“A mis padres por ser el pilar fundamental en todo lo que soy, en toda mi educación, tanto académica, como de la vida, por su incondicional apoyo perfectamente mantenido a través del tiempo.

Todo este trabajo ha sido posible gracias a ellos.”

Facundo

“Dedicado a mis padres Julio y Cristina, maestros y formadores de mi vida, y mi hermano Federico, por el esfuerzo y dedicación siendo mi sostén a través del camino recorrido.

Este cierre de etapa es gracias a ustedes.”

Gabriel

Agradecimientos

A nuestras familias, las cuales nos brindaron la posibilidad de acceder a una carrera universitaria y poder concluirla.

Al Instituto Universitario Aeronáutico, el cual nos abrió las puertas de sus laboratorios, permitiéndonos utilizar hardware y equipos para poder llevar a cabo este desarrollo.

A todo el grupo docente del Instituto de distintas carreras, de los cuales siempre tuvimos una respuesta ante ayuda y consultas, en especial al Ingeniero Javier Fernández, por todo el apoyo y dedicación prestado hacia nuestro proyecto.

Al personal de los laboratorios, quienes siempre estuvieron disponibles para colaborar en lo necesario durante el proceso.

A Fernando Rodríguez Brizuela y José María Ducloux, encargados del laboratorio de Sistemas Embebidos, quienes siempre tuvieron la mejor predisposición y se encontraban disponibles para colaborar ante cualquier situación referida al trabajo final de grado.

“Establecimiento de un canal de voz codificado half-duplex utilizando algoritmo de cifrado”

Ferri, Facundo
Tobares, Julio Gabriel

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Resumen

En el presente trabajo, el objetivo ha sido implementar un sistema embebido que permita una comunicación de voz segura escalable a la red telefónica cableada existente. Para tal fin, se evaluaron distintas técnicas de protección de información junto con herramientas de procesamiento de datos, logrando así que la señal de voz sea comprimida, cifrada y luego enviada por un canal de comunicaciones que ha de cumplir determinadas características.

En primera instancia se presentaron distintas alternativas al diseño del sistema, detallando las plataformas elegidas y algoritmos a implementar. Luego se desarrollaron los aspectos que conciernen a la seguridad del canal, teniendo como principal enfoque el correcto desempeño en plataformas embebidas de bajo rendimiento. A partir de allí, se montó el sistema completo obteniéndose resultados favorables durante las pruebas preliminares de funcionamiento.

Sin embargo, buscando mejorar la eficiencia del sistema, se incorpora el códec de voz G.711 cumpliendo con las especificaciones de un canal telefónico según recomendación de ITU-T. Los resultados, como queda expuesto en las pruebas de validación y verificación, fueron muy satisfactorios en todos los aspectos a excepción de la latencia, aunque esto no supuso ninguna complicación para la comunicación.

Palabras Clave

- Canal de voz
- Canal telefónico
- Cifrado
- Códec de voz
- Comunicación segura
- Diezmado
- Diffie-Hellman
- Distorsión armónica
- Filtro FIR
- Half-Duplex
- Interpolación
- PortAudio
- Raspberry Pi
- RC4

CAPÍTULO 1: Introducción

En la actualidad, las tecnologías de información están teniendo un rápido desarrollo. Debido a esto, comunicaciones seguras entre individuos y corporaciones son una importante prioridad. Hasta ahora, han sido desarrolladas técnicas para asegurar comunicaciones de texto, audio y video, donde lo fundamental de estas técnicas es proteger la privacidad de la información.

Hoy en día, el uso de comunicaciones de voz en tareas de alta importancia, tales como *home banking*, es algo común en la vida de las personas. Por esta causa, asegurar la integridad de una comunicación es una necesidad creciente. Las tecnologías inalámbricas están más abiertas a ataques que las cableadas [1], por lo que estas últimas son mucho más convenientes cuando un alto nivel de seguridad es requerido.

En el presente trabajo, el objetivo ha sido implementar un sistema embebido que permita una comunicación de voz segura escalable a la red telefónica cableada existente, aunque escoger esta infraestructura con lleva algunas restricciones. En primera instancia, el ancho de banda está comprendido entre 300 Hz y 3300 Hz para la comunicación de voz en redes conmutadas [2], por lo cual, la voz debe ser comprimida antes de ser enviada a través del canal. El proceso de compresión reduce la cantidad de muestras y también la calidad de la voz, por lo que la capacidad de una comunicación segura posiblemente es mermada.

Dentro de las soluciones comerciales existentes actualmente para este problema, la firma Crypto AG con su producto HC-2203 [3] es quien mejor aborda esta problemática. Sin embargo, la importación de este producto se ve limitada en nuestro país y requiere un elevado costo para llevarse a cabo. Es por eso que, como un objetivo secundario, este proyecto utiliza la menor cantidad de recursos económicos posibles.

El proyecto se lleva a cabo empleando dos dispositivos Raspberry Pi y dos placas de audio externas USB. La comunicación entre los dispositivos da inicio con el compartimiento de una información secreta común para las partes involucradas, la cual será usada como material criptográfico para posteriormente cifrar, y realizar el envío y recepción de datos de voz. A través de este método, una vez que la primera etapa ha sido exitosa, es prácticamente imposible obtener acceso a los datos secretos.

El resto del documento se encuentra diagramado de la siguiente manera. En el capítulo 2 se presenta el diseño del sistema, donde se detallan las especificaciones con las cuales debe contar según recomendación de ITU-T. El capítulo 3 desarrolla los aspectos que conciernen a la seguridad del canal, teniendo como principal enfoque el correcto desempeño en plataformas embebidas de bajo rendimiento. Luego, en el capítulo 4 se monta el sistema completo detallando las plataformas elegidas y algoritmos utilizados. Las pruebas realizadas, como queda expuesto en el capítulo 5, fueron satisfactorias en todos los aspectos. En el capítulo 6 se presentan las conclusiones del trabajo.

CAPÍTULO 2: Diseño del sistema

El diagrama en bloques del sistema propuesto en este proyecto se muestra en la Figura 1.

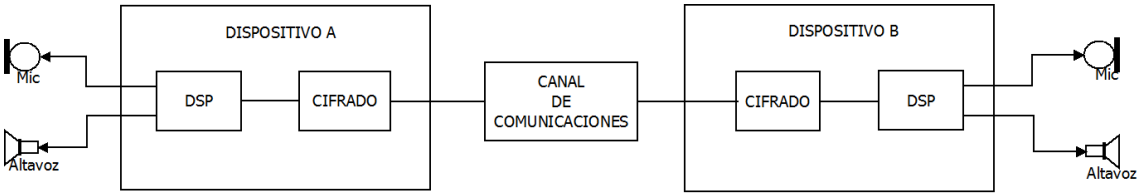

Figura 1: Diagrama en bloques general del sistema

Cada bloque refiere a una etapa de procesamiento distinta, y básicamente el funcionamiento del conjunto es el siguiente:

- Ingresa una señal de voz a través del micrófono.
- La señal de voz es digitalizada y procesada.
- Los datos son cifrados.
- La información cifrada deja el dispositivo de origen y es transmitida hasta el dispositivo destino.
- Los datos son descifrados.
- Se procesa la información para reconstruir la señal de voz.
- La señal de voz emerge a través del parlante.

Debido al alcance del proyecto, la comunicación será unidireccional en cada momento dado, es decir, *half duplex*.

Al ser un requerimiento la escalabilidad hacia la estructura telefónica subyacente y el eje donde se basa el sistema, se tomará como punto de partida el canal de comunicaciones para el diseño.

Canal de comunicaciones

El canal de comunicaciones del proyecto será digital, pero cumplirá con los requerimientos para que funcione sobre un canal telefónico. Se deja en manos de un estudio futuro el proceso de modulación que adapte la señal de salida de este prototipo a la red telefónica.

El canal de telefónico convencional, definido bajo la serie 3000 del documento Tariff FCC No. 260 de la Agencia Federal de Comunicaciones de los Estados Unidos [2], es adecuado para la transmisión híbrida de voz y datos. Acepta velocidades de hasta 14400 bps, con un ancho de banda aproximado de 300 Hz a 3300 Hz [4].

Debido a que la red telefónica convencional está normada y regulada por las entidades correspondientes, se pueden esperar ciertos parámetros de calidad para el diseño.

Según la recomendación M.1020 de la ITU [4]:

- Ruido aleatorio del circuito: en ningún caso excederá el límite máximo de –38 dBm para cableados de más de 10.000 km. En el cableado nacional, las distancias son más cortas y el ruido inducido de esta manera será mucho menor.
- Ruido impulsivo: el límite establecido es que no podrá excederse 18 impulsos de ruido con crestas superiores a –21 dBm en un periodo de 15 minutos.
- Fluctuación de fase: los valores no deben exceder los 10º pico a pico.
- Distorsión total: la relación señal/ruido debe ser mejor que 28 dB cuando se mide con una señal sinusoidal de –10 dBm de nivel.
- Erro de frecuencia: no podrá ser superior a $\pm$ 5 Hz.
- Distorsión armónica: cuando en el extremo de emisión de un circuito punto a punto se aplique una frecuencia de prueba de 700 Hz con un nivel de –13 dBm, el nivel de toda frecuencia armónica en el extremo de recepción será 25 dB inferior, como mínimo, al nivel de la frecuencia fundamental recibida.

Basándose en la no exclusividad de la realización del total de las pruebas anteriormente mencionadas a la hora de medir el rendimiento de un canal digital y escalable, durante el desarrollo de este proyecto se analizarán solo los resultados derivados de las pruebas directamente necesarias.

DSP

DSP es acrónimo de *Digital Signal Proccesor*, o procesador digital de señal. En este bloque se llevarán a cabo distintos procedimientos para adecuar la señal de voz para ser cifrada y transmitida por el canal de comunicaciones.

Para llevar a cabo la implementación del sistema es necesario realizar conversiones en las tasas de muestreo, debido al hardware a utilizar. Si la plataforma escogida no cuenta con DAC o ADC, se debe incluir una placa externa y realizar una conversión sobre las tasas de muestreo.

Conversión de tasa de muestreo

El proceso de conversión de la tasa de muestreo en el dominio digital se puede ver como una operación de filtrado lineal, como se muestra en la Figura 2.

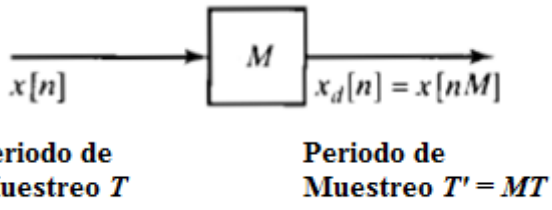

Figura 2: Proceso de conversión de la tasa de muestreo
Extraído de "Signals and Systems; 2nd ed. Prentice hall. Alan V. Oppenheim, Willsky, Young"

La señal de entrada $x(n)$ se caracteriza mediante la tasa de muestreo $F = 1/T$ y la señal de salida $y(m)$ mediante $F' = 1/T'$, donde T y T' son los intervalos de muestreo correspondientes.

Se considerarán dos casos especiales de conversión de tasa de muestreo. Uno es el caso de reducción de la tasa de muestreo por un factor M, y el segundo es el caso de un incremento de tasa de muestreo por un factor entero L.

El proceso de reducción de la tasa de muestreo por un factor M (submuestreo por M) se denomina diezmado. El proceso de incrementar la tasa de muestreo por un factor entero L (sobremuestreo por L) se denomina interpolación.

Diezmado por un factor M

Supongamos que la señal $x(m)$ con espectro $X(\omega)$ se va a submuestrear por un factor entero M. Si se reduce la tasa de muestreo simplemente seleccionando uno de cada M valores de $x(n)$, la señal resultante será una versión alisada de $x(n)$, con una frecuencia de plegado de $F/(2M)$.

Para evitar *aliasing*, primero se debe reducir el ancho de banda de $x(m)$ a $Fmax = F/(2M)$ a través de un filtro pasa-bajos o anti-alias. Después se debe submuestrear por M y así evitar *aliasing*. En la Figura 3 se muestra el proceso de diezmado.

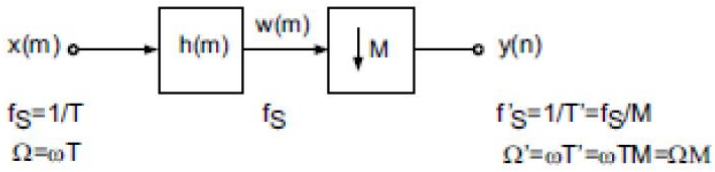

Figura 3: Sistema que implementa un filtrado pasabajos y posterior submuestreo de la señal
Extraído de "Digital Audio Signal Processing; Udo Zölzer. 2nd ed., JohnWiley & Sons Ltd, 2008"

Posteriormente, en la Figura 4 se puede ver un ejemplo del comportamiento del sistema para una señal genérica $x(m)$, muestreada originalmente con una frecuencia de muestreo fs y que es reducida en un factor $M = 3$.

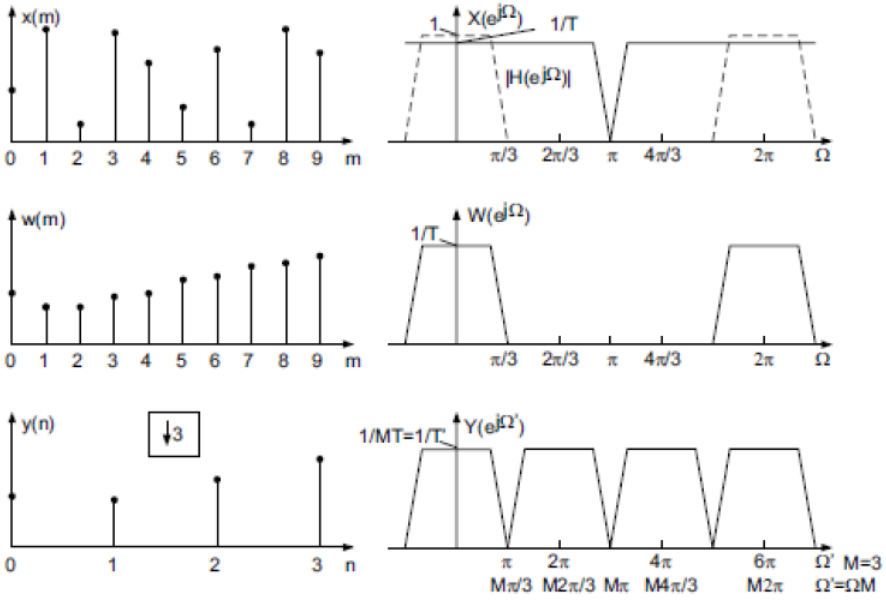

Figura 4: Ejemplo de un filtrado anti-alias y submuestreo por un factor de 3
Extraído de "Digital Audio Signal Processing; Udo Zölzer. 2nd ed., JohnWiley & Sons Ltd, 2008"

Interpolación por un factor L

Se puede lograr un incremento en la tasa de muestreo por un factor entero de L interpolando $L-1$ nuevas muestras entre sucesivos valores de la señal. El proceso de interpolación se puede conseguir de varias formas.

Para llevarlo a cabo, primero se debe realizar una expansión temporal, en la práctica no es más que agregar $L-1$ ceros entre muestra y muestra de la señal $x(m)$. Luego de la expansión se realiza el filtrado pasa bajos para interpolar las muestras de la señal original.

En la Figura 5 se puede ver un esquema del sistema completo, y en la Figura 6 un ejemplo de aplicación en una señal genérica.

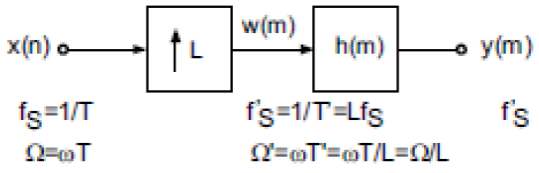

Figura 5: Esquema de un sistema que aumenta la frecuencia de muestreo en un factor entero
Extraído de "Digital Audio Signal Processing; Udo Zölzer. 2nd ED., John Wiley & Sons Ltd, 2008"

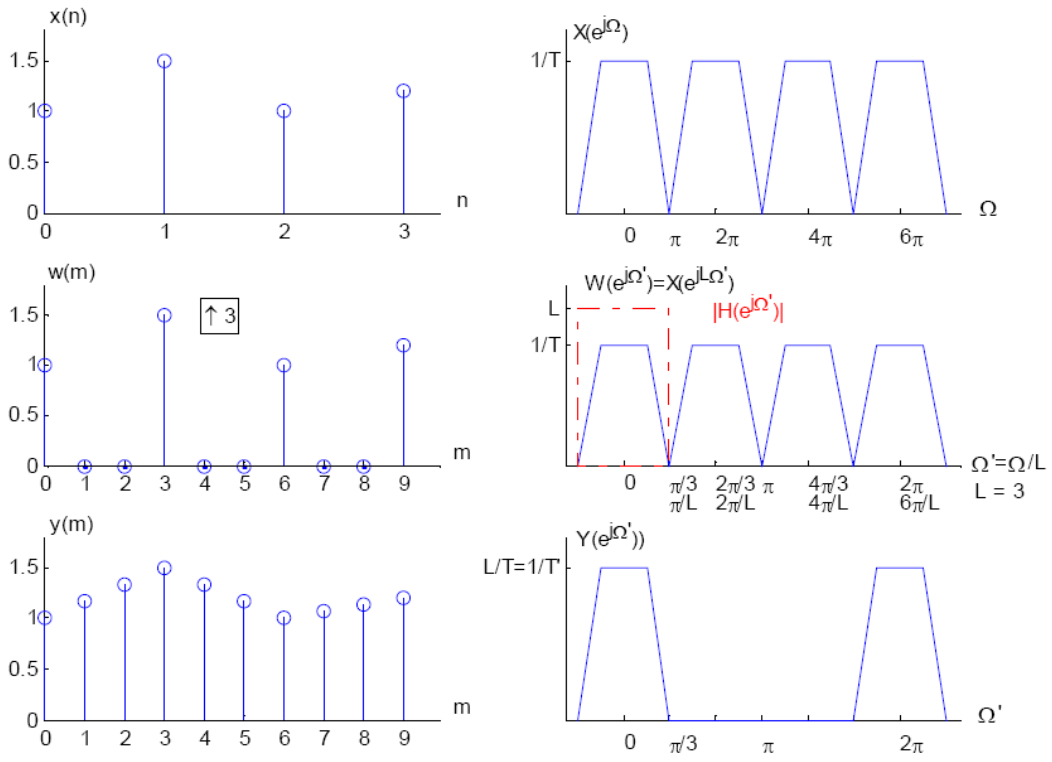

Figura 6: Ejemplo de un sobremuestreo por 3 y posterior filtrado interpolador.
Adaptado de "Digital Audio Signal Processing; Udo Zölzer. 2nd ed., JohnWiley & Sons Ltd, 2008"

El filtro pasa bajos de la figura anterior es el encargado de interpolar las muestras de la secuencia en el tiempo.

Códec de voz

Un códec de voz es un método de compresión y descompresión de una señal de audio que contiene datos de voz [5]. La palabra proviene de la abreviatura del inglés *coder-decoder*.

En 1972, la CCITT publicó la recomendación G.711 la cual constituye la referencia principal en los sistemas de transmisión. El principio básico del algoritmo es codificar voz usando 8 bits por muestra, la señal de banda base debe ser muestreada a 8 kHz, manteniendo el ancho de banda telefónico de 300-3400 Hz. Con esta combinación cada canal de voz requiere 64 kbps.

La idea detrás de digitalizar la red implica un compromiso: maximizar el uso de la infraestructura existente. Esto impone limitaciones de ancho de banda para los flujos de datos de las señales codificadas. Una tasa de 64 kbps fue encontrada como razonable. Se emplea un esquema de cuantificación lineal.

Un inconveniente de este enfoque es que la Relación Señal a Ruido (SNR – *Signal Noise Ratio*) varía con la amplitud de las señales de entrada. Desde el punto de vista de calidad, si la señal posee grandes variaciones, o es una señal que varía con el tiempo (como en el caso de señales de voz), la SNR variará, resultando en un amplio rango de calidad del sistema. Para

abordar este problema, se utiliza cuantificación logarítmica dentro de dos esquemas de transmisión, empleando la ley A, utilizada en Europa y el resto del mundo; o ley μ, utilizada en Estados Unidos y Japón. Ambas presentan un comportamiento lineal para señales de amplitudes pequeñas (siendo el equivalente a un esquema de cuantificación lineal), pero es logarítmico para grandes amplitudes.

Para señales fuera de la zona lineal de cuantificación, la SNR para la ley A es:

$$SNR_A = 6.02\,B + 4.77 - 20\log(1 + \ln A)$$

donde B es el número de bits usados para cuantificar.

ITU escoge el valor $A = 87.56$ para el estándar G.711, con 8 bits por muestra, por lo cual:

$$SNR_A = 6.02\,B - 10.1 = 38.06\,dB$$

G.711 utiliza 1 bit de signo, bits 2-4 para indicar un segmento y bits 5-8 para el nivel. Dentro de cada segmento, la cuantificación es lineal (4 bits o 16 niveles), teniendo 13 segmentos de pendientes distintas para la ley A, la cual trabaja con señales en el rango de –4096 a 4096, implicando un rango de 13 bits [6].

Cifrado

Durante este proceso la información de voz será codificada de manera que no pueda ser interpretada por un atacante externo. Para esto se requerirá un algoritmo de cifrado, que se servirá del uso de una clave compartida entre las partes involucradas.

Según la clave, el cifrado se puede clasificar en dos grupos [7]:

- Cifrado simétrico: es un método criptográfico monoclave, esto quiere decir que se usa la misma clave para cifrar y descifrar. Los métodos más conocidos son *DES* y *AES*.
- Cifrado asimétrico: al usar claves diferentes. Este método utiliza una pareja compuesta por una clave pública, que sirve para cifrar, y por una clave privada, que sirve para descifrar. El punto fundamental sobre el que se sostiene esta descomposición pública/privada es la imposibilidad práctica de deducir la clave privada a partir de la clave pública. El método más conocido es el *RSA*.

Según la forma en la que operan los algoritmos de cifrado y descifrado, también es posible distinguir dos tipos [8]:

- Cifrado en flujo, en los que el algoritmo de cifrado se realiza bit a bit. Están basados en la utilización de claves muy largas tanto para cifrar como para descifrar.
- Cifrado en bloques, donde el cifrado se realiza bloque a bloque. En primera instancia, se descompone el mensaje en bloques de la misma longitud. A

continuación, cada bloque se va convirtiendo en un bloque del mensaje cifrado mediante una secuencia de operaciones.

Es importante además que ambas partes posean algún tipo de llave para poder realizar el cifrado. Este objetivo se logra a través de un protocolo de compartimiento de clave, que se define como un protocolo criptográfico en el que se establece una secuencia de pasos entre dos o más participantes a través de la cual éstos se ponen de acuerdo en un valor de una información secreta compartida [9]. Si el procedimiento es realizado correctamente, ningún oyente externo a las partes puede obtener esta información. Muchos sistemas de intercambio de clave hacen generar ésta por una sola parte, y luego simplemente es enviada al resto, que no tienen influencia sobre ella.

Los protocolos en que todas las parten tienen influencia sobre la generación de llave son los únicos que son capaces de implementar Secreto Perfecto hacia Delante [9] (PFS –*Perfect Forward Secrecy*). PFS es la propiedad de los sistemas criptográficos que garantiza que el descubrimiento de las claves utilizadas actualmente no compromete la seguridad de las claves usadas con anterioridad.

El primer protocolo conocido públicamente que cumple con el criterio anterior fue el protocolo de intercambio de llaves Diffie-Hellman. En este método, las partes son anónimas entre sí, por lo cual es susceptible a ataques Hombre en el Medio (MIM – *Man-in-the-Middle*).

A partir de Diffie-Hellman, se han desarrollado muchos protocolos de intercambio de claves que son robustos a ataques MIM. Algunos ejemplos son MQV, YAK, y el ISAKMP. Sin embargo, estos métodos requieren el uso de entidades certificadas por autoridades para su correcto funcionamiento [9].

Debido a que este proyecto no intenta desarrollar una robustez comercial de alto nivel, sino más bien ser una demostración de la factibilidad del uso de plataformas embebidas para una comunicación cifrada, el protocolo Diffie-Hellman sin protección contra MIM satisface los objetivos correctamente.

CAPÍTULO 3: Seguridad

Introducción a la Criptografía

Antes de desarrollar el tema, es necesario aclarar en qué consiste la criptografía.

Según la RAE [10]:

> Criptografía: Arte de escribir con clave secreta o de un modo enigmático.

Aportando una visión más específica, la criptografía es la creación de técnicas para el cifrado de datos, teniendo como objetivo conseguir la confidencialidad de los mensajes. Si la criptografía es la creación de mecanismos para cifrar datos, el criptoanálisis hace referencia a los métodos para "romper" estos mecanismos y obtener la información. Una vez que un conjunto de datos ha pasado un proceso criptográfico, decimos que la información se encuentra cifrada.

Métodos de cifrado más comunes

DES

Estándar de cifrado de datos (DES - *Data Encryption Standard*). Es un algoritmo de cifrado por bloques. Se toma un bloque de 64 bits y lo transforma mediante una serie de operaciones básicas en otro bloque cifrado de la misma longitud. La clave también tiene 64 bits pero 8 de estos bits se emplean para comprobar la paridad, haciendo que la longitud efectiva de la clave sea de 56 bits [8].

DES se compone de 16 fases o rondas idénticas. Al comienzo y al final se realiza una permutación. Estas permutaciones no son significativas a nivel criptográfico, pues se incluyeron para facilitar la carga y descarga de los bloque en el hardware de los años 70. Antes de cada ronda el bloque se divide en dos mitades de 32 bits y se procesan alternativamente. Este proceso es conocido como esquema Feistel [8]. El esquema Feistel proporciona un proceso de cifrado y descifrado casi iguales. La única diferencia es que las subclaves se aplican de forma inversa al descifrar. En la Figura 7 puede apreciarse un diagrama que muestra el funcionamiento de DES.

Función F-Feistel:

1. Expansión: se toma la mitad del bloque de 64 bits (32bits) que son expandidos a 48 bits mediante la permutación de expansión, denominada E en el diagrama, duplicando algunos de los bits.
2. Mezcla: el resultado se combina con una subclave utilizando una operación XOR. Dieciséis subclaves (una para cada ronda) se derivan de la clave inicial.
3. Sustitución: tras la mezcla, el bloque es dividido en ocho trozos de 6 bits que se pasan a las cajas de sustitución. Cada una de las ocho S-cajas reemplaza sus 6 bits de entrada con 4 bits de salida, de acuerdo con una trasformación no lineal, especificada por una

tabla. Las S cajas constituyen el núcleo de la seguridad de DES, sin ellas, el cifrado sería lineal y fácil de romper.

4. Permutación: finalmente, los 32 bits salientes de las S-cajas se reordenan de acuerdo a una permutación fija.

Generación de claves

De los 64 bits iniciales se toman 56 con la Elección Permutada 1 (PC-1). Estos 56 bits son divididos en dos mitades de 28 bits que serán tratadas de forma independiente [8]. En rondas sucesivas se desplazan los bits de ambas mitades 1 o 2 bits a la derecha. Tras el desplazamiento se toman 48 bits (24+24) mediante la Elección Permutada 2 (PC-2). Al realizar un desplazamiento en cada ronda cada subclave estará empleando un conjunto diferente de bits.

La generación de claves para descifrado es similar, la única variación es que se deben generar en orden inverso.

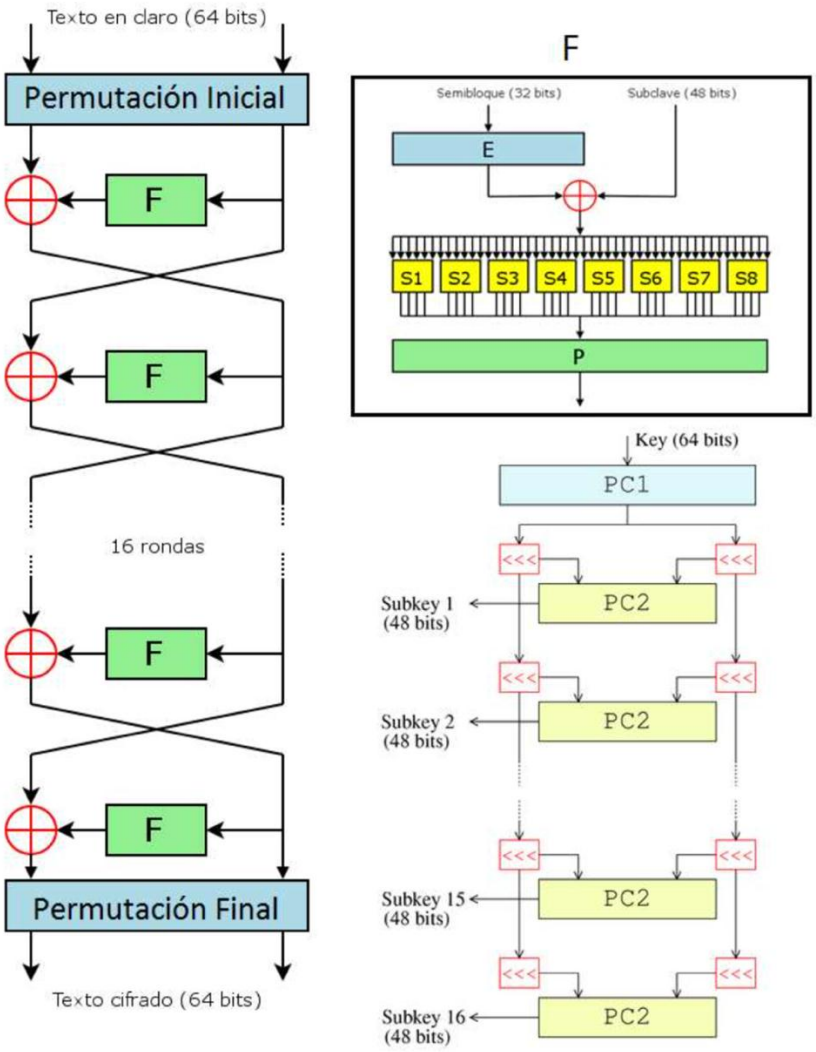

Figura 7: Generación de clave en DES
Extraído de "Criptografía y Métodos de cifrado, Universidad de Acála"

AES

Estándar de cifrado avanzado (AES – *Advanced Encryption Standard*). También conocido como Rijndael, fue el ganador del concurso convocado en el año 1997 por el Instituto Nacional de Normas y Tecnología (NIST – *National Institute of Standards and Technology*) con objetivo de escoger un nuevo algoritmo de cifrado [8]. En 2001 fue tomado como FIPS y en 2002 se transformó en un estándar efectivo. Desde el año 2006 es el algoritmo más popular empleado en criptografía simétrica.

AES opera sobre una matriz de 4x4 bytes. Mediante un algoritmo se reordenan los distintos bytes de la matriz. El cifrado es de clave simétrica, por lo que la misma clave aplicada en el cifrado se aplica para el descifrado [8].

Basado en el algoritmo Rijndael, al contrario que su predecesor DES, Rijndael es una red de sustitución-permutación, no una red de Feistel. AES es rápido tanto en software como en hardware, es relativamente fácil de implementar, y requiere poca memoria [8].

El algoritmo AES funciona mediante una serie de bucles que se repiten. 10 ciclos para claves de 128 bits, 12 para 192 y 14 para 256.

La Figura 8 visualiza el proceso realizado por AES. Suponer que se tienen 2 matrices: matriz *a* y matriz *k*.

$$\begin{matrix} a00\ a01\ a02\ a03 \\ a10\ a11\ a12\ a13 \\ a20\ a21\ a22\ a23 \\ a30\ a31\ a32\ a33 \end{matrix}$$

$$\begin{matrix} k00\ k01\ k02\ k03 \\ k10\ k11\ k12\ k13 \\ k20\ k21\ k22\ k23 \\ k30\ k31\ k32\ k33 \end{matrix}$$

En la matriz *a* se tiene la información y en la matriz *k* una subclave generada a partir de la principal.

El algoritmo de cifrado es el siguiente [8]:

1. Expansión de la clave: mediante una serie de operaciones se obtienen $n + 1$ subclaves a partir de la clave principal (n es el número de ciclos).
2. Ciclo inicial *AddRoundKey*: se aplica una XOR byte byte entre la matriz *a* y la matriz *k*.
3. Ciclos intermedios:
   3.1. *SubBytes*: tomando como referencia una tabla especificada cada byte es sustituido por otro en función de la tabla.
   3.2. *ShiftRows*: cada byte de cada fila es desplazada $n - 1$ huecos a la izquierda (siendo n el número de fila).

3.3. *MixColumns*: los 4 bytes de una columna se combinan entre sí para obtener 4 bytes diferentes. Este proceso se logra multiplicando la columna por una matriz dada.

$$\begin{matrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{matrix}$$

3.4. *AddRoundKey*.

4. Ciclo final:
   4.1. *SubBytes*.
   4.2. *ShiftRows*.
   4.3. AddRoundKey

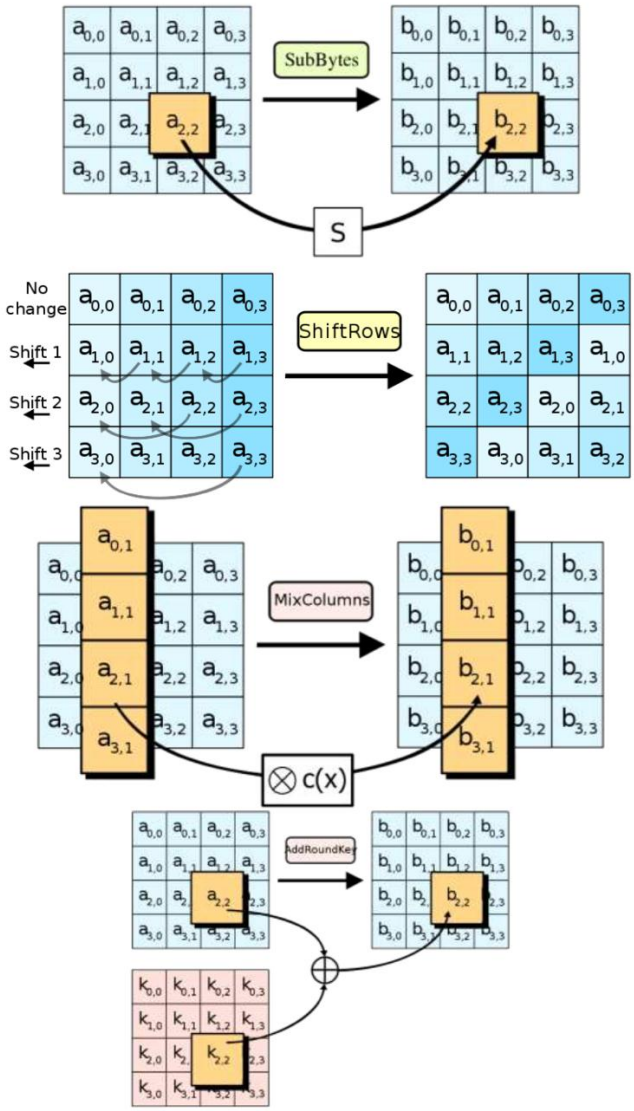

Figura 8: Generación de clave en AES
Extraído de "Criptografía y Métodos de cifrado, Universidad de Acála"

WEP-RC4

Privacidad Equivalente a Cableado (WEP – *Wired Equivalent Privacy*). Es el sistema de cifrado incluido en el estándar IEEE 802.11 [8]. WEP está basado en el algoritmo de cifrado RC4, teniendo dos variantes, una que usa clave de 64 bits (40 bits más 24 bits de vector de iniciación) y otra que usa clave de 128 bits (104 bits y 24 bits de vector de iniciación). En 2003 la Wi-Fi Alliance anunció la sustitución de WEP por WPA y en 2004 se ratificó el estándar 802.11i declarando WEP-40 y WEP-104 como inseguros. A pesar de ello todavía hoy en día se sigue utilizando.

Como se ha dicho, WEP utiliza el algoritmo de cifrado RC4. El funcionamiento es el siguiente:

Se expande una semilla o *seed* para generar una secuencia de números pseudoaleatorios de mayor tamaño. Posteriormente, se "unifican" el mensaje y la secuencia pseudoaleatoria mediante una función XOR.

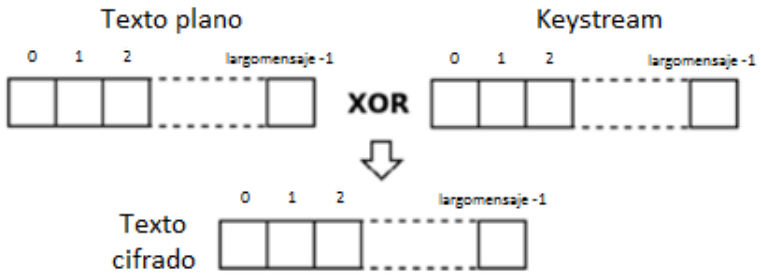

Figura 9: Cifrado a través de la función XOR

El problema que presenta este método es que si se utiliza la misma semilla para cifrar dos mensajes diferentes, obtener la clave a partir de los dos textos cifrados sería trivial. En un intento de evitar esto, en WEP se incluyó un vector de iniciación de 24 bits que se modifica regularmente y se concatena a la contraseña para generar la semilla.

El principal defecto de WEP se encuentra precisamente en este vector de iniciación. El tamaño del vector de iniciación es constante (24 bits), esto genera un número limitado de vectores (2^{24}=16.777.216). El problema es que la cantidad de tramas que pasan a través de un punto de acceso son muy grandes y es fácil encontrar 2 mensajes con el mismo vector haciendo relativamente fácil obtener la clave. Se puede aumentar el tamaño de la clave, pero esto solo incrementará el tiempo necesario para romper el cifrado.

RSA

Rivest, Shamir y Adleman – Es un algoritmo de cifrado asimétrico desarrollado en el año 1977 por los anteriormente citados.

Este algoritmo se basa en escoger 2 números primos grandes, a llamar e y d, elegidos de forma aleatoria y mantenidos en secreto. La principal ventaja desde el punto de vista de seguridad, radica en la dificultad a la hora de factorizar números grandes [8]. RSA es seguro hasta la fecha [7,11].

RSA requiere el uso de un protocolo reversible, denominado "patrón de relleno", dado que sino el mensaje puede conducir a textos cifrados inseguros. Hay múltiples algoritmos de rellenado, a destacar Relleno Asimétrico de Cifrado Óptimo (OAEP – *Optimal Asymetric Encription Padding*) o Relleno Asimétrico de Cifrado Simplificado (SAEP – *Simplified Asymetric Encryption Padding*).

El funcionamiento de RSA es básicamente el siguiente [8]:

Sea M el mensaje a cifrar. Empleando el patrón de relleno convertimos el mensaje M en un número m menor que otro número dado n.

Luego se genera el mensaje cifrado c:

$$c \equiv m^e \ (mod\ n)$$

Y se obtiene el mensaje descifrando a c:

$$m \equiv c^d \ (mod\ n)$$

Conclusión

Para algunas aplicaciones, como es el caso de las conversaciones telefónicas, el cifrado en bloques es inapropiado porque los flujos de datos se producen en tiempo real en pequeños fragmentos. Las muestras de datos pueden ser tan pequeñas como 8 bits o incluso de 1 bit, y sería un desperdicio rellenar el resto de los 64 bits antes de cifrar y transmitirlos. RC4, en cambio, convierte el bit limpio a bit cifrado de a uno por vez, haciéndolo mucho más conveniente para los objetivos que intenta cumplir este proyecto.

Funcionamiento de RC4

Como se ha dicho anteriormente, el algoritmo RC4 genera una llave pseudoaleatoria, a nombrar *keystream*, que es usada para generar el texto cifrado a través de la operación lógica XOR con el mensaje. Se le llama pseudoaleatoria porque genera una secuencia de números de una forma que se aproxima a las propiedades de los números aleatorios. Asimismo, no es totalmente aleatoria porque recibe una semilla, la cual va a dar como resultado siempre el mismo valor de salida si ésta no es cambiada, siendo así muy difícil de romper. El *keystream* se genera a partir de una clave de longitud variable en base a los siguientes elementos [12]:

- Un arreglo de 256 bytes, llamado `S`, que contiene la permutación de esos 256 bytes.
- Dos contadores, `i` y `j`, usados para denotar los elementos en el arreglo.

Una vez que el arreglo `S` ha sido inicializado y "aleatorizado" con el Algoritmo Generador de Código (KSA – *Key-Scheduling Algorithm*), éste es modificado con el Algoritmo de Generación Pseudoaleatorio (PRGA – *Pseudo-Random Generation Algorithm*) para generar finalmente el *keystream*. Ahora se describirán en detalle cada uno de estos algoritmos.

KSA

Como fue explicado, este algoritmo es usado para generar el arreglo de permutación.

El primer paso consiste en inicializar el arreglo `S` con la identidad de permutación: el valor del arreglo es igual al valor que almacena [12]. Entonces:

```
NUMERO S[256], KEYSTREAM[largodelmensaje], i, j
PARA i = 0 HASTA i = 255 HACER
      S[i] = i
FIN PARA
```

Una vez listo, el siguiente paso es mezclar el arreglo usando como base la llave para hacer el arreglo permutado. Para esto, iteramos 256 veces las siguientes acciones después de inicializar `i` y `j` a `0`, tal como se muestra en la Figura 10.

```
PARA i = 0 HASTA i = 255 HACER
      j = j + S[i] + key[i % largodekey]
      INTERCAMBIAR(S[i],S[j])
FIN PARA
```

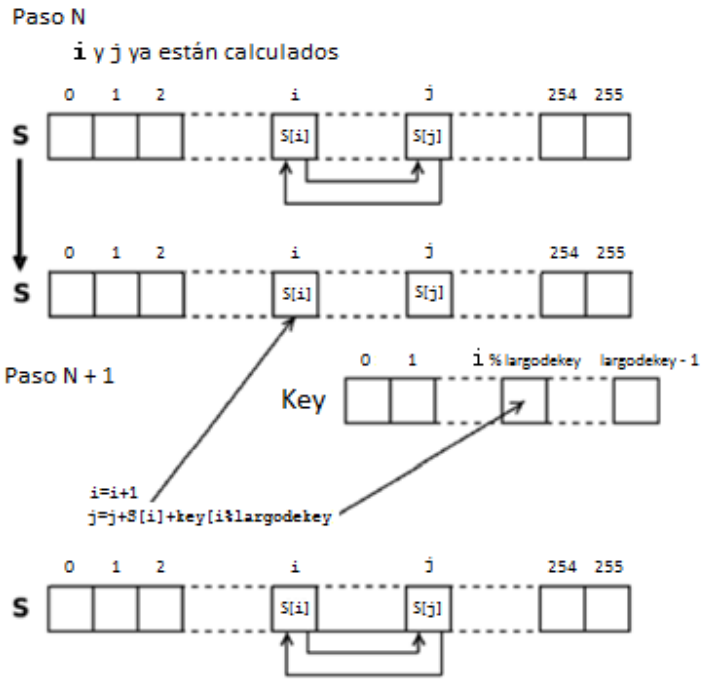

Figura 10: Funcionamiento del algoritmo KSA

Una vez completadas las 256 iteraciones, el arreglo `S` ha sido inicializado correctamente.

El siguiente paso es el algoritmo de generación pseudoaleatorio.

PRGA

Este algoritmo consiste en generar el *keystream* del tamaño del mensaje a cifrar, que puede ser de cualquier tamaño.

Para esto, hay que iniciar los dos contadores a `0` y empezar generando el *keystream* de a un byte por vez hasta haber alcanzado el tamaño del mensaje a cifrar. Por cada byte es necesario realizar el siguiente proceso [12]:

```
PARA a = 0 HASTA a = largodelmensaje HACER
     i = (i + 1) % 256
     j = (j + S[i]) % 256
     INTERCAMBIAR(S[i],S[j])
     KEYSTREAM[a] = S[S[i] + S[j] % 256]
FIN PARA
```

Cifrado y descifrado

Luego de generado el *keystream*, el último paso pendiente es el cifrado, es decir, la función XOR entre el anterior y el mensaje original, la cual dará por resultado el mensaje cifrado:

```
PARA a = 0 HASTA a = largodelmensaje HACER
     CIPHER[a] = MENSAJE[a] XOR KEYSTREAM[a]
FIN PARA
```

Una gran ventaja de este algoritmo es que para el descifrado sólo se debe repetir la operación, pero esta vez entre el texto cifrado y el *keystream*:

```
PARA a = 0 HASTA a = largodelmensaje HACER
     MENSAJE[a] = CIPHER[a] XOR KEYSTREAM[a]
FIN PARA
```

Funcionamiento de Diffie-Hellman

Para el caso del intercambio de llaves entre sólo dos partes, Diffie-Hellman requiere que cada una de éstas posea un par de números, uno llamado clave pública y el otro llamado clave privada. Combinando la privada de uno con la pública del otro, ambas partes pueden calcular un número, a llamar K, que es usado como material criptográfico de cifrado.

Generación de K

Teniendo en cuenta dos partes, Alicia y Bruno. En primera instancia, se acuerda usar dos números, p y g, tales que p es un primo grande, y $g = h^{(p-1)/q} \ (mod\ p)$, donde g es raíz primitiva de p; y donde h y q son tales que q es un primo grande h es un entero entre $1 < h < p - 1$. De esta manera $h^{(p-1)/q} \ (mod\ p) > 1$.

Ninguno de estos números se mantiene en secreto frente a posibles atacantes. Ahora, Alicia elige otro número grande $1 \leq a \leq p-2$, el cual será su llave secreta. Similarmente, Bruno también elige otro número grande $1 \leq b \leq p-2$. Entonces Alicia calcula $A = g^a \ (mod\ p)$, el cual envía a Bruno, y este calcula $B = g^b \ (mod\ p)$, que lo envía a Alicia.

Ahora ambos generan su llave compartida $K = g^{ab} \ (mod\ p)$, que Alicia calcula:

$$K = B^a \ (mod\ p) \rightarrow K = (g^b)^a \ (mod\ p)$$

Y análogamente Bruno lo calcula como:

$$K = A^b \ (mod\ p) \rightarrow K = (g^a)^b \ (mod\ p)$$

De esta manera, ambas parten ahora poseen K [13].

Ataque pasivo

En el caso de que un atacante intentase obtener el valor K de manera pasiva, es decir, a través de la información de carácter público, éste poseería las variables p, g, A y B. Para poder obtener K, debería primero calcular a o b. Esto es teóricamente posible invirtiendo la función, dando como resultado que $a = logdisc_p(B)$ o $b = logdisc_p(A)$. Este problema, conocido como el logaritmo discreto en Z_p^*, se considera computacionalmente intratable siempre que p tenga más de 200 dígitos y no cumpla ciertas características debilitantes [13].

CAPÍTULO 4: Implementación

Tomando como referencia el diagrama en bloques sobre el diseño del sistema del Capítulo 2, a continuación se detalla el diagrama del sistema a implementar:

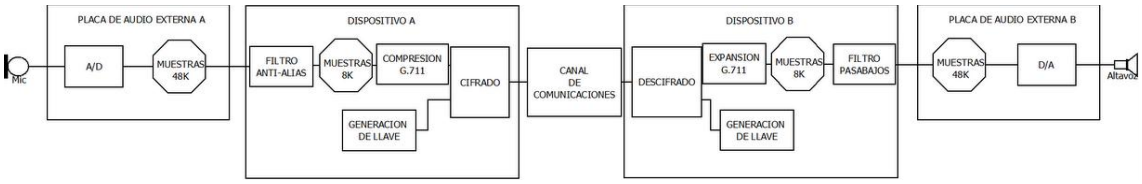

Figura 11: Diagrama en bloques detallado del sistema

Para llevar a cabo la implementación del sistema es necesario seleccionar el hardware a utilizar. A continuación se detalla una comparación [14] sobre especificaciones, performance, consumo de energía y temperatura de diferentes plataformas de sistemas embebidos.

Selección de Hardware

Los sistemas embebidos a comparar serán los siguientes:

- Arduino Yun
- Beaglebone Black
- Intel Galileo
- Raspberry Pi

Especificaciones

En la Tabla 1 se observan las especificaciones de cada placa.

	Arduino Yun	Beaglebone Black	Intel Galileo	Raspberry Pi
SoC	Atheros AR9331	Texas Instruments AM3358	Intel Quark X1000	Broadcom BCM2835
CPU	MIPS32 24K and ATmega32U4	ARM Cortex-A8	Intel X1000	ARM1176
Arquitectura	MIPS and AVR	ARMv7	i586	ARMv6
Frecuencia	400mhz (AR9331) & 16mhz (ATmega)	1ghz	400mhz	700mhz
Memoria RAM	64MB (AR9331) & 2.5KB (ATmega)	512MB	256MB	256MB (model A) o 512MB (model B)
FPU	Software	Hardware	Hardware	Hardware
Networking	10/100Mbit ethernet y 802.11b/g/n WiFi	10/100Mbit ethernet	10/100Mbit ethernet	No posee (model A) o 10/100Mbit ethernet (model B)
Alimentacion	5V de micro USB conector B o header pin.	5V de micro USB mini conector B, 2.1mm jack, o header pin.	5V de 2.1mm jack, o header pin.	5V de micro USB conector B o header pin.
Dimensiones	68.6mm x 53.3mm	86.4mm x 53.3mm	106.7mm x 71.1mm	85.6mm x 56mm
Precio Aproximado	U$D 75	U$D 55 (rev C) U$D 45 (rev B)	U$D 80	U$D 25 (model A), U$D 35 (model B)

Tabla 1: Especificaciones de las distintas plataformas embebidas evaluadas
Extraído de "Embedded Linux Comparison; Adafruit Industries"

En la Tabla 2 se compara cada una de las capacidades I/O de cada placa.

	Arduino Yun	Beaglebone Black	Intel Galileo	Raspberry Pi
Pines I/O digitales	20	65	14	17
Digital I/O power	5V	3,3V	3,3V o 5V (seleccionado con jumper)	3,3V
Entrada Analogica	12 con 10-bit ADC, 0-5V (posee entrada de referencia externa)	7 con 12-bit ADC, 0-1.8V (no posee entrada de referencia externa)	6 con 12-bit ADC, 0-5V (no posee entrada de referencia externa)	-
UART	2 (1 cableado a AR9331)	4	2 (1 via 3.5mm jack)	1
SPI	1	2	1	2
I2C	1	2	1	1

Tabla 2: Capacidades I/O de las plataformas embebidas evaluadas
Extraído de "Embedded Linux Comparison; Adafruit Industries"

Performance

Para evaluar la performance de estos dispositivos fue empleada la herramienta de benchmarks *nbench,* originalmente creada para medir la performance de cada computadora Pentium-class, donde la aplicación lleva a cabo una serie de test que simulan la carga de trabajo real del procesador. El resultado de cada test es combinado para crear un índice de performance sobre memoria, enteros y punto flotante de cada sistema.

Para la prueba de memoria, *nbench* fue compilado en cada placa usando GCC 4.7, con la configuración mínima de optimización, la cual no produce resultados rápidamente. Los resultados se muestran en la Figura 12 y Tabla 3.

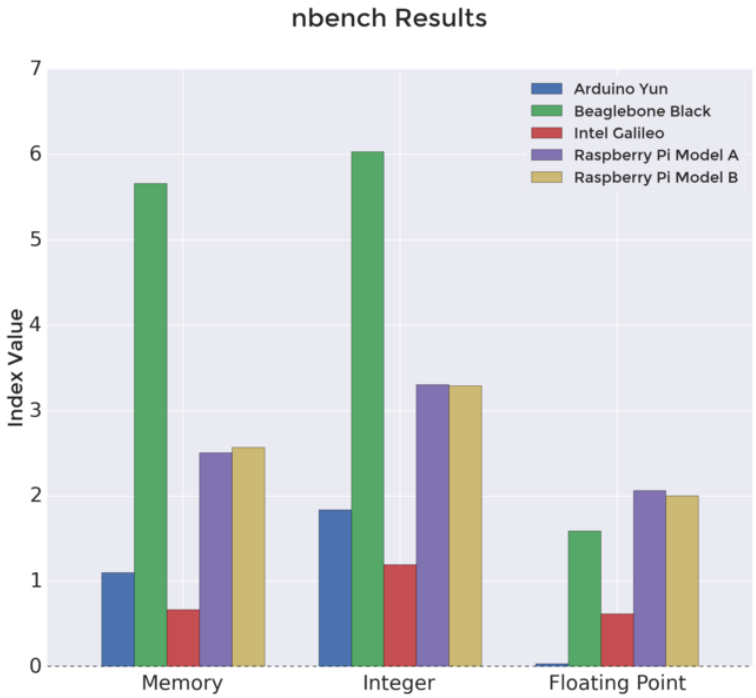

Figura 12: Resultados del sofware nbench en cuanto a variables y uso de memoria
Extraído de "Embedded Linux Comparison; Adafruit Industries"

Indice	Arduino Yun	Beaglebone Black	Intel Galileo	Raspberry Pi Model A	Raspberry Pi Model B
Memoria	1,104	5,661	0,699	2,509	2,57
Enteros	1,84	6,032	1,198	3,305	3,291
Punto Flotante	0,038	1,591	0,621	2,064	2,002

Tabla 3: Resultados del sofware nbench en cuanto a variables y uso de memoria
Extraído de "Embedded Linux Comparison; Adafruit Industries"

Observando los resultados en la Tabla 3, Beaglebone Black posee la mayor performance en memoria y enteros, sin embargo la performance sobre punto flotante se encuentra por debajo de Raspberry Pi. Esto se debe ya que el procesador ARM Cortex-8 sobre Bleaglebone tiene 'VFPLite' de unidad de Punto Flotante la cual no es más veloz que otros ARM FPU'S.

Otra comparación interesante es sobre Arduino Yun e Intel Galileo. Ambas placas corren a una frecuencia de 400 MHz, pero aparentemente la arquitectura MIPS de Arduino posee mayor performance que Intel de Galileo. La performance de punto flotante sobre Yun es bastante baja ya que no cuenta con una Unidad de Punto Flotante (FPU – *Floating-Point Unit*) implementada en hardware y puede correr las operaciones sobre software.

Finalmente ambas Raspberry Pi son idénticas en performance, ya que no existe diferencia entre el procesador de cada placa. El modelo B solo cuenta con más memoria RAM y periféricos que el modelo A.

Consumo de energía

El consumo de energía es medido a través de un monitor de consumo *INA219* conectado a un Arduino y midiendo la entrada de 5 V. En cada placa fue utilizando el mínimo de periféricos

posibles durante el test. Beaglebone Black, Raspberry Pi model B e Intel Galileo fueron conectadas a la red por el puerto Ethernet, Arduino Yun fue conectada a la red vía WIFI. Raspberry Pi model A no fue conectada a la red, se conectó teclado USB y monitor HDMI.

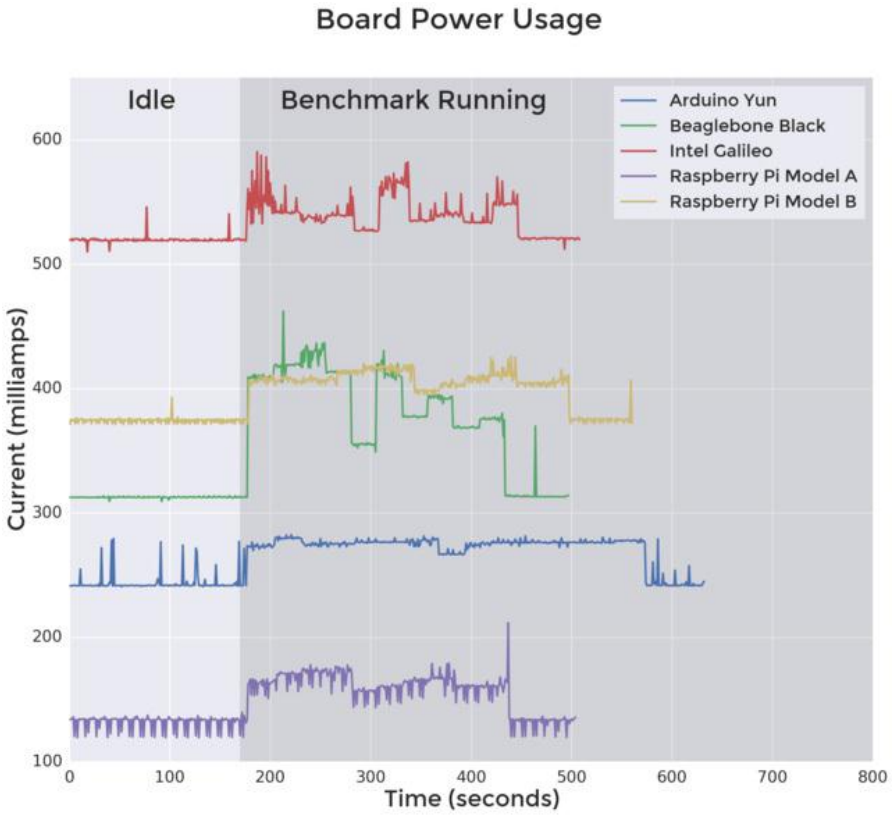

Figura 13: Resultados del sofware nbench en cuanto al consumo de energía
Extraído de "Embedded Linux Comparison; Adafruit Industries"

Según la Figura 13, sobre los resultados obtenidos es notable observar el amplio margen de consumo de energía. Raspberry Pi model A es quien menores valores muestra con 150 mA de consumo promedio, por otro lado, Intel Galileo consume por encima de 500 mA de corriente. Esta diferencia se observa por la variedad de periféricos y chips soportados en cada placa.

BeagleBone y Raspberry Pi model B tienen similares valores de consumo promedio, pero la primer placa es notablemente menor.

Temperatura

La temperatura de los dispositivos fue medida durante el proceso *nbench,* con el sensor de temperatura LM35D conectado al Arduino y a un voltaje de referencia.

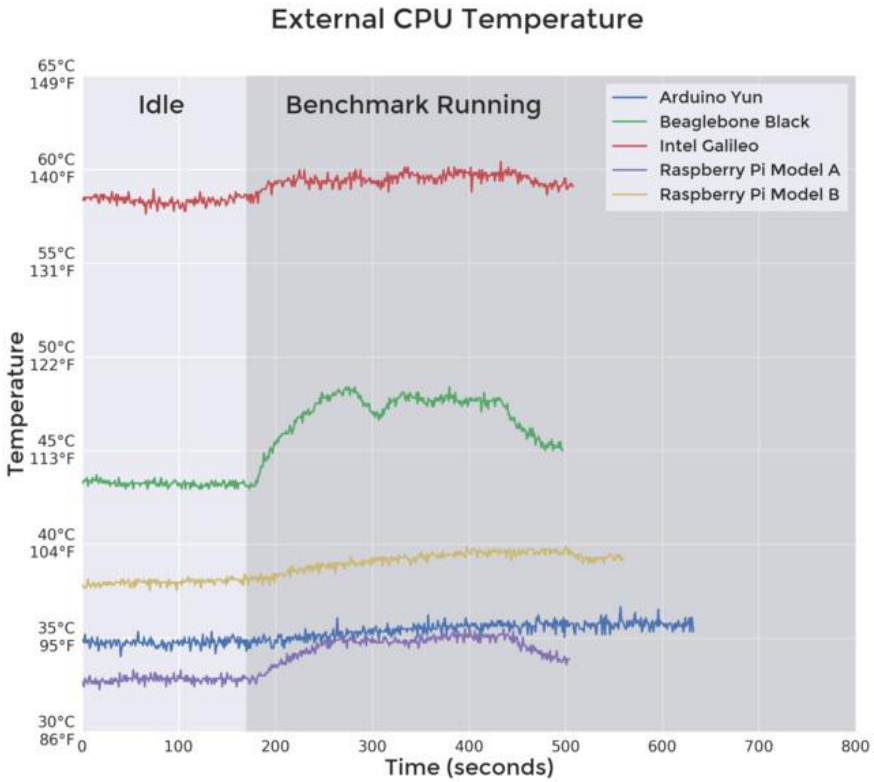

Figura 14: Resultados del sofware nbench en cuanto a las temperaturas alcanzadas
Extraído de "Embedded Linux Comparison; Adafruit Industries"

En la Figura 14 se detallan los resultados obtenidos, donde se puede ver a Intel Galileo con el mayor consumo en 60°C, según su hoja de datos, el procesador de Galileo trabaja a 70°C. La mayoría de los dispositivos trabajan a o debajo de 40°C, lo cual no es excesivamente más alto que la temperatura ambiente.

Evaluando los parámetros obtenidos en el benchmark, el dispositivo que se ajusta a los requerimientos del proyecto es Raspberry Pi model B, ya que se destaca por un bajo costo, consumo de energía promedio y temperatura de trabajo media en relación con la performance de procesamiento.

Configuración de dispositivos

Para llevar a cabo la implementación son necesarios los siguientes elementos:

- 2 Dispositivos Raspberry Pi Model B
- 2 Tarjetas de memoria SD de 4 GB clase 4
- 2 Placas de audio USB
- Misceláneos (cables de alimentación, conectores, par trenzado, etc.)

El proyecto se encuentra desarrollado sobre la última distribución vigente del sistema operativo Raspbian, distribución basada en Debian. En primera instancia se lleva a cabo el

grabado de memorias SD utilizando el software *Win32DiskImager*. Sobre terminal, conectando los dispositivos a través del puerto de red, se debe actualizar el sistema.

Se establece una red LAN entre los dispositivos y notebooks, asignando IP estáticas, para el control remoto de las Raspberry vía SSH, ya que los procesos se desarrollan sobre notebooks investigando y realizando pruebas parciales. A través del protocolo SSH se transfirieren datos hacia los dispositivos, administrando directorios, modificando archivos del sistema, instalando paquetes, etc.

Los dispositivos Raspberry Pi posee solamente un DAC como salida de audio analógica, por lo cual es necesario incorporar dos placas de audio USB, para la captura y muestreo de la voz. Conectando cada placa de audio al dispositivo, se puede observar una detección automática por parte del sistema. Se deben editar archivos de ALSA para que la placa externa sea tomada por defecto por Raspbian. Por último, se llevan a cabo pruebas de grabación y reproducción para corroborar su funcionamiento.

Para un entendimiento más detallado sobre la configuración, referirse al Apéndice B de este trabajo.

Librería de audio: PortAudio

Dentro de una gama de librerías disponibles para la captura y reproducción de los datos se escoge la librería de audio PortAudio, ya que su característica principal es streaming de audio y se encuentra desarrollada sobre el lenguaje de programación C/C++.

PortAudio es una librería I/O de audio de uso libre y de código abierto, la cual permite desarrollar simples programas de audio en C o C++, y puede ser compilado y ejecutado en múltiples plataformas, a destacar Windows, UNIX y MAC OS [15].

Con la librería instalada, se utiliza el comando `ldconfig` para actualizar las nuevas librerías instaladas para que Raspbian las reconozca, luego reiniciar el sistema para efectuar los cambios.

Verificar que la cabecera `asoundlib.h` se encuentre en `/usr/include/sys/`. En algunas distribuciones también se encuentra en `/usr/include/alsa/`. Por seguridad, es recomendable colocarlo en ambos directorios.

Utilizando los ejemplos provistos por la librería de audio se llevan a cabo pruebas de funcionamiento sobre la Raspberry y la placa de audio USB.

Primero utilizando el ejemplo `paex_saw.c` se genera una señal seno donde en la salida de audio se puede escuchar un tono y por osciloscopio observar la señal analógica.

Luego a través del ejemplo `paex_record.c` se logra capturar audio durante una cantidad determinada de segundos y luego reproducirlo a través de altavoces.

Diezmado e Interpolación

Las placas de audio disponibles en el mercado manejan dos frecuencias de muestreo típicas en 48 kHz y 44.1 kHz, pero la digitalización de la voz humana se lleva a cabo con una resolución menor, a una frecuencia de 8 kHz, por lo cual es necesario llevar a cabo técnicas de diezmado e interpolación dentro del proceso.

Para implementar ambas funciones es necesario obtener una tasa de muestreo de 8 kHz, diezmando e interpolando el buffer original con muestras digitales en un factor de $M = 6$, ya que

$$\frac{Fs}{M} = 8\ kHz$$

donde $Fs = 48000\ Hz$. Realizando el proceso inverso se recupera una señal similar a la original a través de la interpolación.

A nivel práctico, al aplicar diezmado primero se debe pasar las muestras por un filtro anti-*aliasing* para eliminar frecuencias no deseadas. Luego del buffer con muestras se toma una muestra cada seis y se desecha el resto, con lo cual se genera un nuevo buffer de muestras seis veces menor que el original.

Para implementar la técnica de interpolación, se toma el buffer con muestras diezmadas y se van ubicando en un buffer seis veces mayor, cada seis posiciones de memoria una muestra, donde el resto del buffer se rellena con ceros. Posteriormente este nuevo buffer es pasado por un filtro pasa bajos para recuperar una señal similar a la de entrada.

Los algoritmos de diezmado e interpolación fueron desarrollados sobre el lenguaje de programación C, utilizando buffers con valores conocidos para llevar a cabo pruebas individuales sobre cada función.

Diseño de Filtros

El diseño de los filtros utilizados dentro de las funciones de diezmado e interpolación, anti-alias y pasa bajos, se llevó a cabo en dos etapas. Primero, implementar el algoritmo de funcionamiento básico de un filtro FIR en C [16]. Segundo, realizar el diseño de los filtros a través de la herramienta FDATool de MATLAB R2011a, para luego exportar los coeficientes cuantificados al proyecto.

A continuación se puede observar el código fuente de un filtro FIR:

```
double fir(M, h, w, x)        /* Usage: y = fir(M, h, w, x); */
double *h, *w, x;             /* h = filter, w = state, x = input sample */
int M;                        /* M = filter order */
{
       int i;
       double y;              /* output sample */

       w[0] = x;              /* read current input sample x */

       for (y=0, i=0; i<=M; i++)
              y += h[i] * w[i];       /* compute current output sample y */

       for (i=M; i>=1; i--)           /* update states for next call */
              w[i] = w[i-1];          /* done in reverse order */

       return y;
}
```

Figura 15: Algoritmo de filtro FIR
Extraído de "Introducción a los filtros digitales – filtros FIR"; cátedra DSP – UNC

La variable `X` pasa a ser el buffer con muestras de voz, el cual va dando en cada ciclo de trabajo una muestra al vector `W`, se lleva a cabo el producto entre los dos vectores `H` y `W` elemento a elemento, donde `H` es el vector con los coeficientes y `W` es un vector en el cual se van desplazando las muestras logrando implementar el proceso interno de funcionamiento de un filtro. Los resultados obtenidos son almacenados en la variable `Y`.

La herramienta FDATool proporciona una vía rápida y práctica para la generación de coeficientes, especificando parámetros tales como tipo de filtro, método de diseño, orden del filtro, frecuencias de corte y paso, atenuación en banda de corte y paso, mostrando diagramas de módulo y fase del filtro teórico.

A continuación se puede observar el diseño de los filtros sobre FDATool con las especificaciones necesarias, mostrando diagramas de módulo y fase.

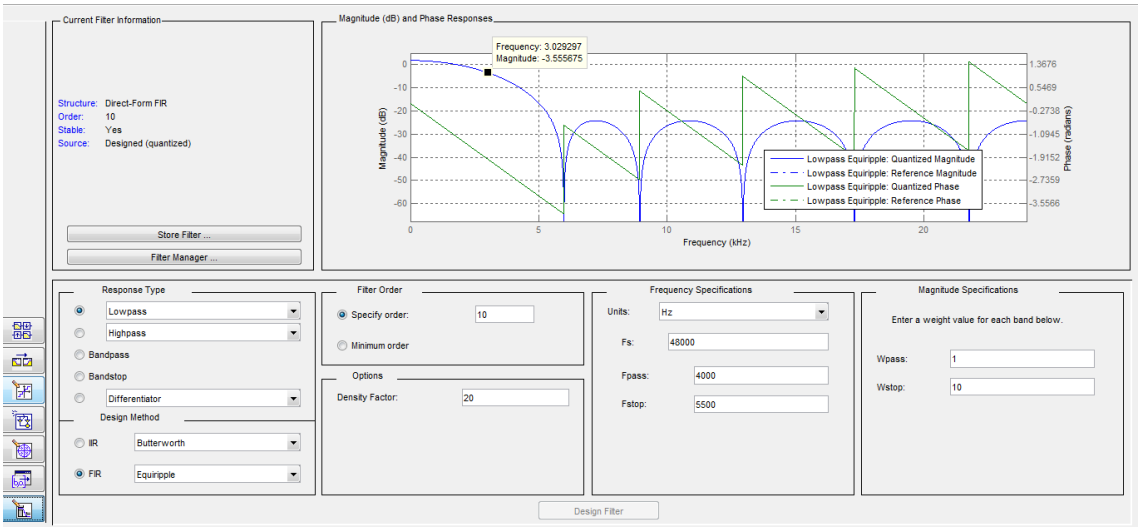

Figura 16: Captura de pantalla, diseño de filtro FIR pasa bajos sobre FDATool, MATLAB

Dentro de FDATool, para cuantificar los coeficientes generados, es necesario especificar la longitud en bits de los datos de entrada y salida del filtro, y si sus operaciones internas son de punto flotante o punto fijo.

Por lo tanto, los parámetros de diseño del filtro son:

- Filtro pasabajo FIR Equiripple
- 10 coeficientes
- Frecuencia de paso 4000 Hz
- Frecuencia de corte 5500 Hz

G.711

Para poder cumplir con las especificaciones de los canales telefónicos, es necesario incluir dentro del desarrollo el códec de audio G.711. La herramienta de software G.191 provee los códigos escritos en lenguaje C, con implementaciones de la recomendación G.711. A continuación se muestra la documentación de las funciones para comprimir y expandir según G.711 utilizando ley A. Dentro del desarrollo se emplea ley A ya que es utilizada dentro del sistema telefónico del país.

Compresión con ley A

La función `alaw_compress()` implementa la regla de codificación Ley A.

Las muestras de entrada deberán estar justificadas a la izquierda, y las muestras de salida son justificadas a la derecha con 8 bits.

Sintaxis

```
    #include "g711.h"
    void alaw_compress (long smpno, short *lin_buf, short
*log_buf)
```

Variables

- `smpno`: Es el número de muestras en `lin_buf`.
- `lin_buf`: Es el buffer de muestras de entrada; cada muestra del tipo `short` deberá contener muestras en PCM lineal (en complemento a 2 y 16 bits de ancho), justificadas a la izquierda.
- `log_buf`: Es el buffer de muestras de salida; cada muestra `short` contendrá las muestras en Ley A, de 8 bits de ancho y justificadas a la derecha.

Expansión con ley A

La función `alaw_expand()` implementa la regla de decodificación Ley A, acorde a la Recomendación G.711. Las muestras de salida estarán justificadas a la izquierda, y las muestras de entrada deberán estar justificadas a la derecha con 8 bits.

Sintaxis

```
    #include "g711.h"
    void alaw_expand (long smpno, short *log_buf, short
*lin_buf)
```

Variables

- `smpno`: Es el número de muestras en `log_buf`.
- `log_buf`: Es el buffer de muestras de entrada; cada muestra `short` deberá contener las muestras en Ley A, de 8 bits de ancho y justificadas a la derecha.
- `lin_buf`: Es el buffer de muestras de salida; cada muestra `short` contendrá las muestras en PCM lineal (en complemento a 2 y de 16 bits de ancho) justificadas a la izquierda.

Canal digital

Para establecer el canal de comunicaciones entre los dispositivos Raspberry Pi, se emplea una comunicación serial ya que simula el par trenzado de cobre UTP de la red telefónica, donde el propósito de este trabajo busca a futuro que el producto desarrollado pueda ser escalable a la red existente. En la Raspberry Pi esto es posible a través del puerto GPIO el cual cuenta con dos pines UART.

Para establecer el canal de voz entre ambas placas son necesarios pin 8 TX, pin 10 RX y cualquier pin GND del GPIO. Es necesario modificar archivos dentro de Raspbian para liberar ambos puertos, ya que por defecto en Raspberry Pi es configurado para ser utilizado por sesiones vía terminal. Realizados estos cambios es necesario reiniciar el sistema para tener el canal entre los dispositivos liberado.

El nivel o capa de enlace de datos es la responsable de la transferencia fiable de información a través de un circuito de transmisión de datos. Su objetivo es conseguir que la información fluya libre de errores entre dos dispositivos conectados directamente.

Este procedimiento se lleva a cabo usando la API de *POSIX terminal interface*, que contiene funciones para el envío y recepción de datos usando el puerto UART. Para poder acceder a la

API, se deben incluir dentro del proyecto las siguientes cabeceras, que se encuentran dentro de Raspbian por defecto:

```
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
```

Es necesario configurar los parámetros requeridos para iniciar la comunicación, esto se realiza mediante una estructura de datos, llamada `termios`, que contiene los siguientes elementos:

- `c_iflag`: detalla el control básico de acceso al terminal, para la cual solo utilizamos una bandera: `IGNPAR`. Esta bandera especifica que se ignoren los controles de paridad de la librería.
- `c_oflag`: especifica cómo debe ser tratada la salida del UART. Utilizamos la salida por defecto, es decir, `c_oflag=0`.
- `c_cflag`: define el control que el terminal va a ejercer sobre el hardware. Aquí se especifica: tamaño del caracter igual a 8 bits (`CS8`), ignorar el estado de la línea de módem (`CLOCAL`), activar la recepción (`CREAD`) y definir la velocidad de transmisión en 9600 Bps (`B9600`) para Diffie-Hellman o en 115200 Bps (`B115200`) para el canal de voz.
- `c_lflag`: este argumento es usado para varias configuraciones adicionales del terminal, las cuales no requerimos ninguna, por ende `c_lflag=0`.

Además, otras banderas deben ser asignadas para poder utilizar el API, las cuales están definidas en la librería `fcntl.h`:

- `O_RDWR`: modos de acceso, en lectura y escritura.
- `O_NDELAY`: modo sin bloqueo, para que las peticiones de lecturas devuelvan un mensaje de error en lugar de bloquear el sistema.
- `O_NOCTTY`: es necesario prevenir que el sistema operativo dé privilegios al programa sobre el control de los puertos, para evitar errores inesperados en el sistema.

Canal de voz no seguro

Para llevar a cabo la implementación del canal de voz, se toma como referencia el código fuente que provee la librería, `paex_read_write_wire.c` en el cual es posible capturar y reproducir la voz a través de dos funciones fundamentales dentro de PortAudio, las cuales son `Pa_WriteStream()` y `Pa_ReadStream()`.

El código fuente originalmente fue desarrollado para establecer un *streaming* de datos de entrada y salida en un dispositivo para la captura y reproducción de los datos en estéreo, por lo cual fue modificada la lógica para ser aplicado en este trabajo.

En el preprocesador, se definen las variables globales a utilizar:

- `SAMPLE_RATE` a 48000 Hz.
- `FRAMES_PER_BUFFER`, cantidad de tramas por buffer 6144.
- `NUM_SECONDS`, duración del proceso en segundos, a modo de ejemplo 15 segundos.
- `PA_SAMPLE_TYPE`, tipo de variable para el formato de las muestras por parte de PortAudio.

Dentro de la función principal, se definen e inicializan las variables locales del proceso:

`PaStreamParameters`, es una estructura que contiene parámetros para una dirección (entrada o salida) del *stream*, por lo cual se definen ambas estructuras de datos, `inputparameters` y `outputparameters`.

Se inicializan los campos de las estructuras de entrada y salida, especificando el dispositivo (placa de audio) a través de la función `Pa_GetDefaultInputDevice()` y `Pa_GetDefaultOutputDevice()`, o simplemente colocando 0, ya que anteriormente fue configurado Raspbian para tomar las placas de audio por defecto al ser conectadas a las Raspberry Pi; cantidad de canales, tipo de variable de muestreo.

Al momento de definir el tipo de variable del buffer de datos se escoge el tipo `short`, el cual es una variable de 16 bits con signo. Esto se debe a que las placas de audio manejan una resolución de 16 bits/muestra y PortAudio utiliza variables de tipo `paInt16`, enteros de 16 bits con signo.

Cabe aclarar que el tipo `short` es el mismo que `int16` o `short int`, de 16 bits respectivamente.

A lo largo del desarrollo del proyecto se busca manipular datos del mismo tipo de variable para evitar inconvenientes al momento de pasar de un tipo de variable a otro por temas de resolución de bits, salvo en la etapa de filtrado donde es necesario utilizar datos de tipo `float` (32 bits) para los cálculos ya que los coeficientes extraídos de MATLAB son números de punto flotante.

Las funciones empleadas de PortAudio para el *streaming* de I/O en el dispositivo son:

- `Pa_Initialize()` inicializa las estructuras de datos internas de la librería.
- `Pa_OpenStream()` abre el *stream* de datos para entrada y salida.
- `Pa_StartStream()` inicia el procesado de audio.
- `Pa_StopStream()` finaliza el procesado de audio, esperando que todos los buffers de audio sean reproducidos antes de su retorno.
- `Pa_Terminate()` desasigna todos los recursos que fueron asignados por PortAudio desde que fueron inicializados por la función `Pa_Initialize()`.

- `Pa_ReadStream()` lee muestras del *stream* de entrada, esta función no retorna hasta que el buffer este completamente lleno. Recibe como argumento el puntero que maneja las muestras, el puntero al buffer local para el almacenado y la cantidad de tramas del buffer.
- `Pa_WriteStream()` escribe las muestras en el *stream* de salida, esta función no retorna hasta haber escrito todos los datos. Recibe como argumento el puntero que maneja las muestras, el puntero al buffer local y la cantidad de tramas a ser escritas desde el buffer local.

Por lo cual fue posible establecer un canal de voz no seguro sin cifrar, a través del UART del GPIO de las Raspberry Pi. Generando un flujo de datos a una tasa de transferencia de 768 kbps, ya que:

$$16\ [bit/muestra]\ x\ 48000\ [muestra/s] = 768000\ [bit/s]$$

Luego incorporando al código las funciones de diezmado e interpolación desarrolladas anteriormente, se logra disminuir la tasa de muestreo a 4000 muestras/segundo, obteniendo una tasa de transferencia a 64 kbps:

$$16\ [bit/muestra]\ x\ 4000\ [muestra/s]\ =\ 64000\ [bit/s]$$

Para poder cumplir con las especificaciones detalladas sobre valores de tasas de trasmisión según la Recomendación de ITU-T, se incluyen dentro del proceso dos funciones de compresión y expansión según el códec G.711, logrando mantener la tasa de transferencia a 64 kbps pero bajo las condiciones de los canales telefónicos actuales.

$$8\ [bit/muestra]\ x\ 8000\ [muestra/s]\ =\ 64000\ [bit/s]$$

Reduciendo la calidad de las muestras a 8 bits y aumentando la tasa de muestreo a 8000 muestras/s, obtenemos nuevamente una tasa de 64 kbps.

Además se incorporan nuevos buffer de datos, ya que en un principio se maneja 6144 tramas por buffer, con una resolución de 16 bit/muestra, que es equivalente a 12288 bytes; pasando a manipular 1024 tramas por buffer y utilizando un factor de 6 en el diezmado e interpolación, lo que generó un buffer de 2048 bytes. Luego al incorporar el códec de audio se reordenan las muestras dentro de nuevos buffers de la mitad de tamaño para optimizar el proceso, de un tamaño de 512 tramas, 1024 bytes.

Canal de voz seguro

Al tener listo el canal de voz no seguro, es momento de incorporar las funciones para el intercambio de llave y cifrado.

Para el intercambio de llaves a través del protocolo Diffie-Hellman, se utilizó OpenSSL, la cual es una librería desarrollada para la implementación de los protocolos SSL y TLS. Es un proyecto altamente reconocido, sumamente completo, robusto, de nivel comercial y de código abierto.

El primer paso es generar los números p y g, ejecutando en terminal:

```
openssl dhparam -out file.txt -2 -c 256
```

Dónde `dhparam` es la función, `file.txt` es el archivo de salida donde se almacenarán las variables, `-2` es el valor de g, `-c` especifica que la salida sea compatible con código C, y `-256` es la cantidad de bytes deseados para la clave. El resultado es una función de código C llamada `get_dh1024()` que devuelve una estructura con todos los parámetros para iniciar protocolo:

```
DH *get_dh1024()
{
    static unsigned char dh1024_p[]={
        0x8C,0xF7,0xFC,0x24,0x30,0x49,0x3E,0x1B,0x6D,0xE3,0x57,0x05,
        0x67,0xBC,0xA9,0x60,0x58,0xB1,0xBD,0x84,0xDD,0xEB,0xE8,0xAD,
        0x69,0xDA,0x49,0xCC,0x49,0xB8,0x5D,0xB0,0x42,0xC7,0x11,0x25,
        0x8E,0xE9,0x7E,0x93,0xFB,0x5A,0x5C,0xB9,0xA0,0x89,0xFE,0x65,
        0x5A,0xF5,0xBE,0x8B,0x46,0x78,0x01,0xD9,0x1F,0x7D,0x15,0x9C,
        0xBD,0x35,0x62,0xF2,0x32,0xAB,0x1D,0xB7,0xA1,0x92,0x0C,0x76,
        0xD6,0x85,0x9D,0xB6,0xD7,0xDD,0x7E,0xA2,0xB8,0xDA,0xD1,0x9B,
        0x12,0x91,0xAA,0xAA,0x04,0x9A,0x67,0xD3,0x53,0x6E,0x8D,0x0C,
        0x93,0xFD,0x90,0x36,0x62,0x48,0xFB,0xFD,0xD4,0xE2,0xEC,0x62,
        0x96,0xEC,0xE6,0x52,0xBC,0x2A,0xC6,0x37,0x7F,0x4C,0x01,0xDD,
        0x78,0xA8,0x25,0x2F,0x65,0xB2,0xC9,0x3B};


    static unsigned char dh1024_g[] = {0x02};
    DH *dh;

    if ((dh=DH_new()) == NULL) return(NULL);
    dh->p = BN_bin2bn(dh1024_p, sizeof(dh1024_p), NULL);
    dh->g = BN_bin2bn(dh1024_g, sizeof(dh1024_g), NULL);
    if ((dh->p == NULL) || (dh->g == NULL))
    { DH_free(dh); return(NULL); }
    return(dh);
}
```

Figura 17: Resultado del proceso de inicialización de Diffie-Hellman en OPENSSL

Luego, el proceso es simple, se ejecuta el algoritmo de Diffie-Hellman, como se describió en el Capítulo 3. Para cada parte:

```
INICIALIZAR ALGORITMO DIFFIE HELLMAN
      GENERAR(a)
      A = DIFFIEHELLMAN(a, p, g)
TRANSMITIR(A)
RECIBIR(B)
k = GENERAR_LLAVE(B, a, p)
```

A partir de este momento, el *keystream* se encuentra grabado en `k` y es igual en ambas partes.

En la siguiente etapa hay que realizar todas las operaciones descritas en la sección RC4 del Capítulo 3, a través del algoritmo PRGA, y luego finalmente cifrar el buffer de voz a través de la operación XOR.

Comunicación Half-Duplex

Para que el usuario pueda alternar fácilmente entre modo transmisión y modo recepción, se incorporaron funciones teniendo en cuenta la simplicidad de uso principalmente. El objetivo es lograr alternar entre modos pulsando una tecla.

Primero, al no tener acceso a la librería `conio.h`, la cual sólo está disponible para Windows, se emularon las funciones `kbhit()` y `getch()` con las librerías disponibles para UNIX. Si el programa fuese compilado sobre Windows, detectaría la existencia de `conio.h` y omitiría nuestras funciones emuladas para usar las originales.

Luego, se incluyeron dos subrutinas `goto`, una para transmisión y otra para recepción. Ambas rutinas se ejecutan en bucle, comprobando al final que no esté la tecla de cambio de modo presionada. Si este fuera el caso, el programa altera de modo transmisión a recepción, o viceversa.

Se compiló el código final exitosamente en cada dispositivo Raspberry Pi, y se procedió a realizar pruebas sobre el sistema completo las cuales se encuentran en el capítulo siguiente.

CAPÍTULO 5: Validación y verificación

Según las especificaciones detalladas para el diseño del proyecto, a continuación se encuentran las pruebas realizadas sobre el sistema completo, las cuales son:

- Detección de errores de transmisión (Suma de comprobación)
- Latencia
- Ganancia
- Saturación
- Distorsión Armónica
- Uso de CPU

Durante el desarrollo, la incorporación del códec de voz se realizó luego de haber llevado a cabo mediciones de latencia y distorsión armónica, por lo que ambas instancias están incluidas para realizar un contraste y verificar los beneficios del códec.

Para las mediciones previas al códec, las especificaciones del sistema fueron:

- Baudrate: 115200 Bps
- Bitrate: 64 kbps
- Fs: 4000 muestras/s
- Resolución: 16 bits/muestra

E incluyendo el códec:

- Baudrate: 115200 Bps
- Bitrate: 64 kbit/s
- Fs: 8000 muestras/s
- Resolución: 8 bit/muestra

Suma de comprobación

Una suma de comprobación o *checksum*, es una función que tiene como propósito principal detectar cambios accidentales en una secuencia de datos para proteger la integridad de estos, verificando que no haya discrepancias entre los valores obtenidos al hacer una comprobación inicial y otra final tras la transmisión [17]. La idea es que se transmita el dato junto con su valor de *checksum*, de esta forma el receptor puede calcular dicho valor y compararlo así con el recibido. Si hay una discrepancia, significa que ha habido un error, y se procede a rechazar los datos y/o pedir una retransmisión.

Una vez que la llave ha sido generada con éxito desde Diffie-Hellman, se recurre a esta función para la comprobación de integridad de datos. El proceso es el siguiente:

- Sumar todos los valores de la llave consecutivamente para obtener un valor, a llamar `csum`.
- Intercambiar este valor con la contraparte.

- Si ambos `csum` son iguales, la probabilidad de error es prácticamente cero, por lo que se considera exitosa la trasmisión.
- Si fuesen distintos, se ha detectado la existencia de un error y se procede a realizar nuevamente el intercambio.

A continuación el pseudocódigo que detalla el proceso anterior:

```
csum = 0
PARA i = 0 HASTA i = largodellave HACER
      csum = csum + k[i]
FIN PARA
ENVIAR (csum)
RECIBIR (csum2)
SI csum = csum2 HACER
      IMPRIMIR (La llave ha sido transmitida exitosamente)
      FIN
SI NO HACER
      IMPRIMIR (Error, retransmitiendo...)
IR_A (TRANSMITIR_LLAVE)
```

Durante las pruebas, el 100% de las trasmisiones de llave fueron libres de errores. Sin embargo, esto se debió a que el ambiente controlado del laboratorio no tiene ningún tipo de interferencias. En el caso de que este proyecto llegase a desarrollarse comercialmente, el uso del algoritmo de suma de verificación se justificaría y cobraría mayor protagonismo.

Latencia

Se denomina latencia a la suma de retardos temporales dentro de un sistema [18].

El traspaso de información de un mecanismo a otro sufrirá siempre este retardo, que normalmente es estimado en milisegundos.

Sin el códec de voz

Para la medición de latencia del presente trabajo se optó por armar el sistema y medir la diferencia de tiempos de entrada y salida, en el osciloscopio TEKTRONIX TDS1001B, visualizando en CH1 la señal de entrada seno a una determinada frecuencia. En CH2 se observa la salida del sistema implementado, como puede apreciarse en la Figura 18.

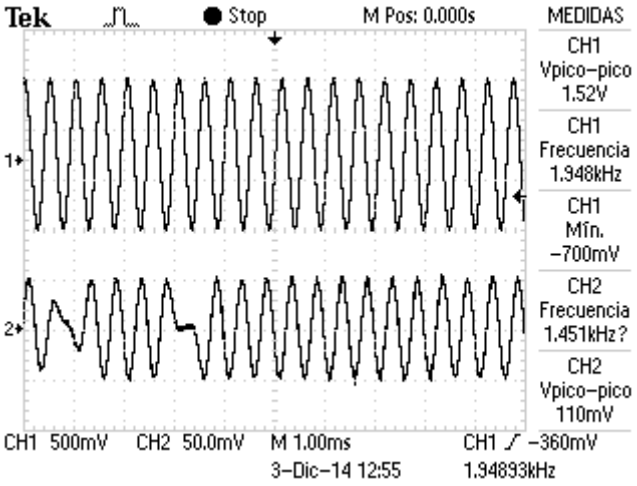

Figura 18: Entrada (CH1) y salida (CH2) del sistema

Para observar la latencia en la pantalla del osciloscopio se ajusta la base de tiempos a 100 ms/div y atenuando la señal de entrada a –60 dB, y luego quitando la atenuación se obtiene la diferencia con los dos cursores, como se observa en la Figura 19.

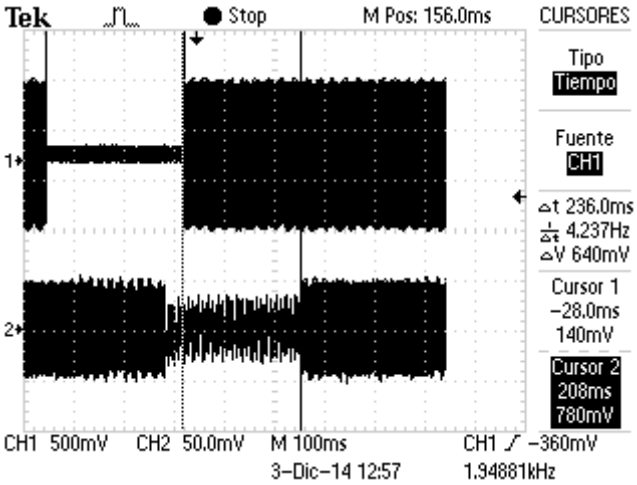

Figura 19: Retardo del sistema

Luego para obtener una medición más exacta se ajusta la base de tiempos a 50 ms/div como se observa en la Figura 20.

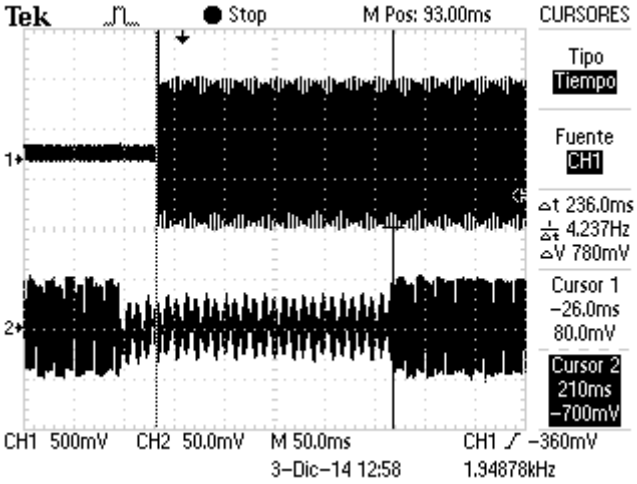

Figura 20: Retardo del sistema

Para obtener una medición de latencia promedio, se tomaron 30 muestras variando la frecuencia, se observa en la Tabla 4 las muestras obtenidas.

FRECUENCIA [Hz]	500	1000	1500	2000
MEDICIÓN [mseg]	228	226	170	278
	228	244	232	230
	241	254	232	230
	248	258	252	238
	254	264	256	238
	179	230	-	-
	242	230	-	-
	252	190	-	-
	256	168	-	-
	260	234	-	-

Tabla 4: Resultados de latencia a distintas frecuencias

Se calcularon los siguientes parámetros

- Promedio: $\mu = \sum \frac{Xi}{N}$, donde Xi representa a cada número, y N es el tamaño de la muestra.
- Desviación estándar. Esto representa el rango cubierto por tu conjunto de datos. Es igual a $\sigma = \sqrt{\sum \frac{(Xi-\mu)^2}{N}}$
- Error estándar (del promedio). Este dato indica qué tanto el promedio de la muestra se acerca al promedio real de la población de donde se extrajo el conjunto de datos. Mientras más larga sea la muestra, más pequeño será el error estándar, y más próximo estará el promedio de la muestra al promedio de la población. Puedes obtenerlo dividiendo la desviación estándar entre la raíz cuadrada de N, el tamaño de la muestra. Se calcula como $\frac{\sigma}{\sqrt{N}}$

Por lo tanto los resultados obtenidos fueron:

- Muestras: $N = 30$
- Promedio: $\mu = \sum \frac{Xi}{N} = 234.73\ ms$
- Desviación estándar: $\sigma = \sqrt{\sum \frac{(Xi-\mu)^2}{N}} = 26.17\ ms$
- Error: $\frac{\sigma}{\sqrt{N}} = 4.77\ ms$

Con el códec de voz

Al incorporar el códec de audio G.711 dentro del sistema fue necesario llevar a cabo las mediciones de latencia nuevamente.

Similarmente, se ajustó la base de tiempos a 250 ms/div se atenuó la señal de entrada a –60dB, luego quitando la atenuación se obtiene la diferencia con los dos cursores, se aprecia en la Figura 21.

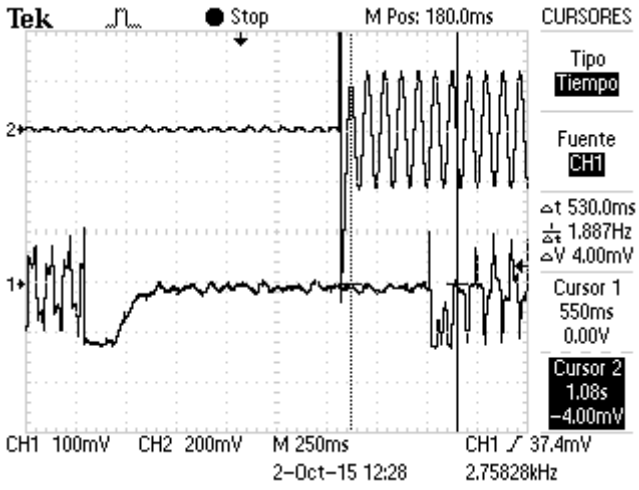

Figura 21: Retardo del sistema

Para esta nueva medición se tomaron 18 muestras variando la frecuencia.

FRECUENCIA [Hz]	1000	2000	3000
MEDICIÓN [mseg]	410	460	420
	500	410	440
	470	450	490
	460	480	500
	510	520	450
	480	470	510

Tabla 5: Conjunto de muestras de latencia

En este caso los resultados fueron:

- Muestras: $N = 18$
- Promedio: $\mu = \sum \frac{Xi}{N} = 468.33\ mseg$
- Desviación estándar: $\sigma = \sqrt{\sum \frac{(Xi-\mu)^2}{N}} = 33.04\ mseg$
- Error: $\frac{\sigma}{\sqrt{N}} = 7.78\ mseg$

Ganancia

Para llevar a cabo las mediciones de ganancia o atenuación se utilizó un generador de señales con una señal seno a la entrada del sistema, en 2 kHz y posteriormente en 3 kHz, y un osciloscopio TEKTRONIX TBS 1001B midiendo entrada y salida del mismo.

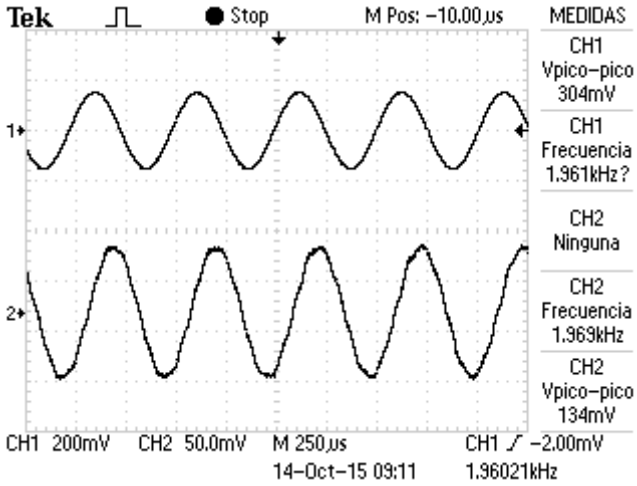

Figura 22*:* Medición de ganancia

En la figura anterior, se observa la señal de entrada sobre CH1 y la señal de salida sobre CH2 con la señal de prueba en 2 kHz, podemos obtener un valor de ganancia tomando un cociente entre los valores de tensión de entrada y salida.

$$G = \frac{V_{out}}{V_{in}}$$

Por lo cual la ganancia del sistema es la siguiente,

$$G = \frac{134\ mV}{304\ mV} = 0{,}44$$

Observando la siguiente figura donde la señal de prueba se encuentra con una frecuencia de 3 kHz, la ganancia del sistema disminuye.

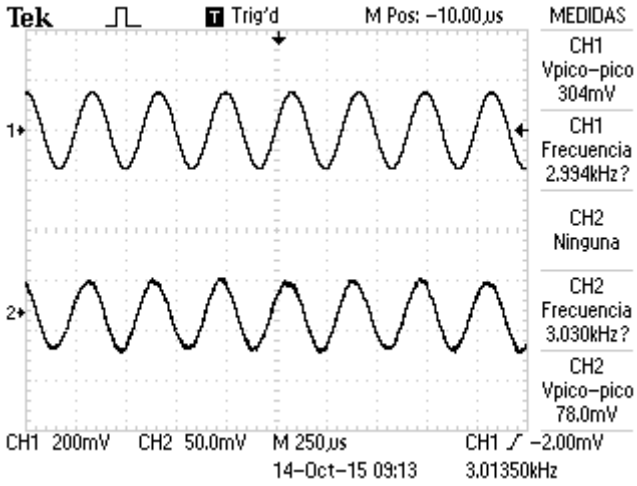

Figura 23: Medición de ganancia

Bajo estas condiciones la ganancia del sistema es la siguiente

$$G = \frac{78\ mV}{360\ mV} = 0.25$$

Queda en evidencia la atenuación de la señal de salida al aumentar la frecuencia de la señal de prueba debido a los valores de atenuación en bandas de supresión de los filtros pasa-bajos empleados dentro del sistema.

Saturación

La saturación dentro de un sistema es un factor importante, ya que establece los límites máximos de tensión para las señales de entrada. Para llevar a cabo la medición se ingresa al sistema con una señal de prueba a una frecuencia fija aumentando la tensión de entrada y observando en el osciloscopio la respuesta a la salida.

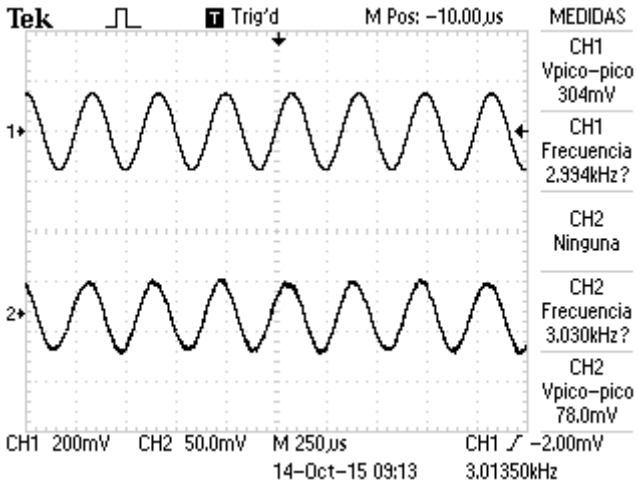

Figura 24*:* Medición de saturación del sistema

En la figura anterior, la señal de entrada sobre CH1, posee una tensión pico a pico de 300mV. Incrementando la tensión de la señal de prueba por encima de estos valores, la salida comienza a saturarse, y es visible en el osciloscopio como las crestas de la señal se van aplanando a medida que aumenta la amplitud de entrada.

Distorsión Armónica Total

La Distorsión Armónica Total (THD - *Total Harmonic Distortion*) es un parámetro técnico utilizado para definir la señal de audio que sale de un sistema. Se produce cuando la señal de salida de un sistema no equivale a la señal que ingresó en él. Esta falta de linealidad afecta a la forma de la onda, porque el equipo ha introducido armónicos que no estaban en la señal de entrada. Puesto que son armónicos, es decir múltiplos de la señal de entrada, esta distorsión no es tan disonante y es más difícil de detectar.

Se define la THD de la siguiente manera:

$$THD = \frac{\sum Potencia\ de\ los\ armónicos}{Potencia\ de\ la\ frecuencia\ fundamental} = \frac{P_1 + P_2 + P_3 + \cdots + P_N}{P_0}$$

donde P_0 es la potencia del tono fundamental y P_i con $i > 0$ es la potencia del armónico i-ésimo que contiene la señal. Todas las medidas de potencia se realizan en la salida del sistema, mediante un filtro paso banda y un osciloscopio o bien mediante un analizador de espectro.

Las mediciones se llevaron a cabo dentro del laboratorio de Sistemas Embebidos del Instituto Universitario Aeronáutico, observando el espectro de salida del sistema con la herramienta FFT del Osciloscopio TEKTRONIX TDS 1001b. Están sujetas al margen de error que introduce la herramienta FFT del osciloscopio, al no haber podido utilizar un Analizador de Espectro.

Con una señal conocida, señal seno, en la entrada del sistema se fue barriendo en frecuencia y observando a la salida en el osciloscopio utilizando una amplitud al 50% de la señal máxima de saturación para obtener una correcta medición.

Sin códec de voz

Las mediciones se llevan a cabo con una tasa de muestreo de 4 kHz/muestra, con una señal seno de entrada a 115200 Bps, utilizando un ventaneo tipo Hanning.

Las escalas utilizadas en las Figuras 25-29 en eje vertical son de 10 dB/div y eje horizontal 500 Hz/div. En las Tablas 6–10 se observan los valores obtenidos durante la medición.

Se toma como referencia para el cálculo el nivel de 0 dB sobre el eje horizontal marcado por M, con 0 dB = 10 mV_{rms}.

500 Hz

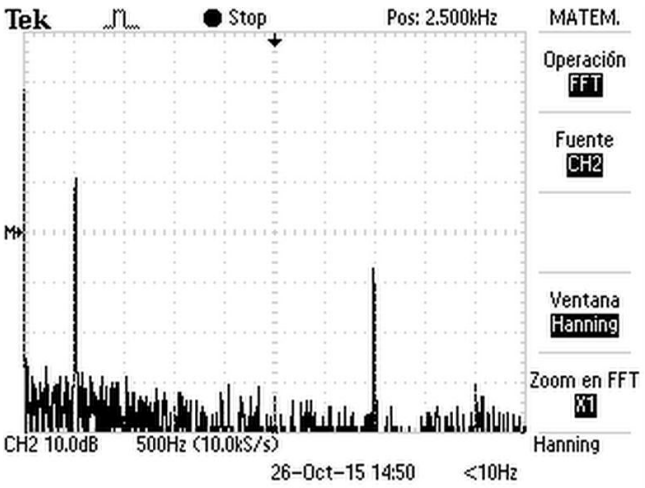

Figura 25: Espectro de frecuencias en 500 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	500	11
1er armónico	3500	-7
THD		12.58%

Tabla 6: Mediciones de distorsión. Espectro de frecuencias en 500 Hz

1000 Hz

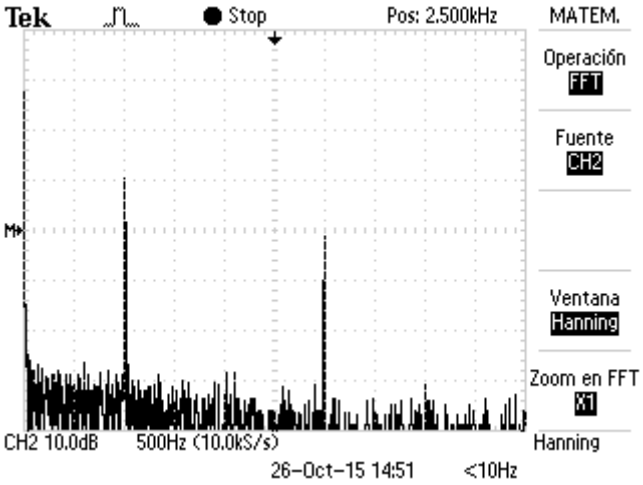

Figura 26: Espectro de frecuencias en 1000 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	1000	11
1er armónico	3000	0
THD		28.18%

Tabla 7: Mediciones de distorsión. Espectro de frecuencias en 1000 Hz

1500 Hz

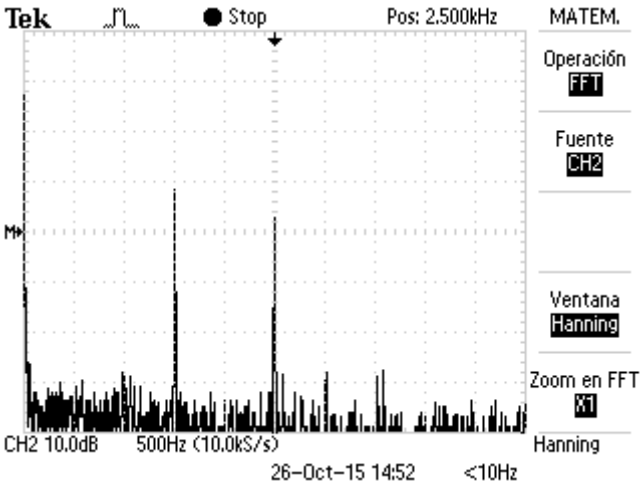

Figura 27: Espectro de frecuencias en 1500 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	1500	8
1er armónico	2500	3
THD		56.25%

Tabla 8: Mediciones de distorsión. Espectro de frecuencias en 1500 Hz

2000 Hz

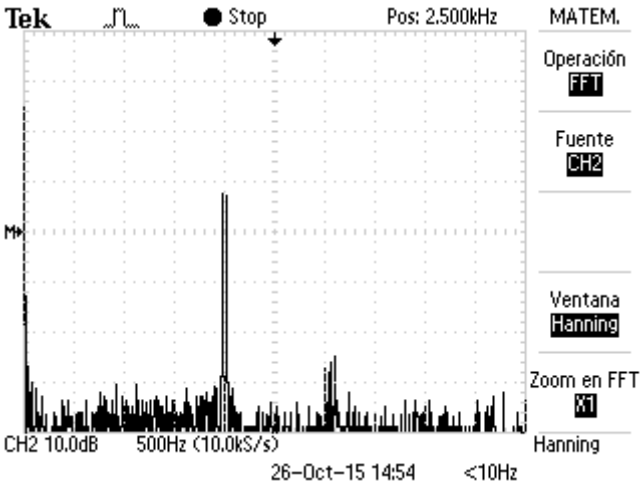

Figura 28: Espectro de frecuencias en 2000 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	2000	8
1er armónico	3000	-24
2do armónico	3100	-27
THD		4.26%

Tabla 9: Mediciones de distorsión. Espectro de frecuencias en 2000 Hz

2500 Hz

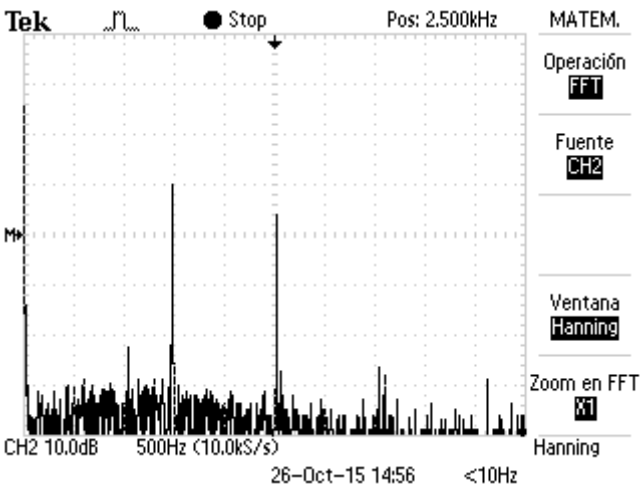

Figura 29: Espectro de frecuencias en 2500 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	2500	4
1er armónico	1500	10
2do armónico	1000	-22
THD		204%

Tabla 10: Mediciones de distorsión. Espectro de frecuencias en 2500 Hz

Conclusiones sin el uso del códec

Al observar las mediciones obtenidas (Figuras 25-29 y Tablas 6-10) se puede concluir que el canal de voz posee una calidad aceptable a una tasa de muestreo de 4 kHz/muestra con una tasa de transmisión de 115200 Bps, donde se comprueba el correcto funcionamiento de los filtros implementados durante el tratamiento de datos, ya que se utiliza una tasa de muestreo del doble de la frecuencia máxima de las señales de voz filtradas. Las mediciones sobre una frecuencia de 2500 Hz arroja esos valores de distorsión, ya que los filtros utilizados durante esas pruebas tenían una frecuencia de corte en 2000 Hz.

A continuación se presentan las mediciones finales sobre THD:

Frecuencia [Hz]	THD [%]
500	12.58
1000	28.18
1500	56.25
2000	4.26
2500	204

Tabla 11: Mediciones finales de distorsión armónica

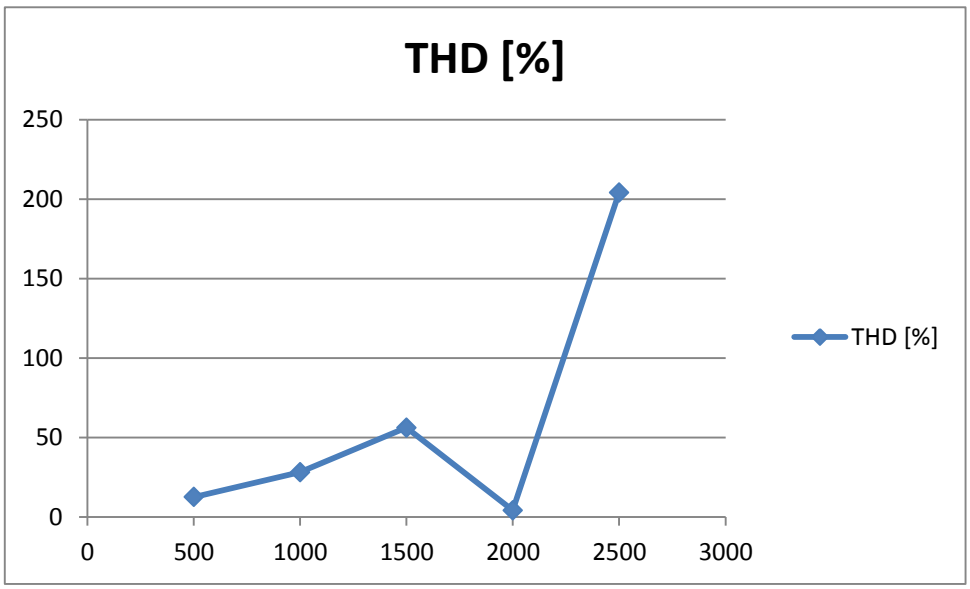

Figura 30: Gráfico comparativo de distorsión armónica

Con el códec de voz

Con una señal de prueba en la entrada del sistema, se va barriendo en frecuencia y observando el espectro de frecuencias en el osciloscopio. Las mediciones están sujetas al margen de error que introduce la herramienta FFT del osciloscopio al no haber podido utilizar un analizador de espectro.

Las escalas utilizadas en las Figuras 31-36 en eje vertical son de 10 dB/div y eje horizontal 500 Hz/div. En las Tablas 12–17 se observan los valores obtenidos durante la medición.

Se toma como referencia para el cálculo el nivel de 0 dB sobre el eje horizontal marcado por M, con 0 dB = 10 mV_{rms}.

Mediciones

500Hz

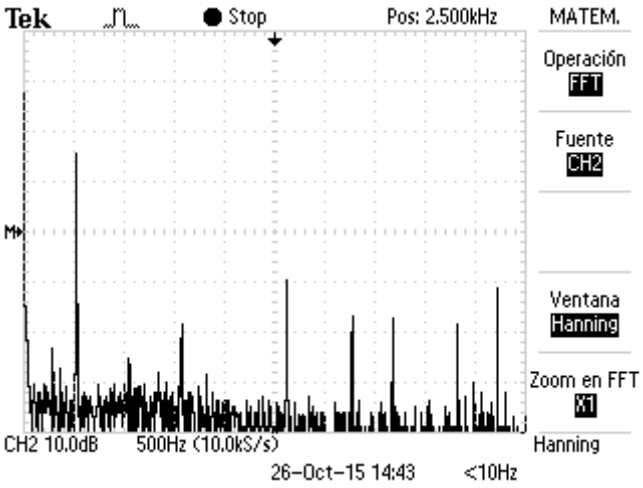

Figura 31*:* Espectro de frecuencias en 500 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	500	16
1er armónico	2600	-10
2do armónico	3300	-16
3ro armónico	1600	-18
4to armónico	300	-24
THD		10.49%

Tabla 12*:* Espectro de frecuencias en 500 Hz

1000Hz

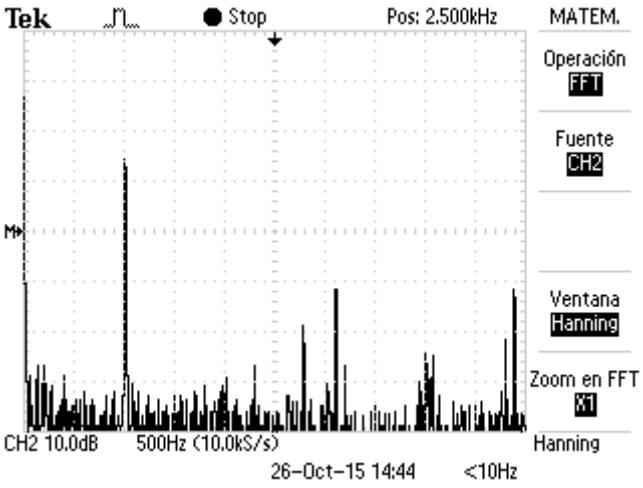

Figura 32*:* Espectro de frecuencias en 1000 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	1000	14
1er armónico	3100	-12
2do armónico	2700	-18
3ro armónico	2300	-26
THD		8.50%

Tabla 13: Espectro de frecuencias en 1000 Hz

1500 Hz

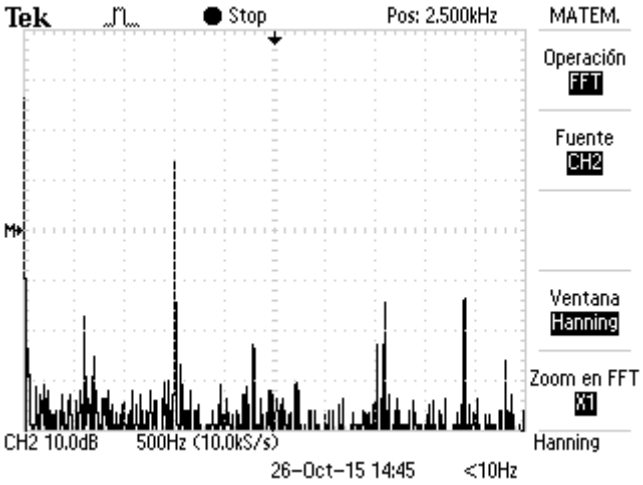

Figura 33*:* Espectro de frecuencias en 1500 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	1500	14
1er armónico	600	-17
2do armónico	2300	-23
3ro armónico	700	-26
THD		5.20%

Tabla 14: Espectro de frecuencias en 1500 Hz

2000 Hz

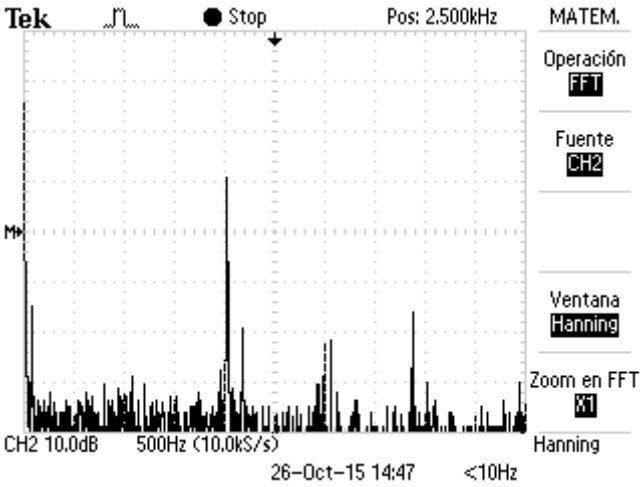

Figura 34: Espectro de frecuencias en 2000 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	2000	11
1er armónico	100	-14
2do armónico	2200	-19
3ro armónico	3000	-22
THD		10.99%

Tabla 15: Espectro de frecuencias en 2000 Hz

2500 Hz

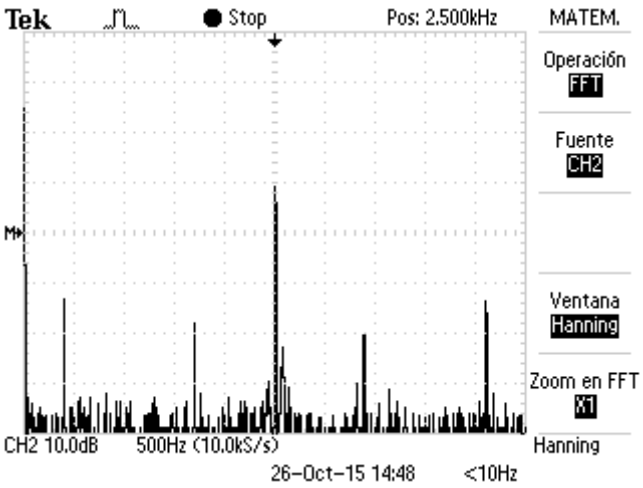

Figura 35: Espectro de frecuencias en 2500 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	2500	10
1er armónico	400	-13
2do armónico	1700	-18
3ro armónico	3400	-20
THD		14.16%

Tabla 16: Espectro de frecuencias en 2500 Hz

3000 Hz

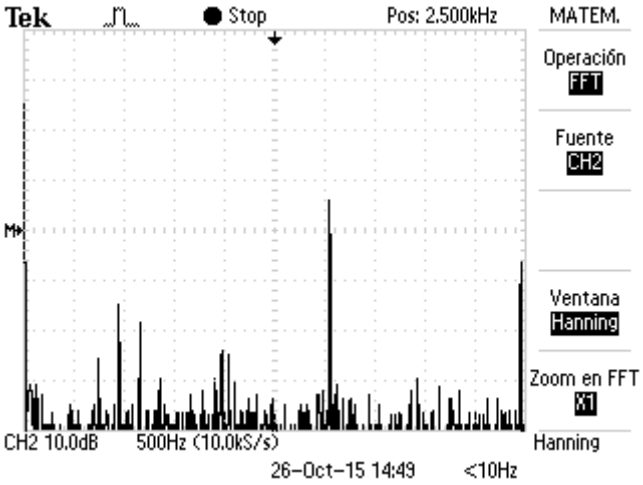

Figura 36*:* Espectro de frecuencias en 3000 Hz

Componentes	Frecuencia [Hz]	Potencia [dB]
Fundamental	3000	6
1er armónico	900	-15
2do armónico	1200	-18
3ro armónico	2000	-24
4to armónico	700	-26
THD		20.8%

Tabla 17: Espectro de frecuencias en 3000 Hz

Conclusiones con el uso del códec

Al observar las mediciones obtenidas (Figuras 31-36 y Tablas 12-17) se puede concluir que el canal de voz disminuye notablemente los valores de distorsión dentro del sistema al incluir el códec de audio G.711 a una tasa de muestreo de 8000 muestras/s con una tasa de transmisión de 115200 Bps, obteniendo una calidad de voz tal que es posible reconocer al locutor en cada uno de los extremos.

A continuación se presentan las mediciones finales sobre THD en la Tabla 18:

Frecuencia [Hz]	THD [%]
500	10.49
1000	8.50
1500	5.20
2000	10.99
2500	14.16
3000	20.8

Tabla 18: Mediciones finales de distorsión armónica

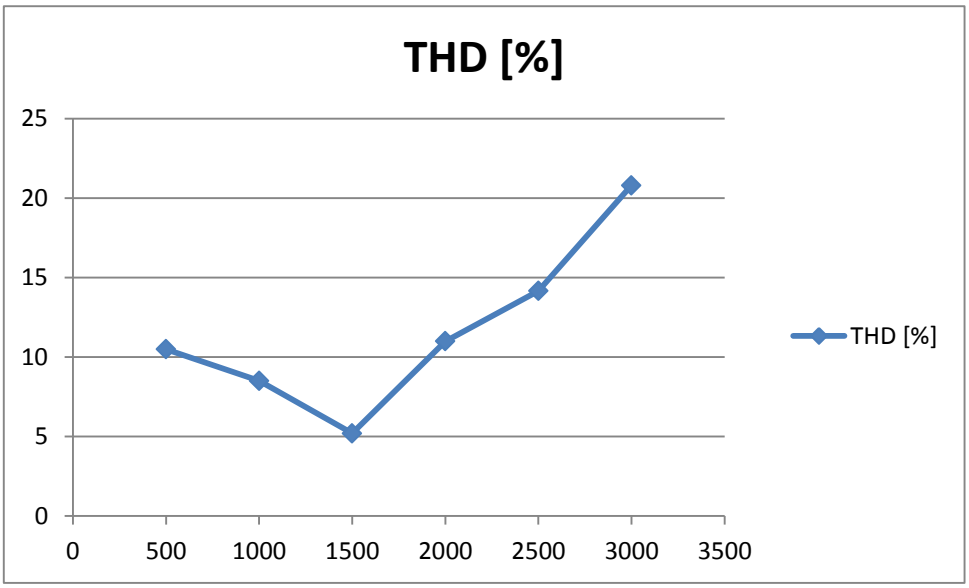

Figura 37: Gráfico comparativo de distorsión armónica

Uso de CPU

El consumo de CPU por parte del proceso es un parámetro importante a tener en cuenta. Durante la ejecución de la aplicación, en cada extremo del sistema dentro de un terminal, se observan los porcentajes de uso del procesador a través del comando `top`.

En la Figura 38, se observa el uso de CPU en el dispositivo transmisor. Durante un lapso de 20 minutos el terminal muestra un consumo promedio de 10% sin grandes fluctuaciones.

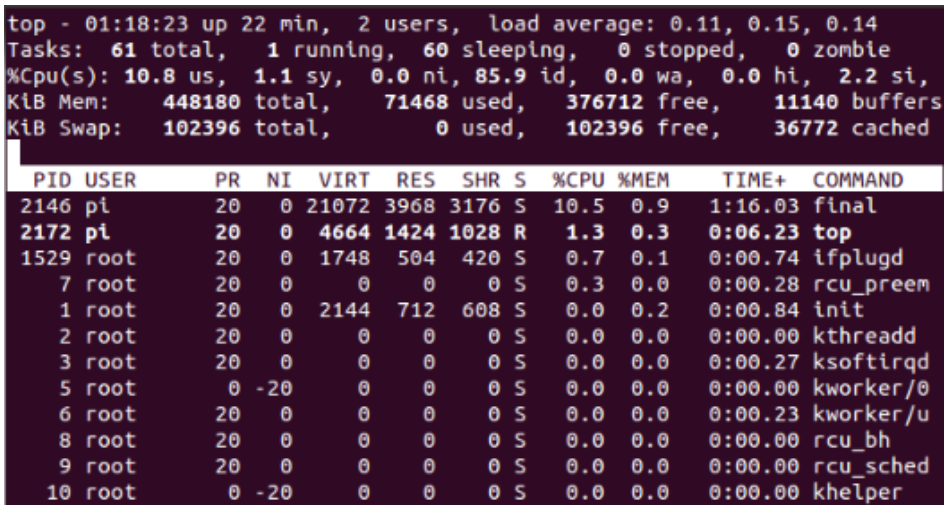

Figura 38: Uso de CPU del dispositivo transmisor

En la Figura 39, se observa el uso de CPU del dispositivo receptor. Al inicio de la medición el proceso presentaba un uso del 88%, que luego fue disminuyendo a 40%, manteniendo este valor con una variación de +/- 5%.

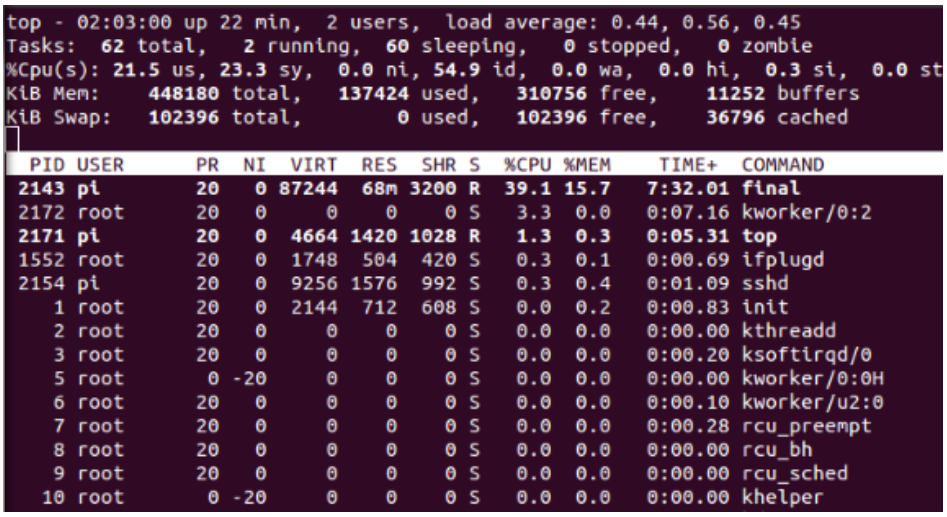

Figura 39: Uso de CPU del dispositivo receptor

CAPÍTULO 6: Conclusiones

El principal objetivo de este trabajo final de grado fue la implementación, utilizando sistemas embebidos, de una comunicación de voz segura entre dos entidades desconocidas entre sí, poniendo como requisito la escalabilidad del canal a la red telefónica subyacente.

Se concluye que los objetivos propuestos fueron alcanzados logrando implementar un sistema mediante técnicas de procesamiento digital de señales y cifrado de información, obteniendo resultados aceptables frente a las pruebas de rendimiento.

Como posibles mejoras, sería deseable la evolución de la comunicación a modo bidireccional simultáneo, es decir, full-duplex, utilizando procesamiento multi-hilos para un uso más eficiente del procesador. Otra mejora sería la compresión de información para disminuir las tasas de transmisión obteniéndose menores valores de ancho de banda.

APÉNDICE A: Matemática

Logaritmo discreto

Sea a un entero cualquiera que sea primo relativo con n y g una raíz primitiva de n, entonces existe en el grupo $\{0, 1, 2, \dots, \phi(n) - 1\}$, donde $\phi(n)$ es la función indicatriz de Euler, exactamente una solución μ tal que

$$a \equiv g^{\mu} \ (mod\ n)$$

El número μ es llamado el logaritmo discreto de a con base g y módulo n y se denota [19]

$$\mu = ind_g a \ (mod\ n)$$

Raíz Primitiva

La raíz primitiva de un número primo p es un entero, tal que $g\ (mod\ p)$ tiene un orden multiplicativo $p - 1$. Gereralizando, si $\mathrm{MCD}(g, n) = 1$, es decir, g y n son primos relativos, y g es de orden multiplicativo $\phi(n)\ (mod\ n)$, donde $\phi(n)$ es la función indicatriz de Euler, entonces g es una raíz primitiva de n. La primera definición es un caso especial de la segunda, ya que $\phi(p) = p - 1$ para p primo.

Si n tiene una raíz primitiva, entonces tiene exactamente $\phi\big(\phi(n)\big)$ de ellas, lo que significa que si p es un número primo, hay exactamente $\phi(p - 1)$ raíces primitivas incongruentes de p. Para $n = \{1, 2, 3, \dots\}$ los primeros valores de $\phi\big(\phi(n)\big)$ son $\{1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 4, 2, 4, 2, 4, 4, 8, \dots\}$. n tiene una raíz primitiva de la forma 2, 4, p^a, o $2p^a$, donde p es un primo impar y $a \geq 1$. Los pocos primeros números para los que las raíces primitivas existen son $\{2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 14, 17, 18, 19, 22, \dots\}$, entonces, el número de raíces primitivas de orden n para $n = \{1, 2, 3, \dots\}$ son $\{0, 1, 1, 1, 2, 1, 2, 0, 2, 2, 4, 0, 4, \dots\}$.

Las raíces primitivas para los primeros números primos p son $\{1, 2, 2, 3, 2, 2, 3, 2, 5, 2, 3, 2, 6, 3, 5, 2, 2, 2, \dots\}$. Aquí hay una tabla de los primeros pocos n para los que una raíz primitiva existe:

n	$g(n)$
2	1
3	2
4	3
5	2, 3
6	5
7	3, 5
9	2, 5
10	3, 7
11	2, 6, 7, 8
13	2, 6, 7, 11

Tabla 19: Primeros números con raíz primitiva

Las raíces primitivas más grandes para $n = \{1, 2, 3, \dots\}$ son $\{0, 1, 2, 3, 3, 5, 5, 0, 5, 7, 8, 0, 11, \dots\}$. Las raíces primitivas más pequeñas para los primeros enteros n están dadas en la tabla siguiente:

n	$g(n)$	n	$g(n)$	n	$g(n)$	n	$g(n)$
2	1	38	3	94	5	158	3
3	2	41	6	97	5	162	5
4	3	43	3	98	3	163	2
5	2	46	5	101	2	166	5
6	5	47	5	103	5	167	5
7	3	49	3	106	3	169	2
9	2	50	3	107	2	173	2
10	3	53	2	109	6	178	3
11	2	54	5	113	3	179	2
13	2	58	3	118	11	181	2
14	3	59	2	121	2	191	19
17	3	61	2	122	7	193	5
18	5	62	3	125	2	194	5
19	2	67	2	127	3	197	2
22	7	71	7	131	2	199	3
23	5	73	5	134	7	202	3
25	2	74	5	137	3	206	5
26	7	79	3	139	2	211	2
27	2	81	2	142	7	214	5
29	2	82	7	146	5	218	11
31	3	83	2	149	2	223	3
34	3	86	3	151	6	226	3
37	2	89	3	157	5	227	2

Tabla 20: Raíces primitivas más pequeñas para los n correspondietes

Sea p cualquier número primo impar tal que $k \geq 1$, y sea

$$s \equiv \sum_{j=1}^{p-1} j^k$$

Entonces

$$s = \begin{cases} -1 \ (mod\ p) & para\ \ p-1 \mid k \\ 0 \ (mod\ p) & para\ p-1 \nmid k \end{cases}$$

Para números m con raíces primitivas, todas las y que satisfagan $(m, y) = 1$ son representables como

$$y \ \equiv \ g^t \ (mod\ m),$$

Donde $t = \{0, 1, \dots, \phi(m) - 1\}$, t es conocido como el índice, e y es un entero. Kearnes (1984) demostró que para cualquier entero positivo m, existen infinitos primos p tales que

$$m \ < g_p < p - m$$

Tomando la última raíz primitiva g_p, Burgess (1962) demostró que

$$g_p \leq C\ p^{\frac{1}{4}+\epsilon}$$

para C y ϵ constantes positivas, y p suficientemente grande.

Matthews (1976) obtuvo la fórmula para las constantes de Artrin bidimensionales para un grupo de primos para el cual m y n son ambos raíces primitivas [20].

APÉNDICE B: Procedimientos

A continuación se detallan los comandos y procedimientos UNIX utilizados dentro del proyecto.

Comandos

- `sudo apt-get update` – Actualización de paquetes
- `sudo apt-get upgrade` – Actualización de software
- `lsusb` – Muestra dispositivos USB conectados
- `sudo reboot` – Reinicio del sistema
- `sudo alsamixer` – Configuración de nivel de ganancia de cada puerto
- `sudo ldconfig` – Actualización de librerías de Raspbian

Procedimientos

Grabar Sistema Operativo Raspbian

- Descargar software Win32DiskImager.
- Insertar tarjeta SD la en la PC y ejecutar el software.
- Seleccionar la imagen de Raspbian dentro del sistema y la letra que Windows asignó a nuestra tarjeta.
- Pulsamos en Write para dar inicio al proceso.

Configurar IP estática sobre Raspberry Pi

Para configurar los parámetros de red se debe dirigir a

```
$ sudo nano /etc/network/interfaces
```

Accediendo al archivo de configuración de red del sistema, agregar las siguientes líneas utilizando direcciones estáticas necesarias:

```
auto eth0
iface eth0 inet static

address         200.100.50.2
gateway         200.100.50.1
netmask         255.255.255.0
network         200.100.50.0
broadcast       200.100.50.255
```

Guardamos el archivo con sus cambios y reiniciamos los parámetros de red. Luego reiniciar parámetros de red ejecutando la línea de comando:

```
$ sudo /etc/init.d/networking restart
```

Conexión via SSH con Raspberry Pi

A través del protocolo SSH se accede remotamente vía terminal introduciendo:

```
$ ssh usuario@ipdispositivo
```

donde `usuario` es pi e `ipdispositivo` es 200.100.50.2.

```
$ ssh pi@200.100.50.2
```

Por último Password: *raspberry*.

Configurar placa de audio USB sobre Raspbian

Para configurar la placa de sonido por defecto, se debe editar el archivo alsa-base.conf. A través del comando

```
$ sudo nano /etc/modprobe.d/alsa-base.conf
```

Se debe comentar la siguiente línea guardar cambios y cerrar archivo.

```
# options snd-usb-audio index=-2
```

Reiniciar el dispositivo *y* verificar la configuración.

Grabación y reproducción con ALSA

Prueba de reproducción a través de salidas

```
$ aplay /usr/share/scratch/Media/Sounds/Vocals/Singer1.wav\
```

Prueba de grabación a través de entradas

```
$ arecord -vv -f dat -d 10 test.wav
```

Instalación de PortAudio sobre Raspberry Pi

Para instalar PortAudio sobre la plataforma Raspberry Pi, es necesario descargar los paquetes de librerías de ALSA ya que Raspbian no lo trae por defecto.

En un Terminal dentro del directorio */home/pi* colocamos los siguientes comandos:

```
    wget ftp://ftp.alsa-project.org/pub/lib/alsa-lib-
1.0.25.tar.bz2
    tar jxf alsa-lib-1.0.25.tar.bz2
    cd alsa-lib-1.0.25
    ./configure
    make
    make install
    make clean
```

Es necesario descargar las siguientes librerías complementarias, pueden ser instaladas todas en conjunto separando con espacios cada una de ellas en la línea de comandos:

```
libasound2
libasound2-dev
libasound2-plugins
alsa-utils
libportaudio2
libgtk2.0-dev
libsamplerate0-dev
libsndfile1-dev
subversion
```

Se deben instalar por separado:

```
portaudio19-dev
libportaudio-dev
```

Por último instalar PortAudio v19 o descargarlo desde el sitio web oficial:

```
http://www.portaudio.com/
```

Compilar programas con PortAudio

Para compilar archivos de PortAudio `*.c` a través del compilador GCC utilizar la siguiente línea de comando:

```
$ gcc archivo.c -o  BIN -lrt -lpthread -lasound -ljack -lm
-lportaudio
```

En la línea anterior la palabra en mayúscula será el binario ejecutable, es necesario incluir dentro de la línea de comando todas las dependencias de la librería de audio o cualquier otra librería incluida dentro del proyecto para una correcta compilación, ya que el sistema recurre a esos directorios para buscar e incluir las cabeceras dentro del código fuente.

Liberar puertos UART en Raspberry Pi

Para deshabilitar que la información de encendido sea enviada al puerto al inicio del sistema, a través del siguiente comando se debe ingresar al archivo y lo editarlo.

```
$ sudo nano /boot/cmdline.txt
```

El contenido del archivo debe ser el siguiente

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200
kgdboc=ttyAMA0,115200 console=tty1 root=/dev/mmcblk0p2
rootfstype=ext4 elevator=deadline rootwait
```

donde se debe remover el texto resaltado en rojo y luego guardar cambios efectuados.

Luego se debe editar el archivo `/etc/inittab` para deshabilitar el acceso del Terminal en el puerto serial. Dentro del archivo buscar la siguiente línea y comentarla con # al inicio de la misma. Guardamos los cambios y salimos del archivo.

```
T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

APÉNDICE C: Código Fuente

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <assert.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include "portaudio.h"

#include <openssl/dh.h>
#include <openssl/bn.h>
#include <openssl/ossl_typ.h>

#define SERIAL (FRAMES_PER_BUFFER/FACTOR)
#define SAMPLE_RATE  (48000)
#define FRAMES_PER_BUFFER (6144)
#define NUM_CHANNELS    (2)
#define NUM_SECONDS     (15)
#define FACTOR          (6)
#define DITHER_FLAG     (0)

#define KEYSIZE (128)

#define TURN 1

#define CHECK_OVERFLOW  (0)
#define CHECK_UNDERFLOW  (0)
#define PA_SAMPLE_TYPE  paInt16
#define SAMPLE_SIZE (2)
#define SAMPLE_SILENCE  (0)
#define CLEAR(a) memset( (a), 0,  FRAMES_PER_BUFFER )
#define CLEARF(a) memset( (a), 0,  FRAMES_PER_BUFFER/FACTOR )
#define PRINTF_S_FORMAT "%d"

#define H_SIZE 11
#define Y_SIZE (FRAMES_PER_BUFFER + H_SIZE - 1)


#if !defined(linux)
#include <conio.h>

#else

static struct termios g_old_kbd_mode;

static void cooked(void)
{
       tcsetattr(0, TCSANOW, &g_old_kbd_mode);
}

static void raw(void)
{
       static char init;

       struct termios new_kbd_mode;
```

```
	if(init)
		return;

	tcgetattr(0, &g_old_kbd_mode);
	memcpy(&new_kbd_mode, &g_old_kbd_mode, sizeof(struct termios));
	new_kbd_mode.c_lflag &= ~(ICANON | ECHO);
	new_kbd_mode.c_cc[VTIME] = 0;
	new_kbd_mode.c_cc[VMIN] = 1;
	tcsetattr(0, TCSANOW, &new_kbd_mode);
	atexit(cooked);

	init = 1;
}

static int kbhit(void)
{
	struct timeval timeout;
	fd_set read_handles;
	int status;

	raw();
	FD_ZERO(&read_handles);
	FD_SET(0, &read_handles);
	timeout.tv_sec = timeout.tv_usec = 0;
	status = select(0 + 1, &read_handles, NULL, NULL, &timeout);
	if(status < 0)
	{
		printf("select() failed in kbhit()\n");
		exit(1);
	}
	return status;
}

static int getch(void)
{
	unsigned char temp;

	raw();
	if(read(0, &temp, 1) != 1)
		return 0;
	return temp;
}

#endif

short *diezmar(short *);
short *interpolar(short *);

void char2short(unsigned char*, unsigned short*);

void swapper(unsigned short*, int, int);
void rc4(unsigned short*, unsigned short*, unsigned short*, int);
DH *get_dh1024();
unsigned char *diffiehellman();
void serial_tx(unsigned char*, int);
void serial_rx(unsigned char*, int);
void cs_tx(unsigned long*, int);
void cs_rx(unsigned long*, int);
BIGNUM *sharekeys(const BIGNUM*);
void printer(const short*, int);
void printer2(const unsigned char*, int);
void char2short(unsigned char*, unsigned short*);
int checksum(unsigned short*);

void  alaw_compress(long lseg, short *linbuf, short *logbuf);
```

```
void  alaw_expand(long lseg, short *logbuf, short *linbuf);

void optimizar(short *Sin, unsigned short *Sout);
void desoptimizar(unsigned short *Sin, short *Sout);


int main()
{
    PaStreamParameters inputParameters, outputParameters;
    PaStream *stream = NULL;
    PaError err;
    short
*Sblock,*Sblock2,*Sblock3,Sb[FRAMES_PER_BUFFER]={0},Sb2[FRAMES_PER_BUFFER/FACT
OR]={0},Sb3[FRAMES_PER_BUFFER/FACTOR]={0};
    unsigned short
*Sblock4,*Sblock5,*Sblock6,Sb4[FRAMES_PER_BUFFER/(FACTOR*2)]={0},Sb5[FRAMES_PE
R_BUFFER/(FACTOR*2)]={0},Sb6[FRAMES_PER_BUFFER/(FACTOR*2)]={0};
    unsigned short *rx_buffer, *rewind;
    rx_buffer = (short*)malloc(8 * sizeof(short));

       Sblock = Sb;
       Sblock2 = Sb2;
       Sblock3 = Sb3;
       Sblock4 = Sb4;
       Sblock5 = Sb5;
       Sblock6 = Sb6;
       rewind = Sblock6;

       unsigned short key[KEYSIZE/2] = {0};
       unsigned short* pkey = key;
       unsigned char* ckey = (unsigned char*) malloc(KEYSIZE/2);
       char ptt = 0;

       while(1)
       {
              ckey = diffiehellman();
              char2short(ckey, pkey);
              if(checksum(pkey) == 0)
                     {
                     printf("\nLlave compartida exitosamente.\n");
                     break;
                     }
              else
                     {
                     printf("\nSe detecto un error en la transmision de la
llave. Retransmitiendo...\n");
                     if(TURN == 1)
                            sleep(1);
                     }
       }

       printf("\nLlave dsp de char2short:");
       printer2(ckey, KEYSIZE/2);

    fflush(stdout);
    CLEAR( Sblock );

    err = Pa_Initialize();
    if( err != paNoError ) goto error;

    inputParameters.device = 0;
    printf( "DispositivH_SIo de entrada # %d.\n", inputParameters.device );
    printf( "Input LL: %g s\n", Pa_GetDeviceInfo( inputParameters.device )-
>defaultLowInputLatency );
    printf( "Input HL: %g s\n", Pa_GetDeviceInfo( inputParameters.device )-
```

```
>defaultHighInputLatency );
    inputParameters.channelCount = 1;
    inputParameters.sampleFormat = paInt16;
    inputParameters.suggestedLatency = Pa_GetDeviceInfo(
inputParameters.device )->defaultHighInputLatency ;
    inputParameters.hostApiSpecificStreamInfo = NULL;

    outputParameters.device = 0;
    printf( "Dispositivo de salida # %d.\n", outputParameters.device );
    printf( "Output LL: %g s\n", Pa_GetDeviceInfo( outputParameters.device )-
>defaultLowOutputLatency );
    printf( "Output HL: %g s\n", Pa_GetDeviceInfo( outputParameters.device )-
>defaultHighOutputLatency );
    outputParameters.channelCount = 1;
    outputParameters.sampleFormat = paInt16;
    outputParameters.suggestedLatency = Pa_GetDeviceInfo(
outputParameters.device )->defaultHighOutputLatency;
    outputParameters.hostApiSpecificStreamInfo = NULL;

       int uart0_filestream = -1;
       int counter = 0,count;

       uart0_filestream = open("/dev/ttyAMA0", O_RDWR | O_NOCTTY | O_NDELAY);
       if (uart0_filestream == -1)
       {
              printf("Error - No se pudo abrir el UART.\n");
       }

       struct termios options;
       tcgetattr(uart0_filestream, &options);
       options.c_cflag = B115200 | CS8 | CLOCAL | CREAD;
       options.c_iflag = IGNPAR;
       options.c_oflag = 0;
       options.c_lflag = 0;
       tcflush(uart0_filestream, TCIFLUSH);
       tcsetattr(uart0_filestream, TCSANOW, &options);

   err = Pa_OpenStream(
              &stream,
              &inputParameters,
              &outputParameters,
              SAMPLE_RATE,
              FRAMES_PER_BUFFER,
              paClipOff,
              NULL,
              NULL );
    if( err != paNoError ) goto error;

    err = Pa_StartStream( stream );
    if( err != paNoError ) goto error;

    printf("\n> CANAL ABIERTO:");

    if(TURN == 1)
    {
       printf("\nModo TX");
       goto tx;
    }
    else
    {
       printf("\nModo RX");
       goto rx;
    }

rx:
```

```
    printf("\nModo RX");

       CLEAR(Sblock);
       CLEARF(Sblock2);
       CLEARF(Sblock3);
       CLEARF(Sblock4);
       CLEARF(Sblock5);
       CLEARF(Sblock6);

       while(uart0_filestream != -1)
              {
                     int rx_length = read(uart0_filestream, rx_buffer, 8);
                     int i;

                     if (rx_length < 0)
                     {
                            //Waiting
                     }
                     else if (rx_length == 0)
                     {
                            //No data waiting
                     }
                     else
                     {
                            counter += rx_length;
                            for(i = 0; i < rx_length/2; i++)
                            {
                                   *rewind = rx_buffer[i];
                                   rewind++;
                            }

                            if(counter == SERIAL+1 || counter == SERIAL)
                            {
                                   rc4(Sblock6, pkey, Sblock5,
FRAMES_PER_BUFFER/(FACTOR*2));
                                   desoptimizar(Sblock5,Sblock3);
                                   alaw_expand((long)(FRAMES_PER_BUFFER/FACTOR),
Sblock3, Sblock2);
                                   Sblock = interpolar(Sblock2);

                                   err = Pa_WriteStream( stream, Sblock,
FRAMES_PER_BUFFER );
                                          if( err && CHECK_OVERFLOW ) goto xrun;
                                          counter = 0;
                                   rewind = Sblock6;
                            }
                     }
                     if(kbhit())
                            if(getch() == 32)
                                   goto tx;
              }

tx:
       printf("\nModo TX");

       CLEAR(Sblock);
       CLEARF(Sblock2);
       CLEARF(Sblock3);
       CLEARF(Sblock4);
       CLEARF(Sblock5);
       CLEARF(Sblock6);

       while(1)
    {
              if(kbhit())
```

```
                        if(getch() == 32)
                                goto rx;
                err = Pa_ReadStream( stream, Sblock, FRAMES_PER_BUFFER );
                if( err && CHECK_UNDERFLOW ) goto xrun;

                Sblock2 = diezmar(Sblock);
                alaw_compress((long)(FRAMES_PER_BUFFER/FACTOR), Sblock2,
Sblock3);
                optimizar(Sblock3,Sblock4);
                rc4(Sblock4, pkey, Sblock6, FRAMES_PER_BUFFER/(FACTOR*2));

                if (uart0_filestream != -1)
                {
                        count = write(uart0_filestream, Sblock6, SERIAL);

                        if (count < 0)
                        {
                                printf("UART TX error!\n");
                        }
                }
        }


        err = Pa_StopStream( stream );
        if( err != paNoError ) goto error;

        CLEAR( Sblock );
           Sblock = Sb;

        Pa_Terminate();
        return 0;

xrun:
        if( stream ) {
           Pa_AbortStream( stream );
           Pa_CloseStream( stream );
        }
        Pa_Terminate();
        if( err & paInputOverflow )
           fprintf( stderr, "Input Overflow.\n" );
        if( err & paOutputUnderflow )
           fprintf( stderr, "Output Underflow.\n" );
        return -2;

error:
        if( stream ) {
           Pa_AbortStream( stream );
           Pa_CloseStream( stream );
        }
        Pa_Terminate();
        fprintf( stderr, "Ocurrio un error durante el uso de portaudio stream\n"
);
        fprintf( stderr, "Error numero: %d\n", err );
        fprintf( stderr, "Error msj: %s\n", Pa_GetErrorText( err ) );
        return -1;
}

short *diezmar(short *x)
{
int i,j,k=0,l=0;
float  h[H_SIZE]={0.05953216552734375,
0.07391357421875,
0.1060028076171875,
0.1345977783203125,
0.15438079833984375,
```

```
0.16144561767578125,
0.15438079833984375,
0.1345977783203125,
0.1060028076171875,
0.07391357421875,
0.05953216552734375};
float w[H_SIZE]={0}, y[Y_SIZE]={0};
short *ptrbkup,z[Y_SIZE],bkup[FRAMES_PER_BUFFER/FACTOR];
ptrbkup = bkup;

    for(i = 0; i < Y_SIZE; i++)
    {
        if (i < FRAMES_PER_BUFFER)
            w[0] = *(x+i);
        else
            w[0] = 0;

        for (j = 0; j < H_SIZE ;j++)
            y[i] +=  h[j] * w[j];

        for(j = H_SIZE-1; j >= 0 ; j--)
            w[j] = w[j-1];

    }

for(i=0 ; i<FRAMES_PER_BUFFER ; i++)
{
if (y[i] > 32767) y[i] = 32767;
else if (y[i] < -32768) y[i] = -32768;
z[i] = (short)y[i];
}

for(i=0 ; i<FRAMES_PER_BUFFER ; i += FACTOR)
{
 *(bkup+j) = *(z+i);
 j++;
}
return ptrbkup;
}

short *interpolar(short *x)
{
int i,j=0;
float  h[H_SIZE]={0.05953216552734375,
0.07391357421875,
0.1060028076171875,
0.1345977783203125,
0.15438079833984375,
0.16144561767578125,
0.15438079833984375,
0.1345977783203125,
0.1060028076171875,
0.07391357421875,
0.05953216552734375};
float w[H_SIZE]={0}, y[Y_SIZE]={0};
short *ptrz, z[Y_SIZE],*buf;

buf = (short*)malloc(FRAMES_PER_BUFFER * sizeof(short));
ptrz = z;

for(i=0 ; i<FRAMES_PER_BUFFER/FACTOR ; i++)
{
buf[j] = *(x+i);
j+=FACTOR;
}
```

```
    for(i=0; i<Y_SIZE; i++)
    {
        if (i < FRAMES_PER_BUFFER)
            w[0] = buf[i];
        else
            w[0] = 0;

        for (j = 0; j < H_SIZE ;j++)
            y[i] +=  h[j] * w[j];

        for(j = H_SIZE-1; j >= 0 ; j--)
            w[j] = w[j-1];
    }

for(i=0 ; i < FRAMES_PER_BUFFER ; i++)
{
if (y[i] > 32767) y[i] = 32767;
else if (y[i] < -32768) y[i] = -32768;
z[i] = (short)y[i];
}
return ptrz;
}

void rc4(unsigned short* msj, unsigned short* key, unsigned short* cipher, int
msj_size)
{
       unsigned short s[256];
       int i, j, a;
       unsigned short k;

       for(i = 0; i < 256; i++)
              s[i] = i;

       for(i = 0, j = 0; i < 256; i++)
       {
              j = (j + s[i] + key[i % KEYSIZE/4]) % 256;
              swapper(s, i, j);
       }

       for(a = 0, i = 0, j = 0; a < msj_size; a++)
       {
              i = (i + 1) % 256;
              j = (j + s[i]) % 256;
              swapper(s, i, j);
              k = s[ (s[i] + s[j]) % 256];
              cipher[a] = msj[a] ^ k;
       }
}

void swapper(unsigned short* s, int i, int j)
{
       unsigned short temp = s[j];
       s[j] = s[i];
       s[i] = temp;
}

DH *get_dh1024()
{
    static unsigned char dh1024_p[]={
              0x8C,0xF7,0xFC,0x24,0x30,0x49,0x3E,0x1B,0x6D,0xE3,0x57,0x05,
              0x67,0xBC,0xA9,0x60,0x58,0xB1,0xBD,0x84,0xDD,0xEB,0xE8,0xAD,
              0x69,0xDA,0x49,0xCC,0x49,0xB8,0x5D,0xB0,0x42,0xC7,0x11,0x25,
              0x8E,0xE9,0x7E,0x93,0xFB,0x5A,0x5C,0xB9,0xA0,0x89,0xFE,0x65,
              0x5A,0xF5,0xBE,0x8B,0x46,0x78,0x01,0xD9,0x1F,0x7D,0x15,0x9C,
```

```
            0xBD,0x35,0x62,0xF2,0x32,0xAB,0x1D,0xB7,0xA1,0x92,0x0C,0x76,
            0xD6,0x85,0x9D,0xB6,0xD7,0xDD,0x7E,0xA2,0xB8,0xDA,0xD1,0x9B,
            0x12,0x91,0xAA,0xAA,0x04,0x9A,0x67,0xD3,0x53,0x6E,0x8D,0x0C,
            0x93,0xFD,0x90,0x36,0x62,0x48,0xFB,0xFD,0xD4,0xE2,0xEC,0x62,
            0x96,0xEC,0xE6,0x52,0xBC,0x2A,0xC6,0x37,0x7F,0x4C,0x01,0xDD,
            0x78,0xA8,0x25,0x2F,0x65,0xB2,0xC9,0x3B};


       static unsigned char dh1024_g[] = {0x02};
       DH *dh;

       if ((dh=DH_new()) == NULL) return(NULL);
       dh->p = BN_bin2bn(dh1024_p, sizeof(dh1024_p), NULL);
       dh->g = BN_bin2bn(dh1024_g, sizeof(dh1024_g), NULL);
       if ((dh->p == NULL) || (dh->g == NULL))
    { DH_free(dh); return(NULL); }
       return(dh);
}

unsigned char *diffiehellman()
{
    DH *dh;
    dh = get_dh1024();
    unsigned char *shared_key = (unsigned char *) malloc(DH_size(dh));
    int sizeofkey;
    int i;

    if(DH_generate_key(dh) == 0)
    {
        printf("\nDH_generate_key delvolvi%c un c%cdigo de error", 162, 162);
        exit(EXIT_FAILURE);
    }
    printf("\nPar%cmetros privados generados:\n\tP = %s\n\tG = %s", 160,
BN_bn2hex(dh->p), BN_bn2hex(dh->g));

    const BIGNUM *public_bob = sharekeys(dh->pub_key);

    sizeofkey = DH_compute_key(shared_key, public_bob, dh);
    if(sizeofkey == -1)
    {
        printf("\n\nDH_compute_key delvolvi%c un codig%c de error", 162, 162);
        exit(EXIT_FAILURE);
    }

    printf("\n\nClave privada de %d bytes generada con %cxito: ", sizeofkey,
130);
    printer2(shared_key, KEYSIZE/2);

    return shared_key;
}

BIGNUM *sharekeys(const BIGNUM *pub_alice)
{
    BIGNUM *pub_bob;
    pub_bob = BN_new();
    unsigned char* tx_data;
    tx_data = (unsigned char*)malloc(2*KEYSIZE * sizeof(unsigned char));
    tx_data = BN_bn2hex(pub_alice);

    unsigned char* rx_data;
    rx_data = (unsigned char*)malloc(2*KEYSIZE * sizeof(unsigned char));

    if(TURN == 1)
    {
             serial_tx(tx_data, 2*KEYSIZE);
```

```
		serial_rx(rx_data, 2*KEYSIZE);
	}

	else
	{
		serial_rx(rx_data, 2*KEYSIZE);
		sleep(1);
		serial_tx(tx_data, 2*KEYSIZE);
	}

	BN_hex2bn(&pub_bob, rx_data);
	return pub_bob;
}

void serial_rx(unsigned char* data, int size)
{
	unsigned char* rx_buffer;
	rx_buffer = (unsigned char*)malloc(8 * sizeof(unsigned char));
	unsigned char* data_bup = data;


	int uart0_filestream = -1;
	int counter = 0, i;

	uart0_filestream = open("/dev/ttyAMA0", O_RDWR | O_NOCTTY | O_NDELAY);
	if (uart0_filestream == -1)
	{
		printf("Error - No se pudo abrir el UART.\n");
	}

	struct termios options;
	tcgetattr(uart0_filestream, &options);
	options.c_cflag = B9600 | CS8 | CLOCAL | CREAD;
	options.c_iflag = IGNPAR;
	options.c_oflag = 0;
	options.c_lflag = 0;
	tcflush(uart0_filestream, TCIFLUSH);
	tcsetattr(uart0_filestream, TCSANOW, &options);

		while(uart0_filestream != -1)
		{
			int rx_length = read(uart0_filestream, (void*)rx_buffer,
8);
			if (rx_length < 0)
			{
				//Waiting
			}
			else if (rx_length == 0)
			{
				//No data waiting
			}
			else
			{
				counter += rx_length;
				for(i = 0; i < rx_length; i++)
				{
					*data_bup = rx_buffer[i];
					data_bup ++;
				}
				if(counter == size)
				{
					break;
				}
			}
		}
```

```
	close(uart0_filestream);
}

void serial_tx(unsigned char* tx_buffer, int size)
{

	int uart0_filestream = -1;
	int i, count;

	uart0_filestream = open("/dev/ttyAMA0", O_RDWR | O_NOCTTY | O_NDELAY);

	if (uart0_filestream == -1)
		printf("Error - Unable to open UART.  Ensure it is not in use by
another application\n");

	struct termios options;
	tcgetattr(uart0_filestream, &options);
	options.c_cflag = B9600 | CS8 | CLOCAL | CREAD;
	options.c_iflag = IGNPAR;
	options.c_oflag = 0;
	options.c_lflag = 0;
	tcflush(uart0_filestream, TCIFLUSH);
	tcsetattr(uart0_filestream, TCSANOW, &options);

	if (uart0_filestream != -1)
	{
			count = write(uart0_filestream, tx_buffer, size);
		if (count < 0)
		{
			printf("UART TX error!\n");
		}

	}
	close(uart0_filestream);
}

void char2short(unsigned char* pchar, unsigned short* pshort)
{
	int i;
	unsigned char* auxchar = pchar;
	unsigned short* auxshort = pshort;

	for(i = 0; i < KEYSIZE/2; i++)
	{
	*auxshort = (auxchar[0] << 8) | auxchar[1];
	auxshort++;
	auxchar += 2;
	}
}

int checksum(unsigned short* key)
{
	int i, n;
	unsigned long sum = 0;
	unsigned char csum[10] = {0}, csum_mirror[10] = {0}, result;
	for(i = 0; i < KEYSIZE; i++)
		sum += key[i];
	printf("\nChecksum: %lu\n", sum);

	n = sprintf(csum, "%lu", sum);

    if(TURN == 1)
    {
	sleep(2);
		serial_tx(csum, n);
```

```
                serial_rx(csum_mirror, n);
        }

        else
        {
                serial_rx(csum_mirror, n);
                sleep(1);
                serial_tx(csum, n);
        }

        printf("\nn: %d\ncsum: %s\ncsum_mirror: %s", n, csum, csum_mirror);

        if(strcmp(csum, csum_mirror) == 0)
                return 0;
           else
                return -1;
}

void optimizar(short *Sin, unsigned short *Sout)
{
       int i;
       unsigned char *auxchar,*pchar; auxchar=pchar;
       unsigned short *auxshort = Sout;

       for(i=0 ; i<FRAMES_PER_BUFFER/FACTOR ; i++)
       {
       auxchar[i] = (char)Sin[i];
       }
       for(i = 0; i < FRAMES_PER_BUFFER/(FACTOR*2); i++)
       {
       *auxshort = (auxchar[0] << 8) | auxchar[1];
       auxshort++;
       auxchar += 2;
       }
}

void desoptimizar(unsigned short *Sin, short *Sout)
{
       int i;
       char swap[2]={0};
       unsigned short *auxps1 = Sin;
       short *auxps2 = Sout;

       for(i=0; i < FRAMES_PER_BUFFER/FACTOR; i++)
       {
                memcpy(&swap, auxps1, 2);
                *auxps2=swap[1];
                auxps2++;
                *auxps2=swap[0];
                auxps1++;
                auxps2++;
       }
}

void alaw_compress(long lseg, short *linbuf, short *logbuf)
{
  short           ix, iexp;
  long            n;

  for (n = 0; n < lseg; n++)
  {
    ix = linbuf[n] < 0
      ? (~linbuf[n]) >> 4
      : (linbuf[n]) >> 4;
```

```
    if (ix > 15)
    {
      iexp = 1;
      while (ix > 16 + 15)
      {
       ix >>= 1;
       iexp++;
      }
      ix -= 16;

      ix += iexp << 4;
    }
    if (linbuf[n] >= 0)
      ix |= (0x0080);

    logbuf[n] = ix ^ (0x0055);
  }
}

void alaw_expand(long lseg, short *logbuf, short *linbuf)
{
  short           ix, mant, iexp;
  long            n;

  for (n = 0; n < lseg; n++)
  {
    ix = logbuf[n] ^ (0x0055);

    ix &= (0x007F);
    iexp = ix >> 4;
    mant = ix & (0x000F);
    if (iexp > 0)
      mant = mant + 16;

    mant = (mant << 4) + (0x0008);

    if (iexp > 1)
      mant = mant << (iexp - 1);

    linbuf[n] = logbuf[n] > 127
      ? mant
      : -mant;
  }
}

void printer(const short* text, int size)
{
    int i;

    for(i = 0; i < size; i++)
    {
        if(i % 16 == 0)
            printf("\n");
        printf("\t%hd", text[i]);
    }
    printf("\n\n");
}

void printer2(const unsigned char* text, int size)
{
       int i;

       for(i = 0; i < size; i++)
       {
              if(i % 16 == 0)
```

```
                printf("\n");
            printf("\t%X", text[i]);
        }
        printf("\n\n");
}
```

Glosario

AES: Advanced Encryption Standard – Estándar de cifrado avanzado.
Aliansing: Efecto que causa que señales continuas distintas se tornen indistinguibles cuando se muestrean digitalmente.
ALSA: API de audio para Linux.
API: Application Programming Interface – Interfaz de programación de aplicaciones.
ARM: Advanced RISC Machine – Máquina RISC avanzada.
C/C++: C y C++ son lenguajes de programación.
CCITT: Consultative Committee for International Telegraphy and Telephony – Comité Consultivo Internacional Telegráfico y Telefónico.
Coder-decoder: Codificador-decodificador.
CPU: Central Processing Unit – Unidad de procesamiento central.
DAC: Digital Analog Converter – Conversor Digital Analógico
DES: Data Encryption Standard – Estándar de cifrado de datos.
DSP: Digital Signal Processing – Procesamiento digital de señales.
FDATool: Filter Desing Analysis Tool – Herramienta de diseño y análisis de filtros. Viene incluido en MATLAB.
FIR: Finite Impulse Response – Respuesta finita al impulso. Un tipo de filtros digitales cuya respuesta a una señal impulso como entrada tendrá un número finito de términos no nulos.
Full-Duplex: Modo de envío de información bidireccional y simultáneo.
GND: GRouNd – Tierra.
GPIO: General Purpose Input/Output – Entrada/Salida de propósito general.
Half-Duplex: Modo de envío de información bidireccional pero no simultáneo.
I/O: Input/Output – Entrada/Salida.
INA219: Integrado monitor de energía desarrollado por Texas Instrument.
ISAKMP: Internet Security Association and Key Management Protocol – Protocolo de intercambio de llave definido en el RFC 2408.
KSA: Key Scheduling Algorithm – Algoritmo programador de llave.
LAN: Local Area Network – Red de área local.
MAC OS: Machintosh Operative System – Sistema Operativo Machintosh.
MATLAB: MATrix LABoratory, es una herramienta de software matemático que ofrece un entorno de desarrollo integrado con un lenguaje de programación propio.
MIM: Man In the Middle – Hombre en el medio. Es un ataque en el que se adquiere la capacidad de leer, insertar y modificar a voluntad los mensajes entre dos partes sin que ninguna de ellas conozca que el enlace entre ellos ha sido violado.
Monoclave: Uso de una única clave.
MQV: Menezes-Qu-Vanstone – Protocolo de intercambio de llave con autentificación basado en Diffie-Hellman.
NIST: National Institute of Standards and Technology – Instituto Nacional de Normas y Tecnología (Estados Unidos).
OAEP: Optimal Asymmetric Encryption Padding – Relleno asimétrico de cifrado óptimo.
OPENSSL: Proyecto de software libre basado en SSLeay, desarrollado por Eric Young y Tim Hudson.
PFS: Perfect Forward Secrecy – traducido generalmente como secreto-perfecto-hacia-adelante. Es la propiedad de los sistemas criptográficos que garantiza que el

descubrimiento de las claves utilizadas actualmente no compromete la seguridad de las claves usadas con anterioridad.
PRGA: Pseudo-Random Generation Algorithm – Algoritmo de generación pseudoaleatorio.
RC4: Ron's Code 4 – 4º código de Ron, siendo Ron el creador del algoritmo.
SAEP: Simplified Asymetric Encryption Padding – Relleno asimétrico de cifrado simplificado.
SD: Secure Digital. Es el nombre propio de un tipo de tarjeta de memoria.
SoC: System on Chip – Sistema sobre chip.
SSH: Secure Shell – Intérprete de órdenes seguro. Es un protocolo de acceso remoto seguro.
SSL: Secure Sockets Layer – Capa de conexión segura.
TLS: Transport Layer Security – Seguridad de la capa de transporte.
UART: Universal Asynchronous Receiver-Transmitter – Transmisor-Receptor asíncrono universal.
UNIX: Sistema operativo portable, multitarea y multiusuario.
USB: Universal Serial Bus – Bus de serie universal.
UTP: Unshielded Twisted Pair – Par trenzado no blindado.
VFPLite: Virtual Floating Point Lite – Coprocesador de punto flotante incorporado.
WEP: Wired Equivalent Privacy – Privacidad equivalente al cableado.
WPA: Wi-Fi Protected Access – Wi-Fi de acceso protegido.
YAK: Nombre propio – Protocolo de intercambio de llave con autentificación de llave pública.

Bibliografía

1. Ahmadian, Z., Somayeh, S., Salahi, A.: Security enhancements against UMTS–GSM interworking attacks. Computer Networks: The International Journal of Computer and Telecommunications Networking, Teherán, Irán (2010).

2. Federal Communications Commission: Tariff FCC No. 260., Federal Communications Commission (1985).

3. Crypto AG: PSTN Encrytion HC-2203. Disponible en: http://www.crypto.ch/en/products-and-services/products/pstn-encryption-hc-2203

4. ITU: M.1020 Carácteristicas de los circuitos internacionales arrendados de calidad especial con acondicionamiento especial en la anchura de banda. Recomendación (1988)

5. SpeechCodecs: Understanding various Speech Codecs. Disponible en: https://speechcodecs.wordpress.com

6. ITU: Software Tool Library 2009 User's Manual. (2009).

7. José Pastor Franco, M.: Criptografía digital: fundamentos y aplicaciones. (1998).

8. Corrales Sánchez, H.: Criptografía y Métodos de Cifrado, Universidad de Acála, Madrid

9. Pornin, T.: Key-agreement protocol. De: StackExchange. Disponible en: http://security.stackexchange.com/questions/35471/is-there-any-particular-reason-to-use-diffie-hellman-over-rsa-for-key-exchange

10. Real Academia Española: RAE. Disponible en: http://lema.rae.es/drae/?val=Criptograf%C3%ADa

11. Sabater, A.: Criptografía de Clave secreta: Cifrado en flujo., Centro Criptológico Nacional. España (2009).

12. Galvane, Q., Uzel, B.: Cryptography - RC4 Algorithm. (2012).

13. Rescorla, E.: RFC 2631 – Diffie-Hellman Key Agreement Method. (1999) Disponible en: http://tools.ietf.org/html/rfc2631

14. Adafruit: Embemed Linux Comparision. Disponible en: https://learn.adafruit.com/downloads/pdf/embedded-linux-board-comparison.pdf

15. PortAudio: PortAudio. Disponible en: http://portaudio.com/

16. Universidad Nacional de Córdoba: Apunte de Cátedra de Procesamiento Digital de Señales.

17. SearchSecurity De: Checksum Definition. Disponible en: http://searchsecurity.techtarget.com/definition/checksum

18. Turner, S.: Entender, analizar y reducir la latencia. Disponible en: http://blog.iweb.com/es/2014/02/entender-analizar-reducir-latencia/2463.html

19. Wolfram MathWorld: Discrete Logarithm. Disponible en: http://mathworld.wolfram.com/DiscreteLogarithm.html

20. Wolfram MathWorld: Primitive Root. Disponible en: http://mathworld.wolfram.com/PrimitiveRoot.html

21. Stallings, W.: Comunicaciones y redes de computadores 7th edn. Pearson Educación (2004).