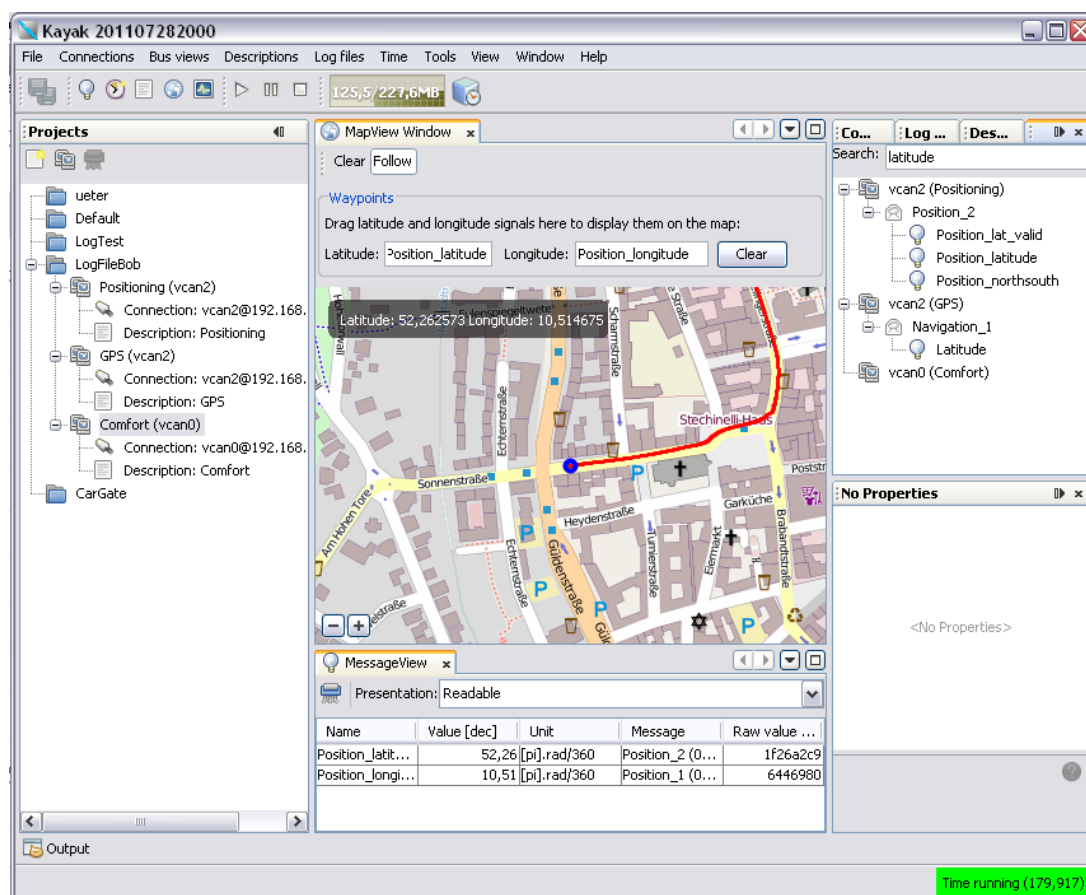


Figura 2.9: Ejemplo de uso de Kayak. Fuente: Github

Kayak se vincula con OpenStreetMaps (Figura 2.9) para realizar el mapeo y gestionar los mensajes CAN. Como aplicación basada en Java es independiente de la plataforma, por lo que se apoya en *socketcand* para manejar la comunicación con los dispositivos CAN.

2.6. PCAN-View

PCAN-View (Figura 2.10) es un software para Windows que permite visualizar, transmitir y grabar mensajes CAN y CAN FD. Se suministra con cada producto de hardware PCAN para PC y permite un inicio rápido y fácil. Para utilizar esta herramienta, se necesita una interfaz PCAN de PEAK-System que se conecte al bus CAN y al ordenador mediante USB, PCI, PC/104 o cualquier otro dispositivo compatible. La herramienta se puede descargar desde el sitio web de PEAK-System. Algunas de sus características son:

- Soporte para las especificaciones CAN 2.0 A/B y FD.
- Tasas de bits de datos hasta 12 Mbit/s (solo al usar una interfaz CAN FD).
- El modo de solo escucha se puede activar.
- Transmisión manual y periódica de mensajes con una resolución máxima de 1 ms.
- Recepción de mensajes con una resolución máxima de 100 μ s.
- Grabación de mensajes en archivos de traza.
- Guardado y recarga de mensajes de transmisión.
- Representación de los identificadores CAN en hexadecimal o decimal.
- Representación de los bytes de datos en hexadecimal, decimal o ASCII.
- Visualización de los estados de recepción, transmisión y error.
- Acceso a la configuración e información específicas del hardware.

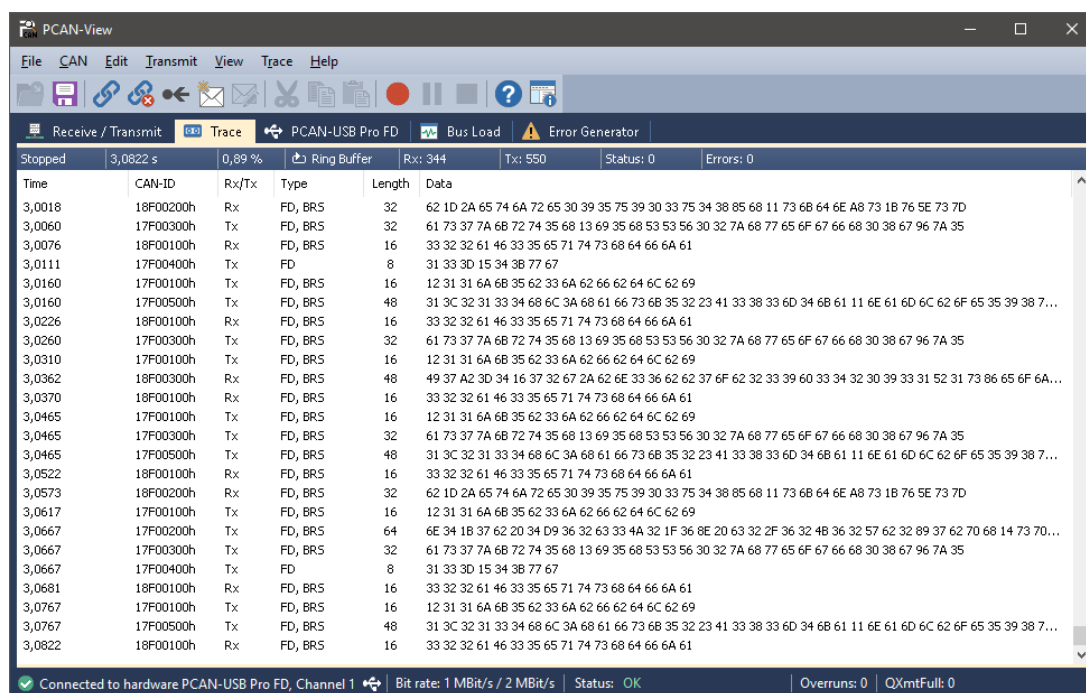


Figura 2.10: Interfaz gráfica PCAN-View. Fuente:Peak-system

PCAN-View es una herramienta que ofrece ventajas y desventajas para el análisis de redes CAN y CAN FD. Algunas de las ventajas son:

- Es una herramienta gratuita y fácil de usar que se suministra con cada producto de hardware PCAN de PEAK-System.
- Soporta las especificaciones CAN 2.0 A/B y FD, así como tasas de bits personalizadas y el modo de solo escucha.
- Permite la transmisión manual y periódica de mensajes con una alta resolución temporal.
- Permite la grabación de mensajes en archivos de traza que se pueden abrir con otras herramientas como *Plot Viewer*.
- Permite el acceso a la configuración e información específicas del hardware, así como el reinicio del controlador CAN.
- Dispone de un generador de errores para simular condiciones de error en el bus CAN.

Sin embargo, también presenta algunas desventajas:

- Es una herramienta simple que no ofrece funciones avanzadas como el análisis de protocolos, la simulación de nodos o la generación de informes.
- Requiere una interfaz PCAN compatible para conectarse a CAN Bus.
- No soporta otras interfaces o protocolos como LIN, J1939 o UDS.
- No dispone de una versión para Linux u otros sistemas operativos.

CAPÍTULO 3

Herramientas empleadas

3.1. Python

Python es un lenguaje de programación de alto nivel, diseñado para ser sencillo de leer y escribir. Es un lenguaje interpretado, es decir, no necesita ser compilado antes de ser ejecutado y orientado a objetos. Fue creado por Guido van Rossum a principios de los años 90 y desde entonces se ha convertido en uno de los lenguajes de programación más famosos y ampliamente utilizado en la actualidad.

Una de las características principales de este lenguaje de programación es su sencilla sintaxis, lo que hace que sea legible y facilita su escritura para obtener un código claro y conciso. Python es muy versátil, puede ser utilizado para una gran variedad de aplicaciones como el desarrollo de aplicaciones web, la creación de aplicaciones de escritorio, ciencia de datos, aprendizaje automático, creación de herramientas de automatización o creación de *scripts* entre otros.

Python es un lenguaje de programación de código abierto, es decir, su código fuente está disponible para su modificación y distribución por cualquier persona. Todo esto hace que este lenguaje de programación tenga una comunidad muy activa de desarrolladores, creando y manteniendo una gran cantidad de bibliotecas y módulos de terceros. Estos módulos y bibliotecas son una parte fundamental de la programación de Python.

Un módulo es un archivo que contiene definiciones de funciones, clases y variables que pueden ser reutilizadas en otros programas. Un conjunto de módulos utilizados para realizar distintas tareas específicas es una biblioteca. Python tiene una gran cantidad de módulos y bibliotecas disponibles, tanto de su biblioteca estándar como de terceros. En la biblioteca estándar de Python se encuentran módulos para realizar tareas como la manipulación de archivos, gestión de base de datos entre otros. Sin embargo también hay bibliotecas de terceros disponibles para multitud de temáticas diferentes como se ha mencionado anteriormente.

Los módulos y las bibliotecas son de gran utilidad y tienen una gran importancia ya que permiten a los programadores reutilizar código ya existente en lugar de escribir todo desde cero. Esto ahorra tiempo y esfuerzo además de asegurar calidad en el código ya que estos módulos y bibliotecas han sido probados y revisados por toda la comunidad en muchos proyectos diferentes.

Para hacer uso de estas bibliotecas de Python, la mayoría se pueden instalar a través de un administrador de paquetes como pip, haciendo que sea fácil y rápido para poder agregar nuevas funcionalidades al proyecto.

Existen varias versiones que se han ido desarrollando con el paso del tiempo, incluyendo mejoras y funcionalidades nuevas al lenguaje de programación. Sin embargo, a veces se hacen modificaciones en versiones nuevas que no son compatibles con las anteriores, requiriendo una modificación del código.

La primera versión pública de Python, la versión 0.9.0, se lanzó en 1991. La versión 1.0 se lanzó en 1994, seguida de la versión 2.0 en 2000, que introdujo nuevas características como la función de generador y la capacidad de anular métodos. La versión 2.0 se mantuvo como la versión principal durante muchos años y aún se utiliza en algunos proyectos antiguos. En 2008, se lanzó la versión 3.0 de Python, que incluyó cambios importantes en el lenguaje. En particular, se eliminaron características obsoletas y se mejoró el manejo de errores. También se introdujeron nuevas características, como la división verdadera y la sintaxis para el manejo de bytes. Actualmente la versión más actualizada es Python 3.11.2 lanzada el 8 de febrero de 2023 con mejoras de eficiencia y optimización y algunas nuevas funcionalidades.

Existen diversas bibliotecas para tratar los paquetes can, una de ellas es python-can. Python-can es una biblioteca de Python que ofrece soporte para la comunicación con CAN Bus. La biblioteca proporciona abstracciones comunes para diferentes dispositivos CAN, así como una serie de utilidades para enviar y recibir mensajes en el bus CAN. También es compatible con CPython y PyPy, y funciona en Mac, Linux y Windows. Además, soporta tanto los identificadores de arbitraje normales como los extendidos, así como el formato CAN FD. La biblioteca también permite leer y escribir archivos de registro en diferentes formatos, aplicar filtros de mensajes en el kernel o en el hardware, leer la configuración del bus desde un archivo o desde variables de entorno, y usar herramientas de línea de comandos para trabajar con los buses CAN.

Otra de las bibliotecas que usaremos en este proyecto ya que trabajamos con una aplicación en la terminal de Linux es python-can-Viewer. Python-can-Viewer es una biblioteca de Python que ofrece una aplicación de terminal simple para visualizar los mensajes de CAN Bus. Soporta Python 2, Python 3, pypy y pypy3. Utiliza la biblioteca python-can para acceder a los diferentes dispositivos CAN y mostrar los mensajes recibidos en una tabla con diferentes columnas: el número de veces que se ha recibido un mensaje con un determinado ID, el tiempo relativo al primer mensaje, el tiempo entre el mensaje actual y el anterior, la longitud del mensaje, los datos, y el código de función y el ID de nodo de CANopen (si se usa). La biblioteca también permite especificar cómo decodificar los datos en bruto a valores reales usando el formato del paquete struct de Python.

3.2. Python curses

Python curses es una biblioteca de Python que proporciona una función de dibujo y manejo de teclado independiente de la terminal para terminales basadas en texto, como la terminal de Linux. La visualización en las terminales admiten varios códigos de control para realizar operaciones comunes, como mover el cursor, desplazar la pantalla y borrar áreas. Aunque las terminales de visualización basadas en caracteres son una tecnología obsoleta, todavía existen ciertos casos donde poder hacer cosas sofisticadas con ellas sigue siendo valioso. Un nicho es en herramientas como instaladores de sistemas operativos y configuradores de kernel que pueden tener que ejecutarse antes de que esté disponible cualquier soporte gráfico. Otro puede ser, como en este caso, la creación de una herramienta que se ejecute desde la terminal.

La biblioteca curses tiene una funcionalidad bastante básica, proporcionando al programador una abstracción de una pantalla que contiene varias ventanas de texto no superpuestas. El contenido de una ventana se puede cambiar de varias maneras: agregando texto, borrándolo, cambiando su apariencia, y la biblioteca curses calculará los códigos de control necesarios para enviar a la terminal para producir la salida correcta.

Con Python curses, es posible crear aplicaciones de línea de comandos con una apariencia y funcionalidad similar a las aplicaciones gráficas. Las aplicaciones pueden tener menús, botones, cajas de texto, y cualquier otra cosa que se pueda imaginar en una aplicación de GUI. Además, se pueden usar colores y diferentes estilos de texto para mejorar la apariencia de la aplicación.

Python curses también permite la captura y el manejo de eventos del teclado, lo que permite la creación de aplicaciones interactivas. Los eventos del teclado pueden ser procesados en tiempo real, lo que significa que los usuarios pueden interactuar con la aplicación a medida que se está ejecutando.

Para este proyecto utilizaremos el módulo curses de Python, es un *wrapper* sobre las funciones de C proporcionadas por la biblioteca curses. En otras palabras, el módulo de Python proporciona una forma más fácil de utilizar la biblioteca curses desde Python, ya que se encarga de manejar las complejidades de la comunicación con la biblioteca en C.

3.3. Instrument Cluster Simulator - ICSim

ICSim es una herramienta *open source* utilizada para pruebas de sistemas electrónicos en vehículos. Su función principal es emular el panel de un automóvil para simular las funciones reales sin necesidad de tener un vehículo físico.

La herramienta permite simular una variedad de escenarios y situaciones que ocurren cuando utilizamos el coche, por ejemplo encender o apagar las luces de intermitencia, acelerar el coche, abrir o cerrar las puertas, mostrar la temperatura, el combustible, etc. Además cuenta con controles para poder manejar de manera virtual las distintas funciones que nos ofrece el simulador.

Todas estas funciones resultan de gran utilidad para el proyecto ya que, junto con otras herramientas, podemos generar tráfico CAN al simular las distintas acciones en un coche que posteriormente podremos tratar con la herramienta.

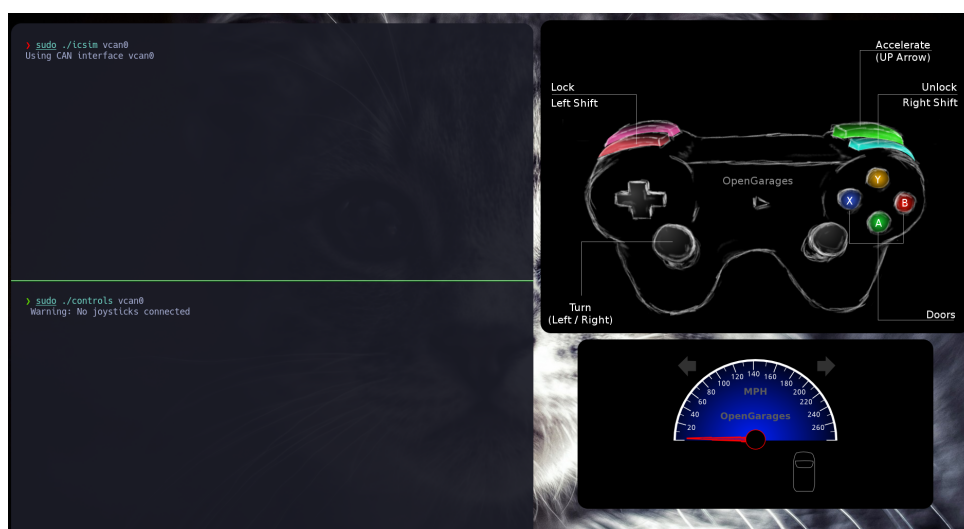


Figura 3.1: Instrument Cluster Simulator

3.3.1. Configuración del entorno

En esta sección se describe el proceso de instalación y configuración de las herramientas necesarias para simular el comportamiento de un vehículo real y poder capturar trazas de la red CAN. Si bien es cierto que realizaremos las pruebas sobre un simulador de cuadro de instrumentos (ICSim) esto se puede trasladar a un coche real, utilizando el hardware y el software específico para poder extraer esta información. Esta configuración es idónea para practicar y aprender el funcionamiento de la red CAN Bus, ya que es un método que no requiere ningún coste adicional y sin riesgo de dañar el vehículo.

Requisitos:

- Cualquier distribución de Linux (en este caso usaré ParrotOS).
- El paquete de herramientas can-utils.
- SocketCAN para la creación de una interfaz virtual.
- ICSim para simular el cuadro de instrumentos del vehículo.

En primer lugar, *Instrument Cluster Simulator* requiere las librerías SDL. Las librerías SDL (Simple DirectMedia Layer) son un conjunto de librerías de desarrollo multiplataforma para gráficos y audio por computadora. Permiten crear aplicaciones que pueden ejecutarse en diferentes sistemas operativos, como Windows, Linux, Mac OS, Android o iOS. Como ICSim dibuja y anima un cuadro de instrumentos virtual, necesita estas librerías para funcionar. Se pueden instalar mediante el gestor de paquetes apt-get en Linux, con el siguiente comando:

```
sudo apt-get install libsdl2-dev libsdl2-image-dev -y
```

A continuación, instalaremos can-utils, un conjunto de herramientas que nos permitirá tanto enviar, recibir o analizar los paquetes CAN. Principalmente se usan 5 herramientas, explicadas en la sección de can-utils: *cansniffer*, *cansend*, *candump*, *canplayer*, *cangen*. Estas herramientas se pueden instalar a través del repositorio de github o con el comando:

```
sudo apt-get install can-utils -y
```

Ahora que can-utils y las dependencias de ICSim están instaladas, vamos a proceder a instalar Instrument Cluster Simulator. Podemos descargarlo clonando el repositorio de Github con el siguiente comando:

```
git clone https://github.com/zombieCraig/ICSim
```

Es importante elegir el directorio donde vamos a clonar la herramienta para tener una buena organización. En mi caso, clonaré el repositorio en el directorio `/opt`. Una vez esté clonado, ya habremos acabado el proceso de instalación de las herramientas.

Ahora, nos movemos dentro del directorio que hemos clonado y encontraremos un script llamado `setup_vcan.sh`. Este script ejecutará una serie de comandos que nos permitirán comunicarnos con CAN. Los comandos que se ejecutan son los siguientes:

```
sudo modprobe can
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

El comando `modprobe` se utiliza para cargar los módulos núcleo como `can` y `vcan`. Los otros dos comandos crean una interfaz virtual `vcan0` para simular la red CAN Bus del vehículo. Podemos ejecutar los comandos de forma manual o podemos ejecutar el script con el comando:

```
./setup_vcan.sh
```

Una vez ejecutado, debemos comprobar que la interfaz se ha creado correctamente. Para ello, ejecutamos el comando:

```
ifconfig vcan0
```

Si todo ha funcionado correctamente, deberíamos de ver la nueva interfaz `vcan0`.

Para ejecutar el simulador se necesitan al menos dos componentes: un cuadro de instrumentos y un panel de control. El cuadro de instrumentos muestra el velocímetro, los indicadores de bloqueo de puertas, los intermitentes y otros elementos. El panel de control permite al usuario interactuar con la red automotriz simulada, aplicando aceleración, frenos, controlando los bloqueos de puertas y los intermitentes.

Para ejecutar el simulador se requieren al menos tres ventanas o pestañas de terminal. Una para ejecutar el cuadro de instrumentos, otra para ejecutar el panel de control y otra para ejecutar las utilidades CAN. Para ejecutar el cuadro de instrumentos se puede usar un archivo llamado `icsim` dentro del directorio clonado que recibe como argumento la interfaz `vcan0`, la interfaz virtual que creamos anteriormente, con el comando:

```
./icsim vcan0
```

En este punto, debemos simular el control para poder enviar los mensajes al simulador a través de la interfaz `vcan0`, como por ejemplo la aceleración, los intermitentes, abrir las puertas, etc. Con la interfaz `vcan0` el simulador será capaz de enviar y recibir las tramas CAN. Tan pronto como se inicie el panel de control, se puede observar que el velocímetro hace algunas fluctuaciones. Esto se debe al ruido simulado por el panel de control. Una vez iniciado el panel de control, se pueden usar las teclas del teclado para simular el tráfico. Para iniciar el panel de control, nos situaremos en el directorio `ICSim` que hemos clonado y ejecutaremos el siguiente comando:

```
./controls vcan0
```

Una vez tenemos el panel de control, a través de los siguientes atajos de teclado podemos simular las distintas acciones que enviaremos al cuadro de instrumentos:

Acciones ICSim	Teclas
Izquierda / Derecha	Flechas izquierda o derecha
Acelerar	Flecha hacia arriba
Desbloquear las puertas delanteras	Shift-Derch+A, Shift-Derch+B
Desbloquear las puertas traseras	Shift-Derch+X, Shift-Derch+Y
Bloquear todas las puertas	Mantener Shift-Derch, presionar Shift-Izq
Desbloquear todas las puertas	Mantener Shift-Izq, presionar Shift-Derch

Tabla 3.1: Atajos de teclado del panel de control

En este punto, ya estaría todo configurado para poder capturar y enviar paquetes por la red. Nuestro entorno se vería como se muestra en la Figura 3.1.

3.4. SocketCAN

Existe una gran variedad de aplicaciones y herramientas para comunicarnos con el protocolo CAN Bus. Muchas de ellas (desarrolladas en Linux) utilizan **SocketCAN**, una tecnología que implementa diferentes protocolos CAN para Linux y nos permite comunicarnos entre los distintos componentes en sistemas embebidos o de control. Además, utiliza *Berkeley Socket API*, la pila de red de Linux e implementa *CAN device drivers* como interfaz red (sobre la capa red de Linux), lo que permite comunicarse con la aplicación.

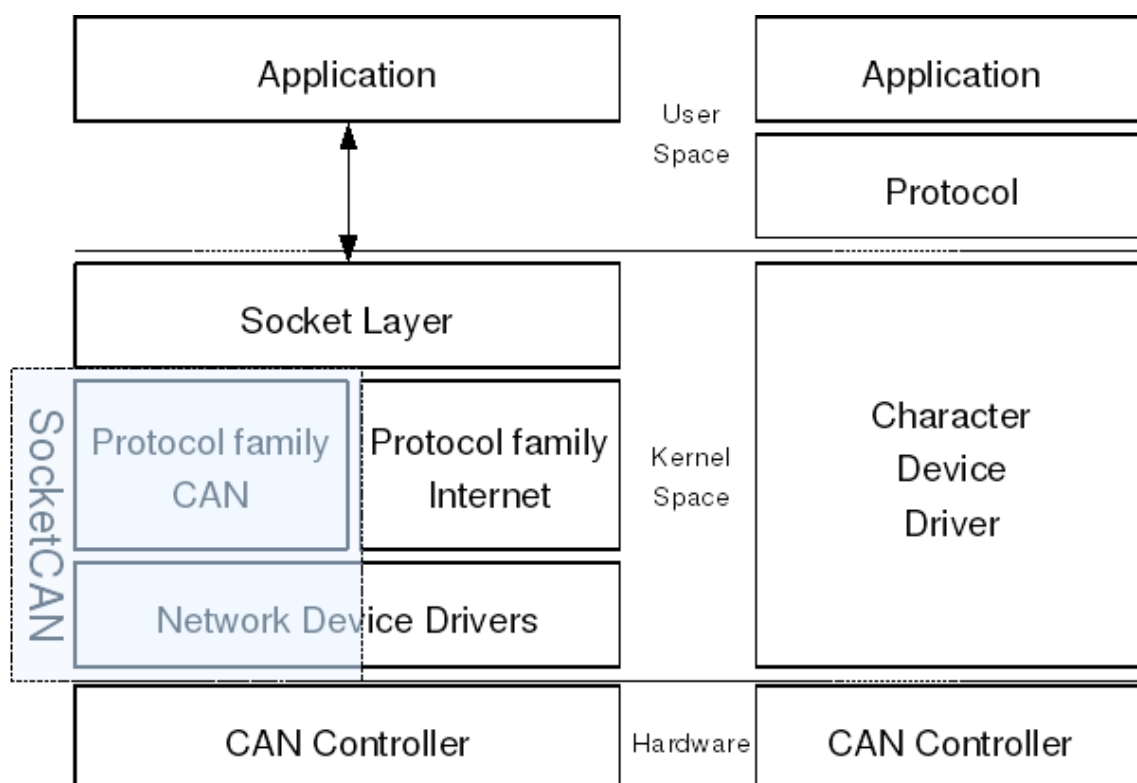


Figura 3.2: Capas típicas de comunicación CAN, con SocketCAN (izquierda) o convencional (derecha). Fuente: WIKIPEDIA

SocketCAN fue creada en 2006 en BerliOS (Berlin Open Source), cuyo objetivo era proporcionar una plataforma para el desarrollo de software libre y de código abierto. *Volkswagen Group Research* contribuyó a la implementación de SocketCAN para dar soporte a chips CAN, conexión USB externa y serial a dispositivos CAN y dispositivos CAN virtuales.

Berkeley Socket API, también conocida como *BSD Socket API* es una interfaz de programación de aplicaciones (API) para la comunicación en red de sistemas operativos basados en UNIX. Fue desarrollada en la década de 1980 en la universidad de California y, en la actualidad, se ha convertido en una de las APIs más utilizadas para el desarrollo de aplicaciones de red en multitud de sistemas operativos. *Berkeley Socket API* proporciona un conjunto de funciones y estructuras de datos para crear, configurar y administrar sockets, que actúan como los puntos finales de las conexiones de red. Esto permite a los programadores a la hora de desarrollar poder establecer una comunicación entre procesos

que se ejecutan en la misma o distinta máquina a través de la red.

Un socket designa un concepto abstracto por el cual dos procesos (posiblemente situados en computadoras distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. Además es una estructura de datos que permite que dos máquinas se comuniquen entre ellas. En *Berkeley Socket API* se representa como un descriptor de archivo (manejador de archivo) en la filosofía Unix que proporciona una interfaz común para la entrada y salida a flujos de datos.

Berkeley Socket API proporciona diferentes funciones, algunas de las principales incluyen:

- ***socket()***. Esta función se utiliza para crear un nuevo socket y devuelve un descriptor de archivo que representa al socket.
- ***bind()***. Se utiliza para asociar un socket con una dirección IP y un número de puerto específicos en el sistema local. Esto es necesario antes de que el socket pueda recibir conexiones entrantes.
- ***listen()***. Esta función marca un socket como un socket pasivo que puede aceptar conexiones entrantes. Se utiliza después de llamar a la función *bind()*.
- ***accept()***. Se utiliza para aceptar una conexión entrante en un socket pasivo. Esta función crea un nuevo socket que representa la conexión aceptada y devuelve un descriptor de archivo para ese socket.
- ***connect()***. Esta función se utiliza para establecer una conexión saliente desde un socket cliente a un socket servidor remoto.
- ***send()* y *recv()***. Estas funciones se utilizan para enviar y recibir datos a través de un socket establecido.

Además de estas funciones básicas, *Berkeley Socket API* proporciona funciones para establecer las opciones de configuración de socket, manejo de errores, administración de direcciones de red y otras operaciones relacionadas con la comunicación de red. Es importante tener en cuenta que *Berkeley Socket API* es una interfaz de bajo nivel, lo que significa que ofrece una gran flexibilidad pero también requiere una comprensión detallada de los conceptos y protocolos básicos de redes. Se convirtió en la base para muchos otros marcos y bibliotecas de redes de alto nivel, como *libcurl* y *libevent*.

SocketCAN es un proyecto de gran utilidad e importancia, ya que previamente a esto existían otras implementaciones para conectar el hardware. La mayor parte de ellas consistían de drivers basados en el carácter del dispositivo hardware donde las funcionalidades eran muy reducidas. Además, estas implementaciones eran dependientes del hardware, es decir, solo funcionaban para ese dispositivo en concreto, de forma que si cambiabas de dispositivo hardware necesitabas de otra implementación diferente para hacerlo funcionar. Básicamente proporcionan una interfaz capaz de enviar o recibir tramas CAN directamente o desde el controlador hardware. El encolado de tramas y los protocolos de transporte de nivel superior como ISO-TP debían de implementarse en aplicaciones del entorno del usuario. Además, estas implementaciones dependientes del hardware solamente soportaban un único proceso a la vez para abrir el dispositivo, similar a la interfaz serial. Por lo tanto, SocketCAN fue diseñado para quitar todas estas limitaciones.

Se implementaron un conjunto de protocolos, proporcionando una interfaz de *socket*

al espacio de aplicaciones del usuario, la cual está basada en la capa de red de Linux, habilitando el uso de todas las funcionalidades de cola. Un driver del dispositivo hardware para el controlador CAN se registra él mismo en la capa de red de Linux como un dispositivo de la red, de manera que las tramas CAN del controlador pueden pasar de la capa de red al conjunto de módulos CAN implementados por SocketCAN, al igual que en el sentido contrario. Además, este conjunto de módulos ofrecen una API para el registro de los módulos de transporte, de manera que se pueda cargar o descargar dinámicamente cualquier número de protocolos de transporte. Es necesario que se pueda cargar al menos un módulo para usar el módulo central CAN ya que este, de forma aislada, no proporciona ningún protocolo y no podría utilizarse. Existe la posibilidad de cargar múltiples sockets al mismo tiempo, en el mismo o en diferente módulo, y pueden escuchar o enviar tramas en diferentes o mismos CAN IDs. Cuando una aplicación quiere comunicarse haciendo uso de un protocolo específico de transporte, por ejemplo ISO-TP, selecciona el protocolo cuando se abre el socket para, a continuación, leer o escribir bytes de datos en línea sin tener que lidiar con los CAN IDs.

SocketCAN es una implementación de CAN basada en sockets, que ofrece varias ventajas frente a otras implementaciones basadas en dispositivos de carácter . Algunas de estas ventajas son:

1. SocketCAN se integra con la pila de red de Linux, lo que permite aprovechar toda la funcionalidad de encolamiento, multiplexación y filtrado que ofrece el sistema operativo. Además, permite usar las mismas herramientas y librerías que se usan para otros protocolos de red, como TCP o UDP.
2. SocketCAN ofrece una interfaz de socket uniforme y estandarizada a las aplicaciones de espacio de usuario, lo que facilita el desarrollo y la portabilidad de las mismas. Las aplicaciones no tienen que preocuparse por los detalles específicos de cada controlador o dispositivo CAN, sino que pueden usar los mismos tipos de socket y opciones para comunicarse con ellos.
3. SocketCAN soporta varios tipos de socket, que permiten adaptarse a las diferentes necesidades y características de las aplicaciones. Por ejemplo, el socket RAW permite enviar y recibir mensajes CAN sin procesar, lo cual es útil para aplicaciones de bajo nivel o de diagnóstico; mientras que el socket BCM permite configurar tareas de transmisión y recepción periódicas o basadas en eventos, lo que es útil para aplicaciones de alto nivel o de control.
4. SocketCAN soporta el estándar CAN FD, que permite aumentar la velocidad y el tamaño de los mensajes CAN, lo que mejora el rendimiento y la eficiencia de la comunicación. SocketCAN también permite elegir entre los modos ISO y no ISO, que se diferencian en la forma de codificar los bits en el bus CAN.
5. SocketCAN soporta una gran variedad de controladores y dispositivos CAN, tanto internos como externos, lo que aumenta la compatibilidad y la flexibilidad de la comunicación. SocketCAN también permite configurar y monitorizar los parámetros y el estado de los controladores y dispositivos CAN usando la interfaz netlink y la herramienta ip.
6. SocketCAN permite una mayor escalabilidad y flexibilidad de la comunicación, ya que se pueden crear interfaces virtuales CAN que no dependen de ningún controlador o dispositivo físico. Estas interfaces virtuales pueden usarse para simular redes CAN,

crear puentes entre redes CAN, o aislar aplicaciones de la red real.

Sin embargo, una posible desventaja de las implementaciones de CAN basadas en dispositivos de carácter frente a SocketCAN es que no pueden aprovechar las funcionalidades de la pila de red de Linux, como el encolamiento, el filtrado o el multiplexado de mensajes. Además, las implementaciones basadas en dispositivos de carácter suelen ser específicas del hardware y requieren adaptar las aplicaciones a la API del controlador. Por el contrario, SocketCAN ofrece una interfaz unificada y transparente para la comunicación CAN, independientemente del controlador o la interfaz utilizados.

Otra posible desventaja es que las implementaciones basadas en dispositivos de carácter no siguen la abstracción habitual de los dispositivos Unix, donde se separa la capa de hardware de la capa de aplicación. Por ejemplo, no se tiene un dispositivo de bloque o carácter para un controlador SCSI o IDE específico, sino que se usa un subsistema que proporciona una interfaz común para todos los dispositivos. SocketCAN hace lo mismo para los dispositivos CAN, usando el marco de red de Linux como subsistema.

Como se ha mencionado anteriormente, existen diferentes tipos de sockets disponibles en SocketCAN:

- **Socket RAW:** Permite enviar y recibir mensajes CAN sin procesar, es decir, sin ningún protocolo de transporte o filtro de contenido. Es el tipo de socket más básico y flexible, que ofrece un acceso directo al bus CAN. Los mensajes CAN se representan mediante la estructura *struct can_frame* o *struct canfd_frame*, que contienen el identificador CAN, la longitud de la carga útil y los datos. Los sockets RAW se pueden configurar mediante varias opciones, como el filtro de identificadores CAN, la recepción de mensajes de error, el bucle local o la recepción de los propios mensajes enviados. El formato de mensaje para comunicarse con los sockets RAW es el mismo que el de la estructura *struct can_frame* o *struct canfd_frame*, precedido por una cabecera que indica la longitud y el tipo del mensaje.
- **Socket BCM:** Permite configurar tareas de transmisión y recepción periódicas o basadas en eventos, que se ejecutan en el espacio del núcleo. Es un tipo de socket más avanzado y complejo, que ofrece una interfaz basada en comandos para filtrar y enviar mensajes CAN cíclicos o secuenciales. Los filtros de recepción se pueden usar para reducir la frecuencia de los mensajes, detectar eventos como cambios de contenido o longitud, y hacer un seguimiento del tiempo de espera de los mensajes recibidos. Las tareas de transmisión periódicas pueden crear y modificar en tiempo de ejecución, tanto el contenido del mensaje como los dos posibles intervalos de transmisión. El formato de mensaje para comunicarse con los sockets BCM es una estructura *struct bcm_msg_head*, que contiene un código de operación, un identificador CAN, unas banderas y unos parámetros temporales, seguida por cero o más estructuras *struct can_frame* o *struct canfd_frame*, que contienen los datos a enviar o filtrar.

La manera más sencilla para implementar los drivers CAN es con una capa de abstracción con carácter de dispositivo, como se hace con la mayoría de los drivers. Sin embargo, la forma más eficiente sería añadiendo todas las funcionalidades a esa capa, como el registro de ciertos CAN IDs, encolando las tramas CAN, o proporcionando una API para el registro de los drivers del dispositivo. Para esto, la forma menos costosa sería haciendo uso del framework de red que nos proporciona el kernel de Linux, tal y como lo hace

SocketCAN.

El objetivo principal de SocketCAN es proporcionar una interfaz socket para las aplicaciones en el espacio de usuario construida sobre la capa de red de Linux. Si hacemos una comparación con la conocida red TCP/IP y ethernet, CAN Bus es un medio broadcast que no tiene capa de direccionamiento MAC, donde el CAN ID se trata como una dirección de origen ya que cada ID es único y está asociado a una ECU.

Existe un problema dentro de la red cuando diferentes aplicaciones quieren hacer uso de los mismos identificadores CAN de la misma interfaz CAN. Solo una aplicación podría recibir las tramas CAN de un determinado identificador, lo que limitaría la funcionalidad y la transparencia de la red. Para solventar este problema, se crearon las *receive lists*, donde varias aplicaciones pueden suscribirse a los identificadores CAN que les interesan y recibir los marcos CAN correspondientes, sin interferir entre sí. Esto permite una mayor flexibilidad y eficiencia en el uso de la red CAN. Las *receive lists* son una característica de la API de SocketCAN que permite al usuario especificar una lista de identificadores CAN que el controlador CAN debe aceptar. Cuando se recibe una trama CAN por el controlador, se compara con la lista de identificadores en la *receive list*. Si el identificador coincide con uno de los identificadores de la lista, entonces la trama se acepta y se pasa a la aplicación. Si el identificador no coincide con ninguno de los identificadores de la lista, entonces la trama se rechaza y se descarta. Esta característica se puede utilizar para filtrar las tramas no deseadas y reducir la carga en la aplicación.

El bus CAN es un medio de difusión que no tiene direcciones MAC como se ha explicado anteriormente. Los identificadores CAN se usan para el arbitraje en la red, por lo que tiene que ser único en el bus. Al diseñar una red de ECUs CAN, los identificadores CAN se asignan para ser enviados por una ECU específica. Por esta razón, un identificador CAN se puede tratar mejor como una especie de dirección de origen. El acceso transparente a la red de múltiples aplicaciones implica el problema de que diferentes aplicaciones pueden estar interesadas en los mismos identificadores CAN, mencionado anteriormente. Es por esto que existe el *loopback* local de las tramas enviadas. Es una funcionalidad que permite que una trama CAN que se ha transmitido con éxito por una interfaz de red se reciba de nuevo por la misma interfaz. Esto permite que las aplicaciones que escuchan en la misma interfaz reciban las tramas que han enviado, reflejando el comportamiento estándar de las redes TCP/IP. El *loopback* local se realiza justo después de la transmisión exitosa, para reflejar el tráfico real en el nodo. Además, se puede realizar por el controlador de la interfaz CAN o por el módulo SocketCAN core como solución alternativa. Esta función está habilitada por defecto y se puede deshabilitar opcionalmente para cada socket de manera individual. El *loopback* local es útil para ejecutar herramientas de análisis como ‘*candump*’ o ‘*cansniffer*’ en el mismo nodo.

El uso del bus CAN puede generar varios problemas en la capa física y de control de acceso al medio. Estos problemas pueden afectar al funcionamiento de la red y causar errores de comunicación entre los dispositivos. Para detectar y registrar estos problemas, el controlador de la interfaz CAN puede generar unos mensajes especiales llamados *Error Message Frames*, que pueden ser enviados a la aplicación de usuario de la misma forma que otros mensajes CAN. Estos mensajes contienen información sobre el tipo y la causa del error, así como la marca de tiempo exacta de su ocurrencia. La aplicación de usuario puede solicitar la recepción de estos mensajes usando los mecanismos comunes de filtrado CAN. Por defecto la recepción de los mensajes de error está deshabilitada. El

formato de los mensajes de error CAN se describe brevemente en el archivo de cabecera “include/uapi/linux/can/error.h”.

El soporte para CAN FD en SocketCAN implica algunas diferencias con el CAN clásico, tanto en el nivel de controlador como en el nivel de aplicación. CAN FD es una extensión del protocolo CAN que permite aumentar la velocidad y la longitud de los mensajes, mejorando el rendimiento y la eficiencia de la comunicación. Para ello, se introducen los conceptos de fase de arbitraje y fase de datos, que pueden tener diferentes parámetros de temporización y velocidad de bits. Los parámetros adicionales de temporización y velocidad de bits para CAN FD son los siguientes:

- `dbitrate`: la velocidad de bits para la fase de datos, que debe ser igual o mayor que la velocidad de bits para la fase de arbitraje (`bitrate`).
- `dsample-point`: el punto de muestreo para la fase de datos, que indica el porcentaje del tiempo cuántico (`dtq`) en el que se muestrea el bit.
- `dtq`: el tiempo cuántico para la fase de datos, que es el inverso del producto del reloj del controlador (`clock`) y el preescalador del bit (`dbrp`).
- `dprop-seg`: el segmento de propagación para la fase de datos, que indica el número de tiempos cuánticos que se usan para compensar el retardo de propagación del bus.
- `dphase-seg1`: el segmento de fase 1 para la fase de datos, que indica el número de tiempos cuánticos que se usan para sincronizar el transmisor con el receptor.
- `dphase-seg2`: el segmento de fase 2 para la fase de datos, que indica el número de tiempos cuánticos que se usan para estabilizar el bit.
- `dsjw`: el ancho del salto de sincronización para la fase de datos, que indica el número máximo de tiempos cuánticos que se pueden ajustar para corregir las diferencias entre los relojes.

Estos parámetros se pueden configurar mediante la interfaz `netlink` usando el programa “`ip`” del conjunto de utilidades “`IPROUTE2`”, añadiendo una “`d`” al principio de cada argumento.

Además, para habilitar el modo CAN FD en el controlador, se debe añadir la opción “`fd on`” al final del comando. Esto también cambia la unidad máxima de transferencia (MTU) del dispositivo a 72 (`CANFD_MTU`), lo que permite enviar y recibir mensajes con hasta 64 bytes de carga útil.

Los mensajes CAN FD se representan mediante la estructura `struct canfd_frame`, que contiene el identificador CAN, la longitud y los flags de la carga útil y los datos. Esta estructura es compatible con la estructura `struct can_frame` usada para los mensajes CAN clásicos, pero tiene un campo adicional `flags` que no está especificado en `struct can_frame`. Por lo tanto, las aplicaciones que usan SocketCAN deben usar `struct canfd_frame` como estructura básica para enviar y recibir mensajes CAN FD.

Otro aspecto a tener en cuenta es que existen dos modos diferentes de implementar CAN FD: el modo ISO y el modo no ISO. El modo ISO es el que cumple con la norma ISO 11898-1:2015 y ofrece una mayor integridad de los datos. El modo no ISO es el que sigue el documento técnico presentado en la Conferencia Internacional CAN 2012 y puede ser incompatible con algunos controladores. El modo actual se anuncia por el controlador

a través de netlink y se puede mostrar con la herramienta “ip” (opción FD-NON-ISO). El modo también se puede cambiar mediante la opción “fd-non-iso on—off” para los controladores que lo permitan.

SocketCAN soporta varios controladores y dispositivos CAN, tanto internos como externos, que se pueden consultar en el archivo “Kconfig” en “drivers/net/can” del árbol de fuentes de Linux. Algunos ejemplos de controladores y dispositivos CAN soportados por SocketCAN son:

- SJA1000: un controlador CAN independiente que se puede conectar a varios buses, como ISA, PCI o SPI.
- MSCAN: un controlador CAN integrado en algunos microcontroladores Freescale MPC5xxx.
- M.CAN: un controlador CAN integrado en algunos microcontroladores Bosch, Infineon, NXP o STMicroelectronics.
- PCAN: una familia de dispositivos CAN externos de PEAK-System, que se conectan por USB, PCI o PCMCIA.

Para finalizar, el protocolo CAN se utiliza ampliamente en la automatización industrial, los dispositivos embebidos y el sector automotriz, ya que permite la comunicación entre diferentes componentes electrónicos de forma fiable, eficiente y robusta. Algunos ejemplos de uso y aplicaciones de SocketCAN en diferentes ámbitos son:

- En la automatización industrial, SocketCAN puede utilizarse para monitorizar y controlar sensores, actuadores, robots, máquinas y otros dispositivos conectados a una red CAN. SocketCAN ofrece una interfaz de socket similar a la de TCP/IP, lo que facilita la programación y la integración con otras herramientas y protocolos de red. Además, SocketCAN permite el uso de filtros, temporizadores, protocolos de transporte y otras funciones avanzadas para gestionar el tráfico y los eventos en la red CAN.
- En los sistemas embebidos, SocketCAN puede utilizarse para desarrollar aplicaciones que se comuniquen con dispositivos CAN como microcontroladores, módulos, placas o periféricos. SocketCAN ofrece una abstracción del hardware específico del controlador CAN, lo que permite la portabilidad y la compatibilidad entre diferentes plataformas y arquitecturas. Además, SocketCAN permite el uso de dispositivos virtuales CAN (vcan) para simular y probar redes CAN sin necesidad de hardware real.
- En el sector automotriz, SocketCAN puede utilizarse para acceder y analizar los datos del bus CAN de los vehículos, que contienen información sobre el estado, el funcionamiento y las averías de los distintos sistemas del vehículo como el motor, la transmisión, los frenos, la dirección, las luces, el aire acondicionado, etc. SocketCAN ofrece una interfaz estándar y uniforme para acceder a los datos del bus CAN, lo que facilita el desarrollo de herramientas de diagnóstico, monitorización, registro o modificación. Además, SocketCAN permite el uso de protocolos específicos del sector automotriz como ISO-TP o J1939 para comunicarse con los dispositivos del vehículo.

SocketCAN es una solución innovadora y flexible para la comunicación con dispositivos

CAN en Linux. Al integrarse con el núcleo y la pila de red de Linux, SocketCAN ofrece una interfaz de socket estándar y uniforme para las aplicaciones de usuario, así como una API para los módulos de protocolo y los controladores de hardware. SocketCAN permite el acceso concurrente y transparente a múltiples interfaces y protocolos CAN, así como el uso de herramientas y funciones de red existentes. SocketCAN también soporta el protocolo CAN FD, que amplía las capacidades del CAN clásico con mayor velocidad y longitud de datos. SocketCAN es una opción robusta, eficiente y versátil para el desarrollo y la integración de sistemas basados en CAN.

3.5. SQLite

SQLite es un sistema de gestión de bases de datos relacional que se caracteriza por ser de código abierto, autocontenido, sin servidor y de configuración cero. Esto significa que SQLite no requiere de un proceso independiente que actúe como servidor, ni de una instalación o administración previa, sino que se integra como una biblioteca dentro del programa que lo utiliza. SQLite almacena toda la información de la base de datos (estructura, tablas, índices y datos) en un único fichero binario, que puede tener un tamaño máximo de 2 terabytes. SQLite implementa la mayor parte del estándar SQL-92, lo que le permite realizar consultas complejas, transacciones atómicas, triggers y funciones definidas por el usuario. Además, soporta múltiples accesos concurrentes de lectura, aunque solo uno de escritura. SQLite utiliza un sistema de tipos dinámicos, que asigna el tipo a los valores individuales y no a las columnas, permitiendo una mayor flexibilidad y compatibilidad con otros lenguajes de programación. Se puede usar con una gran variedad de lenguajes, como C, C++, Java, Python, Perl o PHP, entre otros. SQLite es una herramienta muy útil para desarrollar aplicaciones que requieren de una base de datos ligera, portátil y eficiente, como programas de escritorio, aplicaciones móviles o sitios web estáticos.

Podemos destacar una serie de ventajas que favorecen a este proyecto, por ejemplo:

- Es de código abierto y de dominio público, lo que significa que se puede usar libremente y modificar según las necesidades.
- Es muy ligero y compacto, ocupando solo unos pocos cientos de kilobytes y sin requerir de dependencias externas.
- Es fácil de usar y de integrar, ya que se trata de una simple biblioteca que se enlaza con el programa principal y que no necesita de instalación o configuración previa.
- Es fiable y eficiente, ya que garantiza la integridad y la seguridad de los datos mediante el cumplimiento del modelo ACID, el uso de transacciones y el soporte de cifrado opcional.
- Es portable y compatible, ya que se puede ejecutar en diferentes sistemas operativos y plataformas, y se puede acceder a la base de datos desde diferentes lenguajes de programación.
- Es flexible y versátil, ya que permite trabajar con SQL y con otros tipos de datos como BLOBs, y ofrece la posibilidad de crear funciones definidas por el usuario.

Además, SQLite se puede integrar con Python usando un módulo de Python llamado

sqlite3, el cual ya viene incluido con versiones de Python superiores a 2.5.x. Este módulo proporciona una interfaz compatible.

CAPÍTULO 4

Análisis

4.1. Ingeniería de requisitos

La ingeniería de requisitos es el proceso de especificar, analizar, validar y gestionar los requisitos de un sistema software. Los requisitos son las condiciones o capacidades que el sistema debe satisfacer o poseer para cumplir con las expectativas de los usuarios y los objetivos del proyecto. La ingeniería de requisitos es una actividad fundamental para el éxito de un proyecto software, ya que permite definir el alcance, el comportamiento y la calidad del sistema, así como identificar y resolver los posibles problemas y riesgos que puedan surgir durante el desarrollo. En este proyecto, se ha aplicado la ingeniería de requisitos para definir las características y capacidades que debe tener la herramienta de Python que se ejecuta desde la terminal y que permite al usuario interactuar con el tráfico can bus.

4.1.1. Requisitos funcionales

Los requisitos funcionales son una parte esencial del desarrollo de software, ya que definen lo que el sistema debe hacer para satisfacer las necesidades o expectativas del usuario, en este caso. En este proyecto se han definido los siguientes requisitos funcionales:

RF - 01	Monitorización del tráfico
El sistema debe permitir al usuario monitorizar el tráfico can bus desde la terminal, mostrando los mensajes que se envían y reciben por la red.	

RF - 02	Almacenamiento del tráfico
El sistema debe permitir al usuario almacenar el tráfico CAN Bus en una base de datos, concretamente en una tabla que recibe el mismo nombre que el especificado para el archivo.	
<ul style="list-style-type: none">■ Almacenar la totalidad de los datos recibidos por la interfaz, sin afectar la integridad del contenido.■ Almacenar los datos en columnas con el mismo nombre que se muestran en la función de monitorización de la aplicación.	

RF - 03	Almacenamiento del tráfico en la base de datos
El sistema debe permitir al usuario almacenar el tráfico CAN Bus en una base de datos, concretamente en una tabla que recibe el mismo nombre que el especificado para el archivo.	
<ul style="list-style-type: none">■ Almacenar la totalidad de los datos recibidos por la interfaz, sin afectar la integridad del contenido.■ Almacenar los datos en columnas con el mismo nombre que se muestran en la función de monitorización de la aplicación.	
RF - 04	Almacenamiento del tráfico en un archivo csv
El sistema debe permitir al usuario almacenar el tráfico CAN Bus en un archivo csv, el cual será exportado a la base de datos para almacenar su contenido.	
<ul style="list-style-type: none">■ Almacenar la totalidad de los datos recibidos por la interfaz, sin afectar la integridad del contenido.■ Almacenar los datos en un archivo csv con el nombre que el usuario ha introducido previamente.	
RF - 05	Modificación de las tramas de un archivo csv
El sistema debe permitir al usuario modificar el tráfico CAN Bus almacenado en un archivo, editando los campos de las tramas o añadiendo o eliminando datos.	
<ul style="list-style-type: none">■ Modifica el contenido de los datos de todas las tramas que coinciden con el ID especificado.	
RF - 06	Inyección del tráfico de un archivo csv
El sistema debe permitir al usuario inyectar el tráfico CAN Bus (almacenado en un archivo) en la red, seleccionando el archivo csv correspondiente.	
<ul style="list-style-type: none">■ Inyecta en la red el contenido del archivo csv especificado por el usuario a través de la interfaz.	
RF - 07	Modificación de una trama de un archivo csv
El sistema debe permitir al usuario modificar el tráfico CAN Bus almacenado en un archivo, editando los campos de una trama o añadiendo o eliminando datos.	
<ul style="list-style-type: none">■ Modifica el contenido de los datos de una única trama, seleccionada por el usuario, que coincide con el ID especificado.	
RF - 08	Inyección de una trama en la red
El sistema debe permitir al usuario inyectar una trama CAN Bus especificada en la red.	
<ul style="list-style-type: none">■ Inyecta en la red el contenido de la trama que ha sido creada por el usuario.	

RF - 09	Selección de la interfaz
El sistema debe permitir al usuario seleccionar la interfaz por la que se van a recibir las tramas CAN Bus.	
RF - 10	Filtrado de tramas CAN Bus
El sistema debe permitir al usuario filtrar por las tramas CAN Bus que se desean en el proceso de monitorización.	
<ul style="list-style-type: none"> ▪ Filtra por la cantidad de IDs que el usuario introduzca, mostrando solamente el contenido de los mismos en el proceso de monitorización. 	

4.1.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que especifican cómo debe funcionar un sistema, en lugar de qué debe hacer. Estos requisitos se relacionan con aspectos como la usabilidad, la seguridad, la fiabilidad, el rendimiento o la mantenibilidad del sistema. Los requisitos no funcionales son importantes porque influyen en la satisfacción de los usuarios y en la calidad del producto final.

RNF - 01	Asegurar la integridad de los datos almacenados
El sistema debe asegurar que las tramas almacenadas tanto en el archivo csv como en la base de datos sean los mismos que los que se reciben por la interfaz.	
<ul style="list-style-type: none"> ▪ El sistema debe de almacenar en el archivo csv y en la base de datos las columnas y datos correspondientes, mostrados en la monitorización. 	

RNF - 02	Asegurar la integridad de los datos enviados
El sistema debe asegurar que las tramas enviadas a la red (tanto provenientes de un archivo csv como una única trama) deben de ser las mismas que el usuario ha definido para enviar.	
<ul style="list-style-type: none"> ▪ Correcto funcionamiento del sistema para enviar las tramas definidas por el usuario ▪ Correcto funcionamiento del sistema para enviar correctamente las tramas que están contenidas en un archivo csv. 	

RNF - 03	Accesibilidad
El sistema debe ser accesible para cualquier tipo de usuario, sin importar la experiencia con otras herramientas similares.	
<ul style="list-style-type: none"> ▪ Interfaz amigable para el usuario, contenido bien definido y estructurado. 	

RNF - 04	Rendimiento
El sistema debe ser rápido y eficiente.	
<ul style="list-style-type: none"> ▪ El sistema debe de ser rápido para poder mostrar todas las tramas de datos y eficiente, para poder almacenarlas correctamente. 	

RNF - 05	Actualizaciones y mantenimiento
El sistema debe de garantizar un correcto funcionamiento, actualizaciones para mejorar posibles fallos o incluso la inclusión de nuevas funciones o mejoras.	

4.1.3. Casos de uso

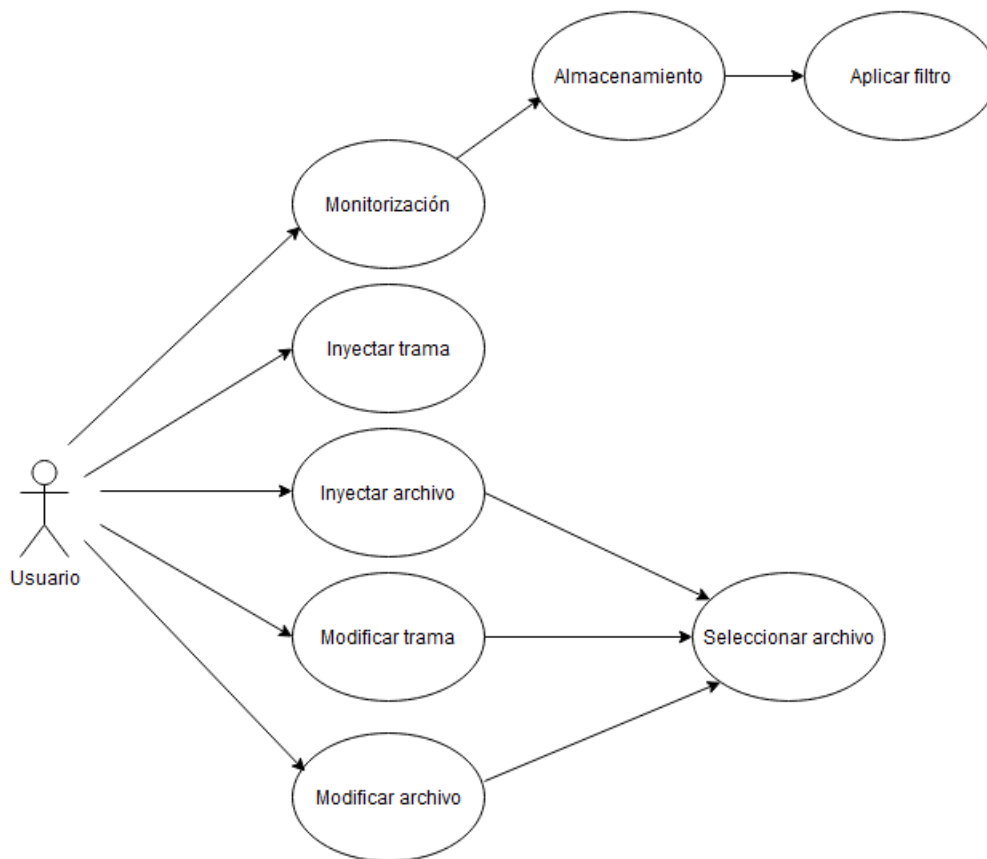


Figura 4.1: Casos de uso

El único actor existente es el usuario que interactúa con la plataforma. Este usuario tiene la posibilidad de seleccionar las diferentes funciones que proporciona la herramienta. En este caso, la interfaz por donde el sistema recibirá las tramas CAN Bus se selecciona por defecto, por lo que esa función no será representada en el diagrama de casos de uso. Figura 4.1.

CAPÍTULO 5

Diseño e implementación

En este capítulo se detalla el desarrollo de la herramienta, utilizando las tecnologías previamente mencionadas en el capítulo 3. La herramienta se divide en tres secciones, dependiendo de las diferentes funcionalidades de la aplicación.

La primera de las funcionalidades es la monitorización, donde el usuario será capaz de monitorizar el tráfico CAN Bus proveniente de una interfaz específica, almacenar el tráfico que se está monitorizando en tiempo real y/o aplicar filtros para mostrar los IDs deseados por el usuario.

La segunda función consiste en la modificación del tráfico que tenemos almacenado. Una vez existen ficheros con los paquetes de la red, el usuario tiene la oportunidad de modificar el campo de datos de esos paquetes para modificar el comportamiento de la red, una vez los inyecte en el sistema.

Una vez modificados los ficheros, se hará uso de la tercera función de la herramienta, la cual consiste en inyectar en la red los paquetes. De esta forma, el usuario puede modificar los paquetes para realizar un comportamiento definido por él e inyectarlos en la red, de forma que el vehículo realizará las acciones definidas por el usuario.

El repositorio donde se encuentra la herramienta está formado por un fichero principal, que corresponde con la aplicación con todas las funcionalidades, y varios ficheros donde se realizan cada una de las funciones descritas. Adicionalmente existe un fichero llamado *procesar.py* para procesar un archivo csv y mostrar solo el identificador y los datos de las tramas de ese archivo.

A continuación, se desarrolla el diseño e implementación de estas funciones en la herramienta.

5.1. Monitorización y almacenamiento

Esta es una de las funciones de mayor importancia de la herramienta ya que, en la mayoría de los casos, es la función que se usará en primer lugar para hacer un análisis de la red CAN Bus del automóvil y detectar y almacenar las tramas para su posterior análisis.

La función de monitorización está dividida en varios ficheros, los cuales son llamados por la herramienta principal, haciendo un sistema modular eficiente. El fichero encargado de la monitorización, desarrollado en Python, tiene como objetivo proporcionar una solución eficiente y versátil para la observación y análisis en tiempo real del tráfico CAN Bus. Para ello, se ha hecho uso de las siguientes bibliotecas:

- **Biblioteca can:** La biblioteca can proporciona la interfaz para la comunicación con CAN Bus. Esta biblioteca es crucial para la comunicación con controladores y dispositivos CAN, ya que permite la transmisión y recepción de mensajes a través del bus. Además, facilita la lectura de mensajes y la configuración de filtros para una captura precisa. Estos son algunos componentes importantes de la biblioteca CAN y cómo se integran en la herramienta de monitorización:
 - Interfaz uniforme: can proporciona una interfaz uniforme para una variedad de tipos de interfaces CAN, como socketcan, kvaser, entre otros. Esto permite que la herramienta sea independiente del hardware subyacente específico, lo que facilita la portabilidad y la adaptabilidad a diferentes entornos.
 - Configuración de filtros: los filtros son algo fundamental para capturar únicamente las tramas relevantes en un entorno con múltiples dispositivos en la red. Los filtros determinan qué tramas se deben capturar en función de su identificador (ID).
 - Manejo de las tramas CAN: facilita la recepción y el envío de mensajes CAN. Para recibir mensajes del bus CAN, la herramienta utiliza la función *bus.recv()*. La biblioteca también permite establecer un tiempo de espera, lo que ayuda a la captura en tiempo real a ser más efectiva.
- **Biblioteca curses:** La biblioteca curses se emplea para la interfaz de usuario en la terminal de Linux. Proporciona un conjunto de funciones que facilitan la creación de interfaces de usuario basadas en texto en terminales. Esta biblioteca es esencial para el desarrollo de la herramienta de monitorización. Permite una presentación interactiva y eficiente de los datos capturados. Estas son algunas de las integraciones de esta biblioteca:
 - Interfaz de texto: permite la creación de la interfaz de usuario, basadas en texto, la cual es especialmente útil para mostrar al usuario las diferentes opciones o detallar toda la información que se desea mostrar de las tramas.
 - Control de teclas y eventos: la biblioteca permite la captura y el manejo de eventos recibidos de teclado. Esto es esencial para la interactividad de la herramienta, permitiendo al usuario realizar acciones como pausar la captura, limpiar la pantalla o salir de la aplicación. La función *getch()* se emplea para leer la entrada del teclado de manera no bloqueante. Además, también se utiliza para establecer los diferentes parámetros previos a la monitorización, como la selección de la interfaz, el almacenamiento o la selección de filtros.
 - Diseño Gráfico en modo texto: proporciona funciones para la manipulación de colores y estilos en modo texto. En la herramienta, se utiliza para resaltar ciertos elementos, como la cabecera, con colores distintivos. Esto mejora la legibilidad y la experiencia visual en la terminal.
 - Actualización dinámica de la pantalla: La capacidad de curses para actualizar

dinámicamente la pantalla es esencial para la visualización en tiempo real de las tramas CAN. La herramienta utiliza ciclos de eventos para recibir mensajes del bus CAN, actualizar la interfaz de usuario y responder a la entrada del usuario sin bloquear la ejecución.

- **Biblioteca struct:** El empaquetado y desempaquetado de datos binarios requiere la biblioteca struct en Python. En el contexto de la función de monitorización de CAN Bus, struct se utiliza para interpretar las tramas CAN, que están representadas como secuencias de bytes. Estos son algunos componentes clave de la biblioteca struct y su función en la implementación:
 - Empaquetado y desempaquetado: la biblioteca proporciona las funciones pack y unpack, que son fundamentales para convertir datos entre su representación en Python y su representación binaria. En la herramienta, se utilizan para empaquetar los datos de las tramas CAN antes de enviarlos al bus y para desempaquetar los datos recibidos del bus.
 - Formatos estandarizados: struct utiliza códigos de formato para especificar cómo se deben interpretar los datos binarios. Por ejemplo, se pueden utilizar formatos como 'I' para representar un entero sin signo de 4 bytes o 'f' para representar un número de punto flotante de 4 bytes. La correcta interpretación de los datos binarios de las tramas CAN se logra mediante el uso de estos códigos de formato.
 - Conversión de datos: facilita la conversión de datos entre su forma natural en Python y su forma cruda en binario. Esto es esencial para interpretar correctamente los campos de las tramas CAN, como el identificador, la longitud del campo de datos y los datos mismos. La herramienta utiliza esta funcionalidad para garantizar la consistencia en la interpretación de los mensajes CAN.

La herramienta principal, llamada *can-analyzer*, ejecuta este fichero añadiendo parámetros para seleccionar la interfaz o los filtros, los cuales son introducidos por el usuario. Además, otra de las elecciones que tiene que realizar el usuario es si desea almacenar el tráfico monitorizado. En caso afirmativo, el sistema llama a *database.py*, fichero donde se crea la base de datos a partir de un fichero csv, con los datos obtenidos de la monitorización. Cuando el usuario selecciona la opción de almacenar los datos monitorizados, la herramienta crea un archivo csv donde almacena toda la monitorización. Posteriormente, se hace una llamada a *database.py*, creando una tabla con el mismo nombre del archivo y almacenando en columnas todos los datos, obtenidos de ese archivo.

Finalmente, el archivo csv creado por la herramienta principal contiene varias columnas con datos sobre las tramas, los mismos que se mostraron en la monitorización. Si el usuario quiere trabajar con ellas para modificarlas y/o enviarlas a la red, solo va a necesitar el identificador y el campo de datos. Para extraer únicamente estos datos se utiliza *procesar.py*, un script en Python donde al introducir como primer argumento el archivo csv con todos los datos de la monitorización, crea otro fichero csv, cuyo nombre es el pasado por el segundo argumento al script, donde solamente almacena el identificador y el campo de datos. De esta forma, con este nuevo archivo podríamos utilizar las dos funciones restantes de la herramienta.

5.2. Modificación

Para el desarrollo de la función de modificación, se han tenido en cuenta dos posibilidades distintas. La primera de ellas es que el usuario quiera modificar una única trama y la otra opción sería que el usuario quiera modificar más de una trama.

Supongamos que el usuario detecta el mensaje el cual hace que las puertas del coche estén bloqueadas y, en lugar de cerrarlas, el usuario quiera modificar los datos para desbloquearlas. Por ejemplo, supongamos que el paquete con identificador 139 y el campo de datos 00 00 00 realiza la función de bloquear las puertas y la trama con identificador 139 y datos 00 00 01 desbloquea las puertas. En este caso, se ha creado una función capaz de modificar una única trama, llamada *modificar trama*. En esta función debes de especificar el identificador junto con el campo de datos que desees para que se realice la modificación. Además, es probable que el usuario quiera modificar una trama en concreto que aparece en una posición determinada. Supongamos que para el ejemplo anterior, el identificador 139 aparece dos veces, una para bloquear y otra para desbloquear, pero solamente quiero modificar la de bloqueo para que siempre quede desbloqueado. En este caso, se ha implementado en la función un parámetro que debe de introducir el usuario indicando qué aparición desea modificar. Para el caso de este ejemplo, el usuario quiere modificar la segunda trama (que corresponde al bloqueo) por lo que debería de introducir un 2. Así logramos una modificación exacta del fichero que deseamos modificar de manera eficiente.

Sin embargo, es posible que el usuario quiera modificar todas las tramas que tengan un identificador específico. Por ejemplo, supongamos que el usuario tiene ha detectado una una trama que cuando su valor es 0A hace que el vehículo acelere y cuando es 00 deja de acelerar. Ahora, el usuario quiere modificar todas las tramas para que siempre esté acelerando, es decir, quiere modificar todos los mensajes de un identificador específico que contengan 00 para que pasen a valer 0A. Para este caso se ha creado la función *modificar archivo*, donde el usuario introduce un identificador y un campo de datos y la herramienta modifica todas las coincidencias de ese id con los datos que ha introducido el usuario.

Con estas dos funciones el usuario puede modificar de forma eficiente y rápida el fichero con el contenido de las tramas. La implementación de esta función es bastante sencilla, ya que hace uso de la biblioteca *csv*, proporcionando una interfaz efectiva para la manipulación de archivos *csv*. Las funciones *csv.reader* y *csv.writer* simplifican la lectura y escritura de datos CSV, haciendo mucho mas fácil la modificación.

5.3. Inyección

La función de inyección consiste en enviar a la red CAN Bus, a través de una interfaz seleccionada, una o varias tramas para que la red las reconozca como si se hubiesen generado dentro de esa red y las ejecute. En la realización de esta función también se diferenciaron dos posibles casos, donde el usuario quiera inyectar una única trama o donde el usuario quiera inyectar un fichero que contiene varias tramas.

Para tratar estas dos funciones se han creado dos ficheros. El fichero *cansend.py* el cual se utiliza para enviar una única trama y el fichero *canplayer.py* para enviar un archivo. Ambos scripts pueden ser ejecutados sin necesidad de ejecutar la herramienta principal, ya que se contempla el caso en el que el usuario quiera utilizar el script directamente desde la terminal, en lugar de ejecutar la herramienta principal mostrando la interfaz con curses.

La implementación de ambos scripts es similar, a diferencia de inyectar archivo donde se hace uso de la biblioteca csv para leer el archivo. A continuación, se realiza un desarrollo más detallado de las bibliotecas:

- **Biblioteca sys:** la biblioteca sys proporciona acceso a algunas variables utilizadas o mantenidas por el intérprete y funciones que interactúan con el intérprete. En este caso, se utiliza para obtener argumentos de la línea de comandos.
- **Biblioteca socket:** la biblioteca socket proporciona funcionalidades de red de bajo nivel. En este caso, se utiliza para crear y manipular un socket CAN en crudo (*socket.PF_CAN*, *socket.SOCK_RAW*, *socket.CAN_RAW*). También se utiliza para enviar el paquete CAN a través del socket.
- **Biblioteca struct:** en este caso, la biblioteca struct se utiliza para empacar el identificador CAN y los datos en un formato específico antes de enviarlos a través del socket.
- **Biblioteca can:** como se ha explicado previamente, la biblioteca can proporciona una interfaz de alto nivel para interactuar con controladores de red CAN. En este caso, se utiliza para convertir el identificador CAN y los datos a formatos apropiados.
- **Biblioteca csv:** la biblioteca csv se utiliza para leer el archivo csv que contiene las tramas CAN a enviar.

Finalmente, se han añadido todas estas funciones a la la herramienta principal para, así, crear una aplicación con la cual puedas realizar cualquiera de las acciones necesarias cuando tratamos de interactuar con una red que hace uso del protocolo CAN Bus. Además, en esta aplicación se han cuidado los detalles visuales, utilizando Python curses, de manera que sea mucho más fácil para el usuario a la hora de interactuar y procesar la información. Para el caso de la inyección y la modificación se ha creado una tabla donde el usuario introduce separadamente el identificador y el campo de datos. Sin embargo, para el apartado de monitorización se ha ordenado la información en columnas de manera ordenada para que el usuario sea capaz de visualizar como cambian los datos de las tramas con un mismo identificador.

CAPÍTULO 6

Conclusiones y líneas futuras

6.1. Conclusiones

La culminación de este proyecto representa un hito significativo en el ámbito de las herramientas para la manipulación y monitorización del Controller Area Network (CAN Bus). La creación de esta herramienta avanzada no solo ha sido un ejercicio técnico, sino un compromiso con la mejora y simplificación de las operaciones relacionadas con las redes CAN Bus.

A través del diseño y desarrollo de esta herramienta, se ha logrado proporcionar a los usuarios una solución integral y fácil de usar para monitorizar el tráfico, inyectar archivos y tramas, así como modificar datos en tiempo real. La interfaz de usuario intuitiva y las funcionalidades robustas de la herramienta buscan reducir las barreras de entrada a los usuarios que se inicien en la seguridad en automóviles, permitiéndoles explorar y comprender mejor las complejidades de las redes CAN Bus.

La implementación de tecnologías modernas, buenas prácticas de desarrollo y la atención cuidadosa a la seguridad han sido pilares fundamentales en la construcción de esta herramienta. Su versatilidad y capacidad para adaptarse a diferentes escenarios y requisitos la posicionan como una herramienta valiosa en entornos de desarrollo, diagnóstico y pruebas.

Este proyecto también destaca la importancia de la colaboración y el intercambio de conocimientos en comunidades tecnológicas. El código abierto y la documentación detallada permiten que esta herramienta evolucione y mejore con la retroalimentación y contribuciones de la comunidad.

En última instancia, la herramienta desarrollada tiene el potencial de contribuir significativamente al avance y comprensión de las redes CAN Bus, brindando a los usuarios una herramienta poderosa y accesible para interactuar con estos sistemas cruciales en una variedad de aplicaciones.

6.2. Futuras líneas de trabajo

Durante el desarrollo de este proyecto se han contemplado múltiples formas de implementar la herramienta y nuevas posibles implementaciones para futuras versiones de la aplicación. En este momento la herramienta es completamente funcional, no obstante, a continuación se procede a desarrollar algunas de las ideas más relevantes que se han tenido para futuras versiones.

6.2.1. Mejora visual

Como ya se ha mencionado anteriormente, la herramienta se ejecuta en la terminal mostrando una interfaz desarrollada haciendo uso de Python curses. Hacer uso de curses ha sido bastante intuitivo y una muy buena forma de desarrollar la primera versión de esta aplicación. Sin embargo, sería necesario investigar mucho más a fondo otras tecnologías que permitan crear de una manera más vistosa algunas de las diferentes pantallas que se muestran, por ejemplo las tablas. Esta parte cobra bastante importancia, ya que al mostrar una gran cantidad de datos es muy importante que sea vistoso para el usuario, que se muestre lo más claro posible haciendo uso de colores, tablas, etc.

Por este motivo sería importante investigar un poco más a fondo en este tema para mejorar la experiencia del usuario y hacer más ágil el visionado de ciertas funcionalidades como la monitorización o la modificación/inyección de tramas o archivos.

6.2.2. Opción para mostrar diferencias

Cuando iniciamos la monitorización de una red CAN Bus se observan una gran cantidad de tramas que están circulando por la red. Aunque el diseño de esta herramienta hace que se pueda diferenciar de manera bastante sencilla los diferentes tipos de IDs de las diferentes tramas, es difícil ver a simple vista qué cambios existen entre las tramas que se reciben. Especialmente cuando decidimos conectar la herramienta a un coche moderno donde hay muchos más mensajes en la red. Frente a este problema una solución interesante sería añadir una función la cual indique con diferentes colores en el apartado de datos cuales son los valores que se están modificando entre la trama actual y la trama anterior recibida. Por ejemplo, supongamos que al bajar la ventanilla la trama que circula tiene ID 041 con un valor 00 00 0A 00 y al subirla, la trama con ID 041 tiene el valor 00 00 0B 00. Si mientras se está monitorizando la red la herramienta muestra que en la posición donde se encuentra “0B” tiene un color específico, resulta mucho más sencillo detectar que trama se está modificando mientras se realiza la acción, siendo posible identificar que su id es 041 y qué valor modificar para realizar un comportamiento diferente.

Esta nueva funcionalidad también podría implementarse al almacenar los datos y buscar la manera de representarlo en la base de datos, ya que incluso sería más útil a la hora de analizar los mensajes almacenados y poder detectar qué tramas se están modificando. Para desarrollar esta funcionalidad sería necesario más investigación en esa parte en concreto.

6.2.3. Compatibilidad con Windows

Aunque el objetivo principal de esta herramienta no es que sea compatible para el sistema operativo Windows, podría ser interesante desarrollar una versión compatible para aquellos casos en los que el usuario no tenga acceso a una terminal Linux. El enfoque para realizarlo sería diferente ya que, en este caso, podría ser más conveniente que la herramienta tenga una interfaz gráfica en lugar de ejecutarse desde la línea de comandos, por lo que sería necesario volver a modelar la parte visual de la aplicación.

Esta funcionalidad haría que la herramienta fuese mucho más competente, ya que algunas de las herramientas que se utilizan en la actualidad son compatibles en ambas plataformas. Para el desarrollo de esta característica sería necesario invertir tiempo en la investigación de la compatibilidad para Windows, así como las diferentes formas de desarrollar la interfaz gráfica.

Bibliografía

1. M. Bozdal, M. Samie, S. Aslam e I. Jennions, Evaluation of CAN Bus Security Challenges. *Sensors* **20**, n^o8 (2020).
2. M. Markovitz y A. Wool, Field classification, modeling and anomaly detection in unknown CAN bus networks. *School of Electrical Engineering, Tel Aviv University, Israel* (2017).
3. C. Smith, *The car hacker's handbook: A guide for the penetration tester (2.a ed.)*
4. D. Santo Orcero, *La Biblia en L^AT_EX*.
5. *GitHub*, (www.github.com).
6. *Medium*, (www.medium.com).
7. *Python Package Index*, (www.pypi.org).
8. *Fundamentos CAN Bus*, (es.scribd.com/doc/162902095/CAN-Bus-Fundamentos).
9. *Wikipedia*, (es.wikipedia.org).
10. *Ventajas, componentes y fallas en el sistema CAN Bus*, (blog.reparacion-vehiculos.es/can-bus-ventajas-fallas-en-el-sistema).
11. *Qué es y qué beneficios tiene el protocolo CAN Bus*, (www.tadig.es/blog/protocolo-can-bus-flotas-de-transporte/).
12. *Protocolo CAN Bus*, (www.redesinformaticas.org).
13. *CAN protocol*, (www.sterlingmedicaldevices.com/our-work/medical-device-projects/can-protocols/).
14. *CAN Bus Applications*, (www.copperhilltech.com/blog/can-bus-tutorial-typical-can-bus-applications/).
15. *Python curses application*, (www.developer.com/languages/python/python-curses-library-example/).
16. *SocketCAN - Controller Area Network*, (www.kernel.org/doc/html/latest/networking/can.html).

Apéndice

APÉNDICE A

Manual de usuario

En esta sección se explica cómo usar la herramienta para aprender todas las funcionalidades implementadas. Es importante destacar que, para este punto, se supone que se han seguido los pasos sobre la configuración del entorno en el capítulo 3, ya que para demostrar el uso de esta herramienta se utiliza *Instrument Cluster Simulator* para simular los paquetes del protocolo CAN Bus.

Una vez tenemos ICSim configurado correctamente, vamos a descargar la herramienta a través del repositorio de GitHub con el siguiente comando:

```
git clone https://github.com/javiki57/can-analyzer
```

Una vez clonado el repositorio aparecerá una carpeta llamada “can-analyzer”. Esta carpeta contiene varios ficheros entre ellos la herramienta principal, que la ejecutaremos con el comando:

```
python3 can-analyzer.py
```

Esto nos mostrará el siguiente menú A.1, donde el usuario deberá de seleccionar una de las opciones pulsando la tecla enter. Si el usuario desea salir de la aplicación, puede seleccionar la opción de salir o pulsar la tecla escape. En general, cuando vamos avanzando por las diferentes opciones, la tecla escape siempre servirá para volver un paso atrás, es decir, a la pantalla anterior a donde nos encontrábamos.

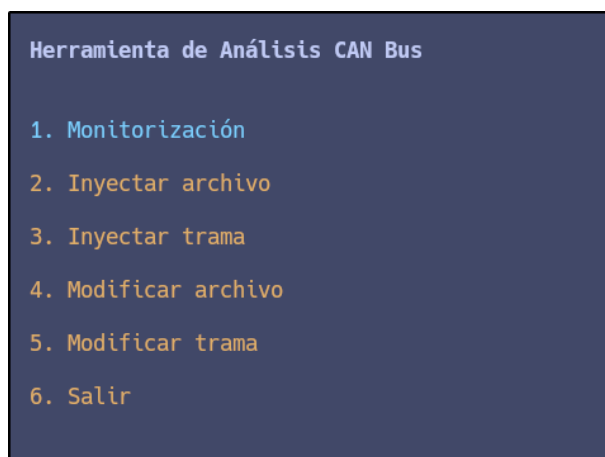


Figura A.1: Menú principal de can-analyzer

A.1. Monitorización

Cuando seleccionamos la opción de “Monitorización” para monitorizar el tráfico que circula en la red CAN Bus, la herramienta nos pregunta si deseamos almacenar el tráfico que vamos a monitorizar para que, de esta forma, el contenido se almacene tanto en un archivo csv como en la base de datos. La aplicación nos mostrará una pantalla de la siguiente forma A.2, donde el usuario debe seleccionar la opción de sí o no, utilizando las flechas de arriba y abajo para el desplazamiento entre opciones y pulsando la tecla *enter* para confirmar la selección.

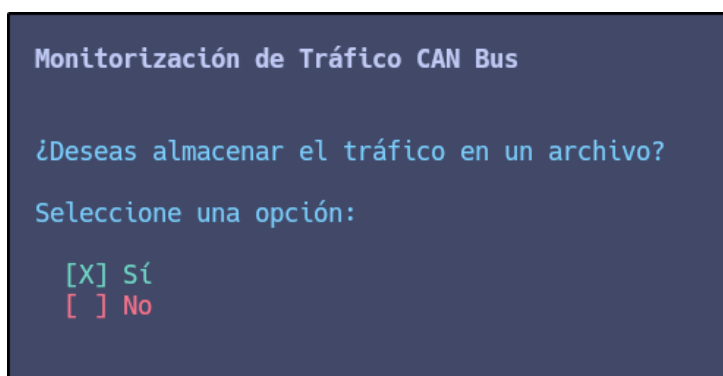


Figura A.2: Menú para seleccionar si se desea o no almacenar el tráfico de la monitorización.

A continuación, independientemente de si el usuario desea o no almacenar el contenido, el sistema lanzará una nueva pregunta en otra pantalla preguntando por la interfaz que el usuario desea usar, es decir, el nombre de la interfaz por donde se van a recibir las tramas CAN Bus. Para el caso de este proyecto, al estar haciendo uso de *ICSim* la interfaz siempre será *vcan0*, por lo que la selección por defecto (el usuario no introduce nada) será esta interfaz. Otro de los motivos por los cuales se ha seleccionado esta interfaz como *default* es con el pensamiento de que la mayor parte de los usuarios que hagan uso de esta herramienta por primera vez, es altamente probable que la ejecuten en un simulador como el que se está haciendo uso en este momento. No obstante, el usuario debe de poder escribir el nombre de la interfaz desde teclado. La pantalla se muestra como en la figura A.3. El usuario debe pulsar la tecla *enter* para continuar con la siguiente pantalla.

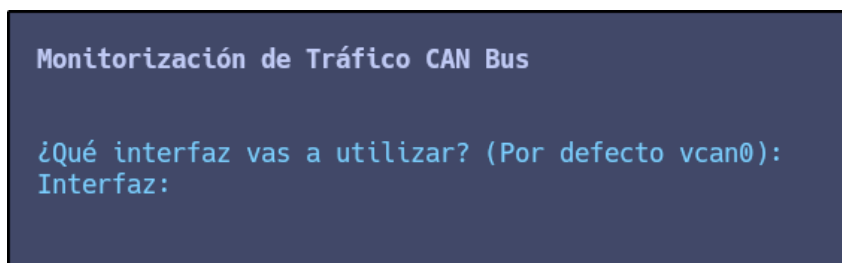


Figura A.3: Menú para seleccionar la interfaz para recibir las tramas de la red CAN Bus.

En este punto la aplicación mostrará opciones diferentes, dependiendo de si el usuario ha seleccionado o no el almacenamiento de datos. Si la selección fue “Sí”, el sistema

preguntará al usuario qué nombre desea para el archivo csv (Figura A.4), donde se almacenarán todos los datos. Además, el nombre seleccionado para el archivo coincidirá con el nombre de la tabla donde se almacenarán los datos en la base de datos. En este apartado, el usuario deberá de introducir solamente el nombre del archivo, sin escribir la extensión csv, ya que el programa la añade automáticamente.

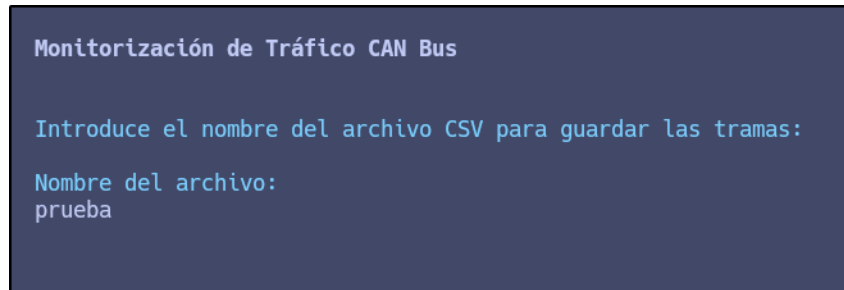


Figura A.4: Menú para seleccionar el nombre del archivo donde almacenar las tramas de la red CAN Bus.

Si la selección fue “No.” en el almacenamiento, el sistema no mostrará la pantalla anterior (figura A.4) y pasará a la siguiente (figura A.5), donde el usuario debe de seleccionar si desea o no filtrar por identificadores. Esto quiere decir que, si se desea filtrar, solo se mostrarán en la monitorización los datos sobre las tramas cuyos IDs haya decidido el usuario. En el caso de no querer aplicar filtros, la aplicación ejecutará el script de monitorización y mostrará todo lo que está pasando por la red.

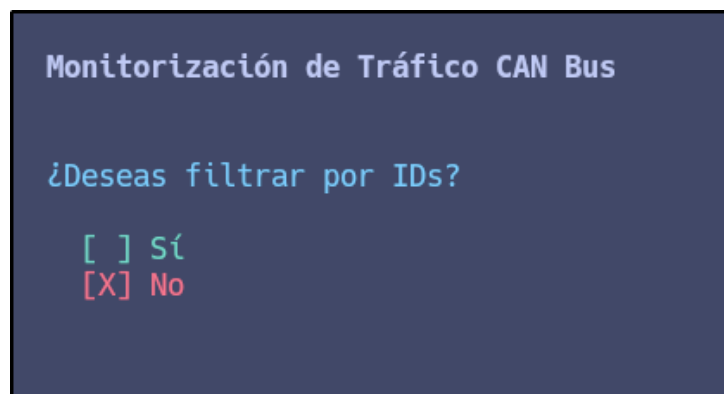


Figura A.5: El usuario debe seleccionar si desea o no filtrar el tráfico de la red CAN Bus.

Cuando no se desea filtrar, el sistema mostrará todo el contenido de la red, como en la figura A.8. Una vez estamos en esta pantalla es muy importante tener en cuenta los diferentes controles que se muestran a continuación, para evitar errores y poder hacer un buen uso de la herramienta:

- Pausar y reanudar la monitorización: como ya sabemos, en una red CAN Bus circulan una enorme cantidad de tramas al mismo tiempo, dificultando el visionado de los datos en tiempo real. En esta herramienta, pulsando el tecla espacio podemos pausar la monitorización. Esto es muy útil cuando el usuario quiere observar algún dato en un momento concreto de la monitorización, o bien si deseas almacenar el

contenido de una acción que vas a realizar y no quieres almacenar tramas “basura” hasta que realices esa acción. Para volver a reanudar, simplemente pulsando de nuevo la tecla espacio se volverá a reanudar la monitorización.

Conteo	Tiempo	Frecuencia	ID	Tam	Datos	Función	ID Nodo
282	4.278630	0.015979	039	2	00 39		
428	4.275625	0.009358	095	8	80 00 07 F4 00 00 00 35	EMCY	21
428	4.280161	0.008186	133	5	00 00 00 00 98		
428	4.280162	0.008172	136	8	00 02 00 00 00 00 00 1B		
428	4.281632	0.009640	13A	8	00 00 00 00 00 00 00 19		
428	4.281636	0.009642	13F	8	00 00 00 05 00 00 00 1F		
427	4.275620	0.011498	143	4	6B 6B 00 C2		
428	4.280154	0.010528	158	8	00 00 00 00 00 00 00 0A		
428	4.280158	0.010517	161	8	00 00 05 50 01 08 00 0D		
428	4.281637	0.009642	164	8	00 00 C0 1A A8 00 00 31		
428	4.278632	0.010114	166	4	D0 32 00 09		
428	4.281639	0.009642	17C	8	00 00 00 00 10 00 00 12		
427	4.273101	0.008993	183	8	00 00 00 0A 00 00 10 01	TPD01	3
9	4.178417	0.505296	188	4	00 00 00 00	TPD01	8
428	4.281641	0.009642	18E	3	00 00 5C	TPD01	14
428	4.280159	0.010516	191	7	01 00 90 A1 41 00 30	TPD01	17
214	4.278619	0.021916	1A4	8	00 00 00 08 00 00 00 01	TPD01	36
213	4.278623	0.021918	1AA	8	7F FF 00 00 00 00 68 01	TPD01	42
213	4.278624	0.021919	1B0	7	00 0F 00 00 00 01 48	TPD01	48
214	4.272000	0.015311	1CF	6	80 05 00 00 00 2D	TPD01	79
213	4.278627	0.021920	1D0	8	00 00 00 00 00 00 00 0A	TPD01	80
214	4.272003	0.015312	1DC	4	02 00 00 2A	TPD01	92
106	4.262648	0.036597	21E	7	03 E8 37 45 22 06 3E	RPD01	30
323	4.278645	0.010932	244	5	00 00 00 01 F3	RPD01	68
106	4.262645	0.036595	294	8	04 0B 00 02 CF 5A 00 3B	TPD02	20
41	4.226053	0.106746	305	2	80 17	RPD02	5
44	4.281642	0.092100	309	8	00 00 00 00 00 00 00 A2	RPD02	9
43	4.256693	0.103787	320	3	00 00 12	RPD02	32
43	4.256694	0.103786	324	8	74 65 00 00 00 00 0E 1A	RPD02	36
42	4.236569	0.101795	333	7	00 00 00 00 00 00 1E	RPD02	51
43	4.256695	0.103786	37C	8	FD 00 FD 00 09 7F 00 1A	RPD02	124
14	4.086574	0.302939	405	8	00 00 04 00 00 00 00 29	RPD03	5
14	4.053526	0.300042	40C	8	00 00 00 00 04 00 00 13	RPD03	12
14	4.086572	0.302939	428	7	01 04 00 00 52 1C 2F	RPD03	40
14	4.053533	0.300048	454	3	23 EF 18	RPD03	84
4	3.705552	0.997758	5A1	8	96 00 00 00 00 00 62 2F	SDO_TX	33

Figura A.6: Ejemplo de monitorización del tráfico de la red CAN Bus.

- Limpiar el contenido: cuando el usuario pausa la monitorización, realmente está pausando lo que se estaba mostrando en la pantalla pero algunas variables, como el tiempo, siguen avanzando a pesar de la pausa. Para solucionar esto, si el usuario pulsa la tecla “c”, se limpiará todo el contenido de la pantalla y se volverán a inicializar los valores de todas las columnas, de forma que al volver a pulsar la tecla espacio para reanudar la ejecución será como iniciar de nuevo la monitorización. Esto es muy útil para realizar acciones como grabar las tramas de una acción en concreto en un momento concreto sin almacenar las tramas no deseadas. De esta forma sería mucho más sencillo identificar las tramas que el usuario está buscando ya que la cantidad de datos sería menor.
- Desplazamiento: el usuario puede usar las teclas de flecha hacia abajo o hacia arriba para desplazar el contenido cuando realizamos una pausa. Esta funcionalidad resulta muy útil cuando no podemos ver todos los datos en la terminal, ya sea porque el tamaño de pantalla es pequeño o porque existen muchos datos diferentes.
- Terminar la aplicación: existen varias formas de cerrar la aplicación, es muy importante cerrar correctamente la herramienta de monitorización ya que esto podría

provocar la pérdida de los datos a la hora de almacenarlos. Para cerrar correctamente la herramienta sin ningún fallo, podemos presionar la tecla “q”. Esto hará que se cierre por completo la herramienta y, en el caso de que el usuario escogiese almacenar la información, se almacene todo de manera correcta. Otra opción válida sería pulsando la tecla escape. Es muy importante no forzar el cierre de la aplicación pulsando “ctrl+c”, ya que esto provocaría que la herramienta no almacene los datos que ha monitorizado, en el caso de haber seleccionado la opción para ello.

Cuando el usuario desea aplicar filtros, se mostrará una nueva pantalla (Figura ??) donde el usuario debe de introducir los identificadores que desea visualizar en la monitorización. Es importante que, al introducir los identificadores, estos estén separados por comas, sin espacios.

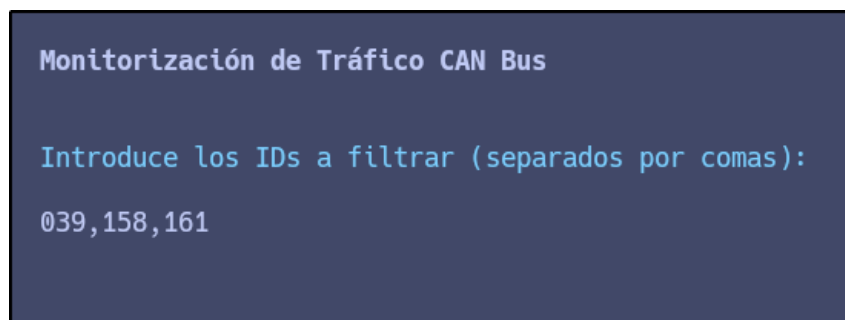


Figura A.7: Ejemplo de filtros seleccionados.

Una vez introducidos, al pulsar la tecla *enter*, se mostrará una nueva pantalla con los datos sobre únicamente las tramas deseadas por el usuario. De la misma forma, si el usuario desea almacenar las tramas, se almacenarán únicamente las tramas que el usuario ha elegido.

Conteo	Tiempo	Frecuencia	ID	Tam	Datos	Función	ID	Nodo
233	3.530312	0.016002	039	2	00 2A			
354	3.531422	0.009978	158	8	00 00 00 00 00 00 00 0A			
354	3.531434	0.009989	161	8	00 00 05 50 01 08 00 0D			

Figura A.8: Ejemplo de monitorización con los filtros seleccionados.

En este caso, podemos realizar las mismas funcionalidades en la pantalla de monitorización que si no aplicamos filtros. Además, es muy importante cerrar bien la aplicación para que los datos puedan ser almacenados correctamente, en el caso de seleccionarlo previamente.

En cualquiera de los casos, al terminar la ejecución de la monitorización, si se ha seleccionado el almacenamiento de los datos se creará un archivo con el nombre elegido en formato csv y una base de datos. Se pueden usar múltiples herramientas para visualizar la base de datos, en este caso se ha utilizado *sqlitebrowser*. Con esta aplicación podremos ver la base de datos con una tabla, cuyo nombre es el mismo que el archivo csv, la cual contiene varias columnas con los datos sobre el tráfico. En este caso también podremos filtrar o realizar alguna búsqueda sobre los datos que hemos almacenado.

Finalmente, existe un script llamado “procesar.py” para convertir el archivo csv que ha sido creado por la herramienta, con el nombre seleccionado por el usuario, en otro archivo csv. Este nuevo archivo csv tendrá el tráfico que tenía el archivo creado por la aplicación pero almacenando únicamente el identificador y el campo de datos. De esta forma, creamos un archivo nuevo con los datos más relevantes de la captura, los cuales nos sirven para después modificar o inyectar el contenido. Es muy importante tener en cuenta esto, ya que la herramienta solo aceptará este nuevo formato a la hora de modificar o inyectar el contenido de un archivo. La manera de ejecutar este script sería la siguiente:

```
python3 procesar.py <archivo_monitorizacion> <nuevo_nombre>
```

El argumento `archivo_monitorización` hace referencia al nombre del archivo que hemos seleccionado al seleccionar que deseamos almacenar la monitorización, y `nombre_nuevo` hace referencia al nombre del archivo que el usuario desea que tenga el nuevo archivo, el cual va a contener el formato “ID#Datos”.

A.2. Modificar

Para que la función modificar funcione correctamente, el archivo csv debe de contener únicamente las tramas en el formato “ID#Datos”, de lo contrario se mostrará un error y no modificará el contenido. A continuación, se detalla cómo utilizar las funciones modificar archivo y modificar trama.

A.2.1. Modificar archivo

Esta función modifica las tramas de un archivo que coinciden con el identificador que el usuario introduce. Para ello, seleccionamos la cuarta opción, modificar archivo. Una vez seleccionada, se mostrará una nueva pantalla donde el usuario debe de introducir la ruta absoluta del archivo que se desea modificar. Para facilitar esto, la herramienta te muestra en color morado la ruta actual desde donde se ha ejecutado la herramienta, ya que lo más común es que el archivo que se desea modificar se encuentre en la misma ruta donde se ha ejecutado la herramienta, o similar. En la figura A.9 se muestra un ejemplo de la ruta introducida por el usuario para seleccionar el archivo prueba.csv.

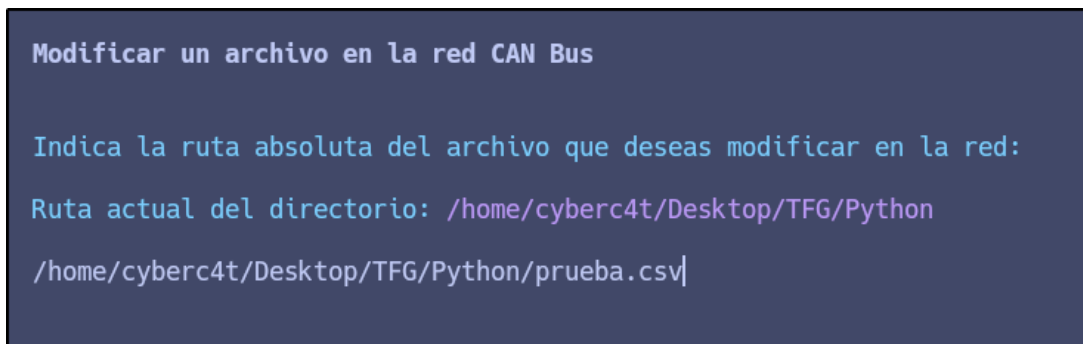


Figura A.9: Ejemplo sobre cómo introducir la ruta de un archivo para modificar.

Una vez introducida la ruta, se mostrará una nueva pantalla con una tabla donde el usuario deberá de introducir el identificador y los datos, en ese mismo orden. La intro-

ducción de los datos se divide en dos, en primer lugar se introduce el ID de la trama que se desea modificar y, posteriormente, se debe de pulsar la tecla tabulador para cambiar al campo de datos. Una vez pulsado el tabulador, el usuario podrá escribir en la parte de la tabla correspondiente a los datos. Finalmente, una vez introducidos todos los datos, pulsando la tecla enter la herramienta ejecutará la modificación en el archivo seleccionado. Si el identificador no coincide con ningún ID dentro del archivo no se realizará ninguna modificación, por lo que es importante asegurar que los datos introducidos son correctos. Por ejemplo, si queremos modificar todas las tramas cuyo ID es 19B, debería de mostrarse como en la figura A.10.

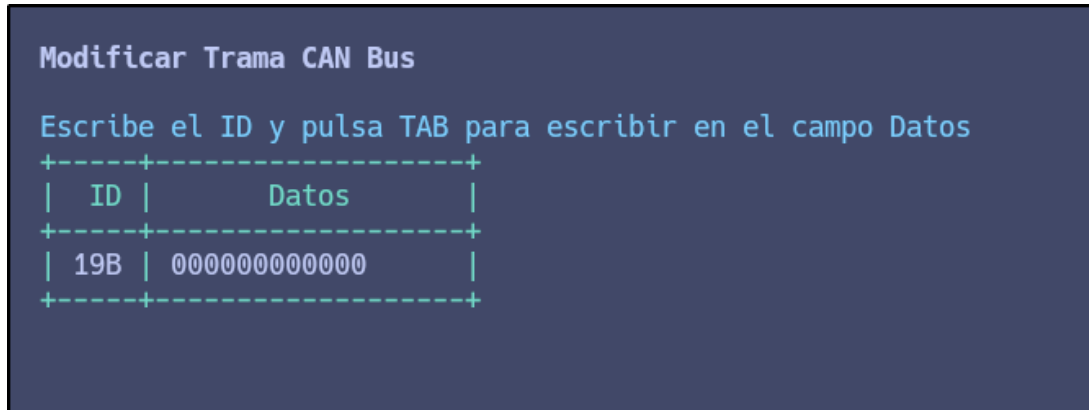


Figura A.10: Ejemplo sobre cómo introducir los datos de un archivo para modificar.

A.2.2. Modificar trama

Esta función es muy similar a modificar archivo, a diferencia de que esta solamente modifica una trama seleccionada por el usuario, mientras que modificar archivo modifica todas las tramas que coinciden con el identificador.

La primera pantalla funciona igual que en modificar archivo, el usuario debe de introducir la ruta absoluta del archivo que desea modificar, como se muestra en la figura A.10. Una vez introducido, el usuario deberá de introducir el identificador y el campo de datos siguiendo las directrices anteriormente explicadas, es decir, pulsando la tecla tabulador para cambiar entre el campo de identificador y el campo de datos. Finalmente, el usuario debe pulsar la tecla enter para confirmar y pasar a la siguiente pantalla.

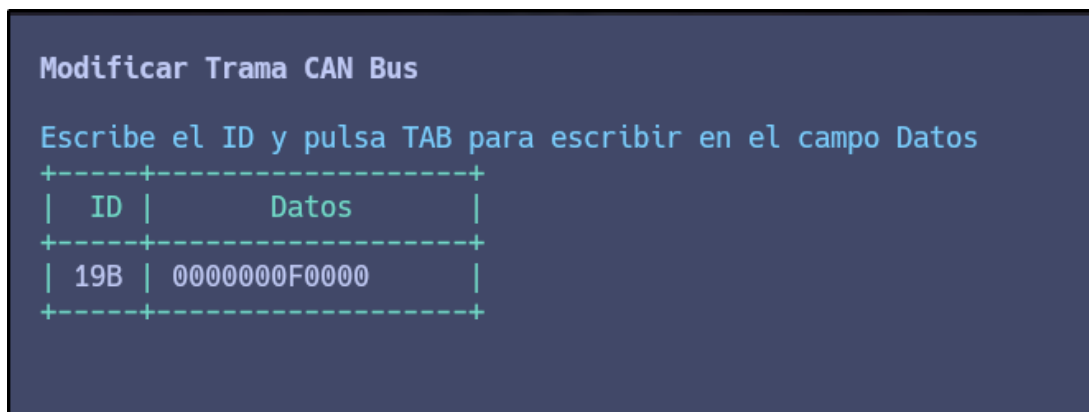


Figura A.11: Ejemplo sobre cómo introducir los datos de un archivo para modificar.

En este caso solamente se va a modificar una única trama. Para ello, el sistema muestra una pantalla donde el usuario debe de introducir un valor numérico indicando qué ocurrencia del identificador que se ha introducido se desea modificar. Por ejemplo, supongamos que el archivo contiene varias tramas con identificador 19B y nosotros queremos modificar solamente la segunda vez que aparece esa trama. Para ese caso, el usuario debe de introducir el número dos. En el caso de solo existir una, es posible introducir un uno o no introducir nada y el sistema modificará esa única ocurrencia. A continuación se muestra un ejemplo, en la figura A.12.

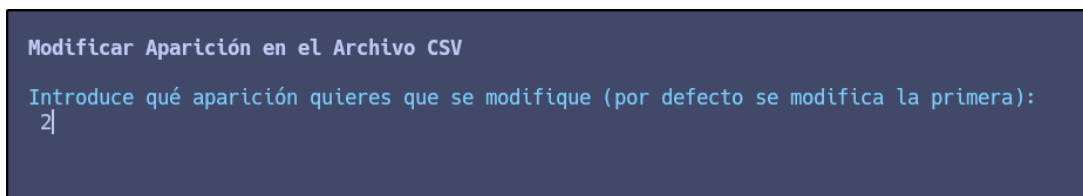


Figura A.12: Ejemplo sobre cómo introducir que trama modificar del archivo seleccionado.

A.3. Inyectar

A.3.1. Inyectar archivo

Esta función permite al usuario inyectar un archivo en la red CAN Bus, es decir, inyectará un conjunto de tramas, las cuales están almacenadas dentro de un archivo csv. Para que la función inyectar archivo funcione correctamente, el archivo csv debe de contener únicamente las tramas en el formato “ID#Datos“, de lo contrario se mostrará un error y no inyectará las tramas en la red.

En primer lugar, el usuario debe de introducir la ruta absoluta donde se encuentra el archivo que se desea inyectar. Para ello, se mostrará la siguiente pantalla A.13 donde en color morado se indica la ruta actual donde el usuario ha ejecutado la herramienta. En este caso es importante añadir la extensión csv al archivo para que el sistema sea capaz de reconocerlo correctamente.

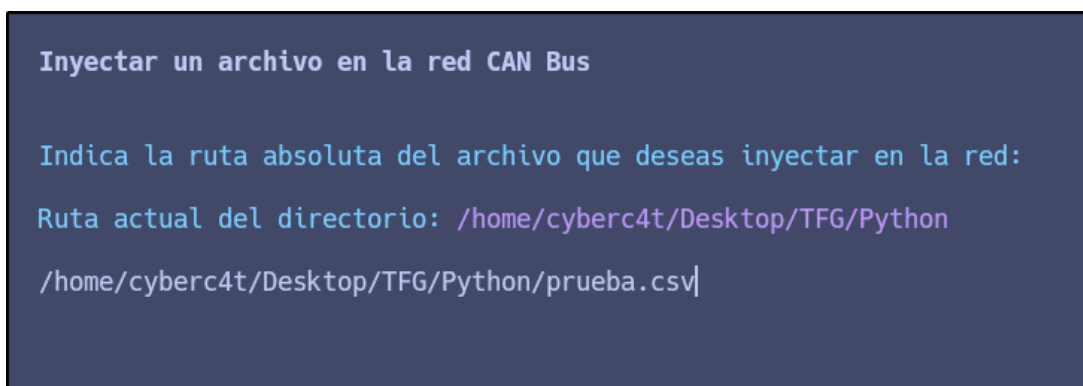


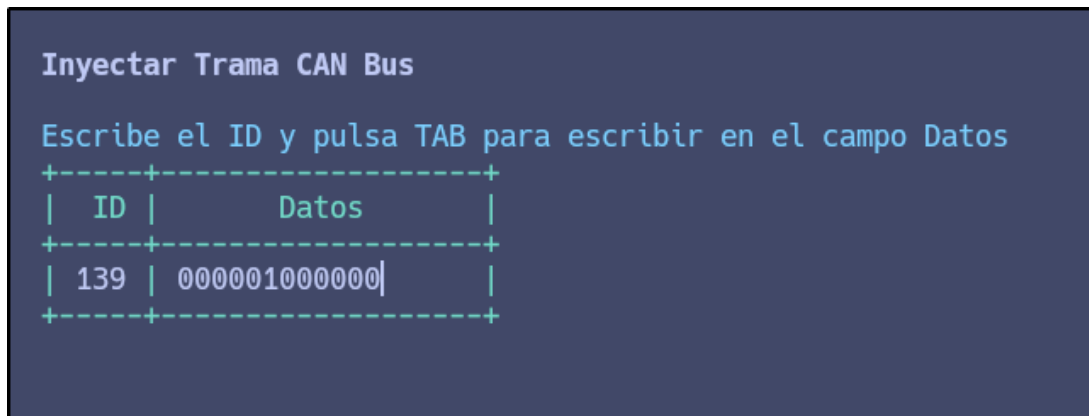
Figura A.13: Ejemplo sobre cómo introducir la ruta del archivo a inyectar.

Una vez seleccionado el archivo, se mostrará una nueva pantalla A.3 donde el usuario deberá de introducir la interfaz por la cual se inyectarán las tramas. En el caso de hacer

uso del simulador, la interfaz será vcan0 por lo que el usuario puede simplemente pulsar enter sin introducir texto, ya que es la interfaz por defecto. En este momento, se ejecutará el script que envía las tramas a la red a través de la interfaz y las tramas serán inyectadas, por lo que se podrá observar el comportamiento de las mismas en el simulador o vehículo.

A.3.2. Inyectar trama

Esta función permite inyectar una única trama en la red CAN Bus. Para ello, el usuario debe de seleccionar primeramente la interfaz, al igual que en la función inyectar archivo. Una vez seleccionada, el sistema mostrará una nueva pantalla, como se muestra en la figura A.14, donde el usuario deberá introducir el identificador y el campo de datos. Es muy importante usar la tecla *TAB* o tabulador para cambiar entre el campo ID y el campo de datos, es decir, una vez escrito el identificador, el usuario debe de pulsar la tecla tabulador y, posteriormente, escribir los datos que desea inyectar. Una vez pulsado el botón enter, la trama se enviará a la red y podremos ver en el simulador o vehículo el resultado de la trama que hemos inyectado.



```

Inyectar Trama CAN Bus

Escribe el ID y pulsa TAB para escribir en el campo Datos
+-----+-----+
| ID |      Datos      |
+-----+-----+
| 139 | 000001000000|
+-----+-----+
  
```

Figura A.14: Ejemplo sobre cómo introducir los datos de una trama a inyectar.



UNIVERSIDAD
DE MÁLAGA

| **uma.es**

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga