UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Sistema seguro para compartir información entre smartwatches y smartphones

Máster Universitario en Ingeniería Informática

Autor: Lucía Rodríguez González

Tutor: Javier Gómez Escribano
Ponente: Germán Montoro Manrique

Departamento de Ingeniería Informática

junio 2019

Lucía Rodríguez González
Sistema seguro para compartir información entre smartwatches y smartphones

Lucía Rodríguez González
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Cada vez más gente usa Internet para navegar y compartir datos, pero por otro lado, cada vez hay más casos de robo de información a usuarios y grandes compañías. Es por esto que debemos preocuparnos de cómo funcionan los sistemas de almacenamiento de los que hacemos uso y hacerlos lo más seguros posible. Para ello, el uso de los modelos de seguridad y privacidad y los algoritmos de encriptación pueden ser la herramienta a utilizar en el camino hacia esta seguridad tan perseguida.

Este proyecto se ha centrado en el traspaso de datos sensibles entre dispositivos y usuarios, concretamente entre relojes inteligentes (smartwatches) y teléfonos inteligentes (smartphones), que utilizan una base de datos o servidor intermediario para guardar los datos. Con el planteamiento propuesto en este documento, si en algún momento este almacén de datos viera comprometida su seguridad, los archivos seguirían manteniendo su confidencialidad gracias al proceso de cifrado al que fueron sometidos previamente.

Se han realizado diversas pruebas y analizado distintas opciones de cifrado para dar con la mejor combinación de seguridad y eficacia y se ha implementado el sistema descrito sobre una aplicación real: la aplicación Taimun-Watch [1] del laboratorio Ambient Intelligence Laboratory (AmILab) .

Palabras clave

Modelos de seguridad y privacidad, datos sensibles, smartphone, smartwatch, Taimun-Watch, AmILab

Abstract

There is a growing number of people that use the Internet to browse and share data, but on the other hand, there is a growing number of cases of information theft to users and large companies. This is why we must worry about how the storage systems we use work and make them as safe as possible. For this, the use of the use of security and privacy models and encryption algorithms can be the tool to be used on the road to this sought-after security.

This project has focused on the transfer of sensitive data between devices and users, specifically between smartwatches and smartphones, which use a database or intermediary server to store the data. With the approach proposed in this document, if at any time the security of this data warehouse is compromised, the files would continue to maintain their confidentiality thanks to the encryption process to which they were previously submitted.

Various tests have been made and different encryption options have been analyzed to find the best combination of safety and efficacy and the system described has been implemented on a real application: the Taimun-Watch [1] application of the Ambient Intelligence Laboratory (AmILab).

Keywords

Security and privacy models, sensitive data, smartphone, smartwatch, Taimun-Watch, AmILab

Índice

Listas

Lista de códigos

Lista de figuras

Lista de tablas

1

Introducción

1.1. Motivación

En la actualidad, el uso de Internet está fuertemente extendido en todo el mundo, registrándose en febrero de 2018 un número de usuarios de más de la mitad de la población mundial [2], con un aumento del número de usuarios de dispositivos móviles y un descenso del de usuarios de ordenadores (tanto sobremesa como portátiles), donde muchos de estos usuarios utilizan más de un dispositivo con conexión a Internet diariamente. En este escenario, surge la necesidad de poder compartir datos e información entre usuarios y dispositivos, pero desafortunadamente, de la misma forma aparecen los robos de información en la red, tanto de forma aislada como masiva.

Antes de poseer la tecnología de comunicaciones que conocemos hoy, la forma más segura de hacer llegar un mensaje teniendo la certeza de que no había sido interceptado por individuos no deseados, era entregándoselo en mano al destinatario. En la actualidad, se puede tener esa misma certeza haciendo uso de los métodos adecuados. Estos métodos parten de la base de la criptografía antigua, desarrollada desde hace siglos en mensajes secretos, que ha ido evolucionando hasta convertirse en la criptografía moderna que se emplea actualmente en los algoritmos de cifrado que permiten garantizar la confidencialidad, integridad y disponibilidad de la información (modelo conocido como CIA triad, por sus siglas en inglés).

Por tanto, para preservar estos valores en la información que transmitimos digitalmente, debemos buscar la forma más segura y eficiente de compartir datos sensibles o personales entre distintos dispositivos, de manera que sepamos que ni han sido ni pueden ser comprometidos.

En este punto es donde se encuentra el desarrollo de la aplicación móvil Taimun-Watch [1], un sistema para ayudar en la autorregulación de personas con TEA mediante el uso combinado de smartwatches (relojes inteligentes) para la detección y ayuda en momentos de estrés; y smartphones (teléfonos inteligentes) para la gestión de contenidos y supervisión. A través de esta aplicación, se quieren compartir los datos obtenidos por el smartwatch, tanto con el smartphone con el que está sincronizado como con una base de datos a la que se puedan conectar otros usuarios que, en caso de estar autorizados, puedan obtener y visualizar dichos datos.

El uso del smartwatch ofrece la ventaja de poder ser utilizado a lo largo del día para monitorizar a los usuarios y recoger datos de su actividad de forma constante. Pero por otro lado, posee una serie de problemas de privacidad [3] y vulnerabilidades frente a ataques que traten de extraer los datos [4] mayor que en otros dispositivos, debido a su relativa inmadurez, que deben tenerse en cuenta a la hora de manejar los datos sensibles con los que trata la aplicación.

1.2. Objetivos del proyecto

El objetivo de este Trabajo de Fin de Máster es el planteamiento, desarrollo e implementación de un sistema capaz de compartir los datos sensibles de la aplicación Taimun-Watch, obtenidos por un smartwatch, con una serie de dispositivos y/o usuarios concretos, de una forma totalmente segura que proteja la confidencialidad de dichos datos.

Para lograrlo, se llevará a cabo un estudio de los sistemas actuales de compartición de datos de este tipo, destacando los puntos fuertes y las debilidades de cada uno e intentando aprovechar las características que puedan ser más útiles para lograr el objetivo del proyecto. En concreto, se pretende alcanzar el equilibrio entre tiempos de encriptación y envío y seguridad de los datos, analizando si se debe sacrificar parte de uno por el otro.

Una vez analizada la situación y las tendencias actuales en este tipo de aplicaciones y las tecnologías que se pueden aprovechar, se establecerá el plan de diseño de la aplicación a desarrollar y las características observadas con las que contará.

Para lograr un desarrollo más sencillo, primero se programará en un entorno aislado donde poder hacer pruebas con los procesos que debe seguir el sistema y una vez completada esta fase se integrará con la aplicación actual Taimun-Watch y se comprobará la correcta integración y compatibilidad con el sistema actual, guardando la relación con el diseño y aspecto visual ya existente, manteniendo el estilo de la aplicación y el nivel de usabilidad.

Por último, se llevarán a cabo una serie de pruebas con las que comparar distintos métodos y modos de encriptación y comprobar si los resultados finales obtenidos son satisfactorios.

2

Trabajo relacionado

Actualmente, el mercado de los relojes inteligentes se divide en dos grandes grupos: relojes con sistema operativo *watchOS* [5] de la marca Apple, conocidos como *AppleWatch*; y relojes que usan el sistema operativo desarrollado por Google, *Wear OS* [6], inicialmente conocido como *Android Wear* (hasta principios de 2018), que está basado en *Android* (la primera versión lanzada estaba basada en Android 4.4.2 "KitKat"). El desarrollo de este proyecto se realizará en Android, ya que es el sistema operativo más usado en todo el mundo (según la web de análisis de tráfico web StatCounter [7], en junio de 2018, el 80.08 % de los smartphones que se usaban en España eran Android. El 19,32 %, iOS) y por lo tanto es más fácil llegar a más usuarios, además de que hay una amplia variedad en gamas y marcas de dispositivos que lo utilizan, lo que se traduce en la posibilidad de optar por productos más básicos pero también más asequibles, o más caros pero con más prestaciones, según las necesidades de cada usuario.

En las primeras versiones de Android Wear, era totalmente necesario contar con un smartphone para realizar la mayoría de las tareas del reloj o poder usar las aplicaciones, con lo que el smartwatch quedaba relegado a ser una pantalla en la que proyectar información que también se podía consultar desde el smartphone emparejado. Para realizar el traspaso de la información entre ambos dispositivos así como para conseguir conexión a Internet en el reloj inteligente, se recurría a la tecnología bluetooth, la cual tiene algunas desventajas como baja velocidad de transmisión y poca seguridad [8].

Más adelante, a partir de mayo de 2016 (con Android Wear 2.0, basado en Android 5.5.1), empezó a ser posible la conexión directa a Internet desde el propio smartwatch a través de la tecnología wifi, más rápida y segura que el bluetooth, por lo que muchas aplicaciones comenzaron a aprovechar esta ventaja para usar servidores como receptores intermedios de la información compartida entre dispositivos. De esta forma, el smartwatch o el smartphone cargan los datos que quieren compartir en el servidor y el otro dispositivo los descarga cuando los necesita.

En la Figura 2.1 se pueden observar los dos mecanismos de comunicación posibles entre dispositivos "wearables" (dispositivos portátiles, llevables o vestibles, como el smartwath) y teléfonos inteligentes, que son: comunicación directa mediante bluetooth, o comunicación a través de una conexión wifi a la red pasando por los servidores de Google.

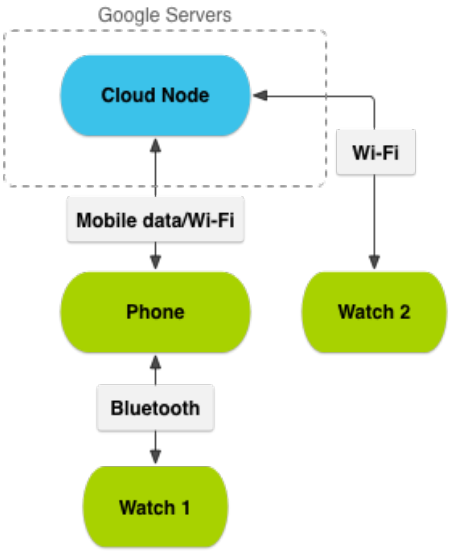

Figura 2.1: Ejemplo de red de comunicación de nodos con dispositivos wearables. `https://developer.android.com/training/wearables/data-layer`

El estudio desarrollado se centrará en la comunicación vía wifi por su mayor capacidad de transferencia de datos y posibilidades de seguridad que ofrece en comparación con el uso de bluetooth. Además, se usará un servidor intermediario para mantener la persistencia de los datos y realizar la comunicación entre los dispositivos. En [9] se mencionan diferentes formas de comunicación entre smartphone y smartwatch y sus características, y las vulnerabilidades de los smartwatches por las cuales se debe proceder con cuidado al manejar los datos y al tenerlos almacenados en el dispositivo, que se han tenido en cuenta para el desarrollo del proyecto.

2.1. Aplicaciones comerciales relacionadas

Al realizar un estudio de la situación actual del mercado de aplicaciones móviles para teléfonos y relojes inteligentes se ha podido observar, como ya se sospechaba, que la información detallada del funcionamiento de estas aplicaciones no es algo que se publique abiertamente, bien por recelo hacia la competencia o bien porque son detalles que consideran carentes de importancia para el usuario promedio y no llegan a difundir. Así pues, comprobar cuáles son las características exactas de seguridad de una aplicación o los métodos de los que se vale para compartir elementos entre dispositivos, es una tarea que se ha realizado de forma individual con cada aplicación que pudiese parecer interesante, ejecutándola y deduciendo mediante ingeniería inversa cuáles son los métodos que usa. A continuación se describen las características y funcionamiento de una serie de aplicaciones que han resultado de interés por su forma de actuar, con funcionalidades aparentemente similares a las que se buscan en el proyecto.

Sleep as Android [10]. Se trata de una aplicación de monitorización de sueño con recopilación de datos a través de los sensores del reloj que pueden ser enviados al teléfono para visualizarse en forma de gráfica, generar estadísticas, ser compartidos, etc. Cuenta con una Application Programming Interface (API) pública [11] que puede utilizar cualquier usuario que quiera añadir funcionalidad o soporte de la aplicación para, por ejemplo, dispositivos wearables que no lo tengan todavía y puedan ser integrados en ella. También ofrecen unas funciones que posibilitan la conexión de dispositivos a su "SleepCloud Storage", donde se almacenan los datos de los usuarios. Para poder acceder a este almacenamiento y los datos, se utilizan permisos de usuario adquiridos mediante el protocolo de autorización *OAuth 2.0*, identificándose contra los servidores de Google, con lo cual dejan que sea Google el que se encargue de la gestión de usuarios y contraseñas. Lo que no podemos saber es en qué forma se encuentran los datos almacenados de la aplicación: si usan algún sistema de encriptación, si tienen sistemas de seguridad internos, etc. Por otro lado, sí se puede comprobar que la transmisión de datos entre el teléfono y el reloj se realiza mediante Internet, por wifi.

Photo Gallery for Wear OS (Android Wear) [12]. Esta aplicación permite el traspaso de imágenes de un teléfono a un reloj que esté sincronizado. Lo característico de esta aplicación es que es posible pasar las fotografías o por medio del bluetooth o por medio del wifi, dependiendo de la configuración de la aplicación, así como tenerla instalada sólo en el reloj o en ambos dispositivos. Si se opta por la primera opción, la aplicación recurre a Google, como en el caso de la aplicación anterior, concretamente a Google Photos. De esta forma, simplemente realiza el inicio de sesión del usuario en su cuenta de Google Photos y comienza a descargar las imágenes en el reloj, con lo que delegan la seguridad a Google. Si queremos que la comunicación entre dispositivos sea vía bluetooth, es necesario tener instalada la aplicación en el teléfono. De esta segunda forma, podemos comprobar que las imágenes son cargadas mucho más despacio que usando wifi.

Google Fit [13]. Traída de la mano de Google, esta app lee y registra los datos de los sensores del reloj inteligente con el objetivo de ayudar al usuario a tener una vida más activa. Para usarla y acceder a los resultados no es necesario, aunque sí recomendable, contar con la aplicación en el teléfono, ya que se pueden consultar en el mismo reloj o en la aplicación web, siempre que se posea una cuenta de Google en la cual se haya iniciado sesión. En cuanto a la seguridad de los datos, no se ha podido averiguar con certeza cómo se almacenan o dónde (más allá de saber que están en algún servidor de Google), pero sí que se ha comprobado que la forma de transportar los datos es a través de Internet usando el wifi del reloj. En el caso de querer compartir los datos al teléfono, se cargarían en el servidor y una vez allí se descargarían en el móvil (Figura 2.2), no se pasan directamente, como sucede cuando usamos bluetooth.

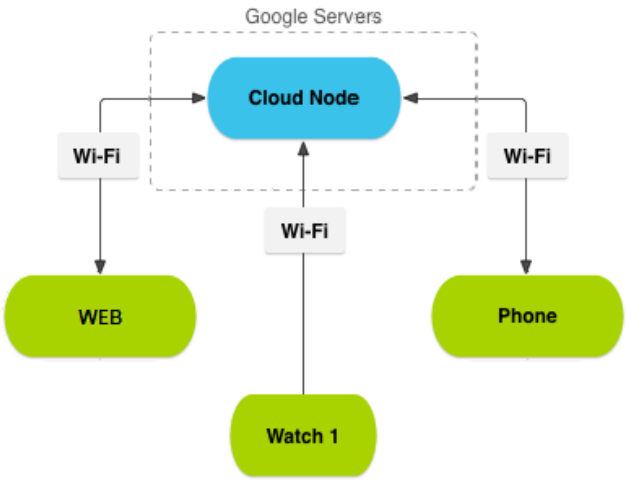

Figura 2.2: Modelo de conexión de la aplicación Google Fit

2.2. Seguridad en Smartwatches

En esta sección se expondrá el estudio llevado a cabo de los diferentes mecanismos que ofrece Android para otorgar seguridad a las aplicaciones para smartphones y smartwatches, lo cual es muy importante ya que, como se ha mencionado anteriormente y como se detalla en el trabajo de Wan Ching y Mahinderjit Singh [3], la seguridad de los smartwatches no es infalible.

"Android es una pila de software de código abierto basado en Linux" [1] que en el nivel más alto utiliza el framework de la Java API, por eso las aplicaciones Android se programan en una versión de Java para Android y también por eso se tiene acceso a muchas de las librerías de Java con las mismas funciones o parecidas, como la librería *javax/crypto*, de la cual a continuación se explicará el funcionamiento de los algoritmos de cifrado más comúnmente usados que ofrece y que son interesantes para el proyecto, distinguiendo entre algoritmos asimétricos y algoritmos simétricos.

2.2.1. Algoritmos de cifrado

El algoritmo de cifrado asimétrico más utilizado actualmente para propósito general y que tenemos disponible en Android es el Rivest-Shamir-Adleman (RSA) [14]. Se basa en la utilización de números primos y potenciación modular. Los usuarios que van a compartir información cifrada por este método poseen una clave privada, conocida solo por el propio usuario, y otra pública, que puede obtener cualquier otro usuario que quiera enviar un mensaje que solo pueda leer el dueño de la misma. Con esta clave pública e, normalmente de longitud n entre 1024 y 2048 bits, el remitente cifra el mensaje M que quiere enviar, por bloques, previa conversión numérica mediante la operación $M^e \mod n$. Para descifrarlo, el destinatario tan solo tendrá que aplicar la operación inversa $C^d \mod n$, siendo *C* el

[1] Android developers, Arquitectura de la plataforma: `https://developer.android.com/guide/platform?hl=es`

mensaje recibido cifrado y *d* la clave privada que solo conoce él y que por tanto nadie más puede usar para descifrar el mensaje.

Este sistema funciona gracias a la generación de las claves pública y privada a partir de dos números primos distintos p y q cuya multiplicación da como resultado n, con el que se puede usar la función de Euler(φ) de la forma $\varphi(n) = (p-1) \cdot (q-1)$ para aprovechar sus propiedades y obtener un e coprimo con $\varphi(n)$ y menor que $\varphi(n)$ que será la clave pública a partir de la cual se puede aplicar el algoritmo de Euclides extendido $e^{-1} \cdot \mod \varphi(n)$ y obtener la clave privada d.

Por otro lado, tenemos los algoritmos de cifrado simétrico, entre los que se encuentran Data Encryption Standard (DES) y algunas variantes de Advanced Encryption Standard (AES) .

El algoritmo de cifrado DES [15] encripta en grupos de 64 bits, o lo que es lo mismo, en grupos de 16 números hexadecimales, utilizando una clave de 56 bits que se usará después para decodificar el mensaje de nuevo (esta clave se guarda como una clave de longitud 64 bits, pero cada octavo bit de la misma es descartado). Si el tamaño del mensaje no es múltiplo de 64 bits, se añadirá un *padding* al final para completar. En primer lugar, se crearán 16 subclaves de 48 bits a partir de la original de 64. Para crear estas subclaves se utilizan unas tablas de permutación que alteran el orden inicial de la clave y descartan 8 bits, por lo que obtenemos una clave de 56 bits, que es dividida en dos mitades de 28 bits cada una para realizar 16 iteraciones en las que se desplazan los bits de cada mitad hacia la izquierda, una o dos posiciones, según indique el algoritmo en cada iteración, de manera que cada iteración depende del resultado de la anterior. A continuación, se unen las dos mitades en una sola clave y se aplica una segunda tabla de permutación de 48 bits sobre cada una de las 16 obtenidas (una por iteración), dando como resultado 16 claves de 48 bits (deshecha los otros 8 bits). Por otro lado, se utiliza una tabla de permutación de 64 bits con cada bloque de este tamaño del mensaje, se divide cada bloque en dos mitades de 32 bits y se realizan 16 iteraciones que hacen uso de la función *Feistel* para realizar *XORs* con la clave previamente obtenida y que dan como resultado de la última iteración el bloque final cifrado. En la Figura 2.3 se representa el funcionamiento descrito del algoritmo. Para descifrar un bloque de texto habría que seguir los mismos pasos pero invirtiendo el orden en el que se usan las subclaves [16].

El algoritmo DES se considera inseguro puesto que sus claves son relativamente fáciles de "romper" (el ataque efectivo conocido que se llevó a cabo para probar su vulnerabilidad tardó menos de 24 horas en descifrar la clave), por lo que normalmente se utiliza su variante *Triple Data Encryption Algorithm (TDEA)* (también conocida como Triple DES, 3DES o Triple DEA), donde se utiliza el algoritmo DES tres veces por separado, encriptando primero con la primera subclave, desencriptando con la segunda y encriptando de nuevo con la tercera subclave:

$$texto\ cifrado = EK3(DK2(EK1(texto\ en\ claro)))$$

Para descifrar el texto cifrado se haría lo inverso:

$$texto\ en\ claro = DK1(EK2(DK3(texto\ cifrado)))$$

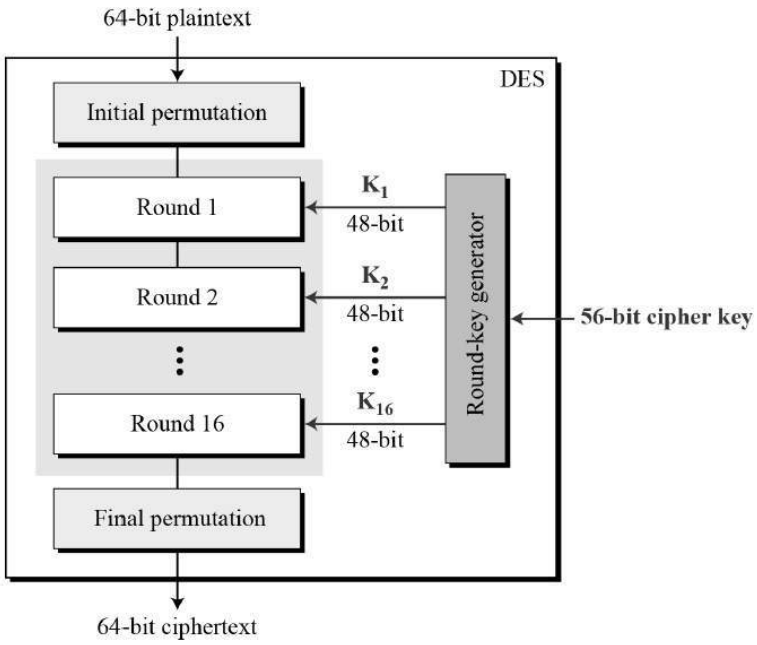

Figura 2.3: Esquema de funcionamiento del algoritmo DES. Extraído de [17]

Otro algoritmo de cifrado simétrico de uso común es AES [18]. Se trata del estándar de cifrado que se utiliza actualmente en casi todos los sistemas seguros y hasta ahora solo se han planteado ataques teóricos que puedan "romperlo", pero ninguno que se pueda llevar a cabo en los sistemas actuales. Este algoritmo existe en sus variantes de 128, 192 y 256 bits de tamaño de clave, todas ellas disponibles para Android.

El funcionamiento de AES se basa en rondas de actuación en las que se realizan sustituciones y permutaciones sobre cada bloque de bits, obtenidos al dividir el texto que se va a cifrar y que tienen un tamaño fijo de 128 bits, independientemente de la longitud de la clave. En primer lugar, se realiza una XOR de cada byte del bloque de texto con un bloque de la clave. Este paso es conocido como *AddRoundKey*. A continuación, se realizan 9, 11 o 13 rondas, dependiendo de si la clave es de 128, 192 o 256 bits respectivamente, en las que se realizan las siguientes transformaciones:

- *SubBytes*: se reemplaza cada byte con otro según una tabla de búsqueda preestablecida por el algoritmo.
- *ShiftRows*: los bytes son rotados cíclicamente un número de veces dependiente de la fila del bloque.
- *MixColumns*: cada columna es multiplicada por un valor fijo (polinomio de Galois), mezclando los bytes.
- *AddRoundKey*: se suma la subclave con la función XOR como al inicio.

En la última ronda realiza una vez más los pasos *SubBytes*, *ShiftRows* y *AddRoundKey*. En el descifrado se realiza el proceso inverso con las funciones inversas.

En la Figura 2.4 se puede observar un esquema explicativo del funcionamiento del algoritmo.

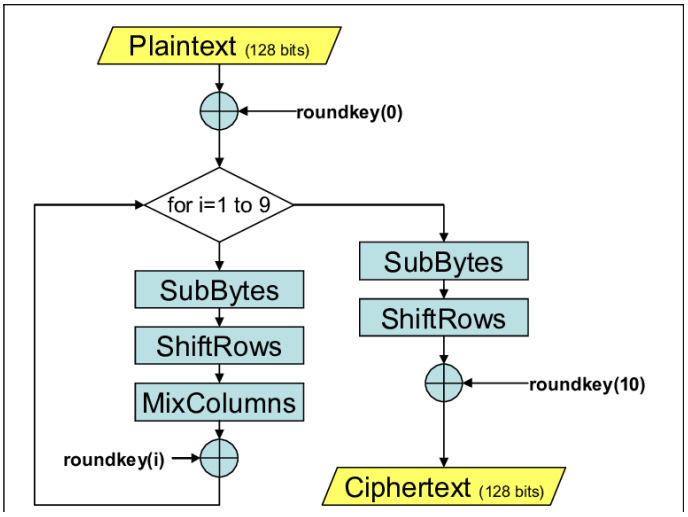

Figura 2.4: Esquema de funcionamiento del algoritmo AES para una clave de 128 bits. Extraído de [19].

2.2.2. Modos de encriptación

Cada algoritmo de encriptación de la librería *javax/crypto* cuenta con una serie de modos de cifrado y de *padding* (relleno) a disposición del usuario. Los modos más comúnmente usados, algunos de ellos explicados en [20], son los siguientes:

- *Electronic Codebook (ECB)* , el más sencillo. Los mensajes se dividen en bloques y cada uno de ellos es cifrado usando la misma clave. No es seguro porque los bloques idénticos tienen una encriptación idéntica, por lo que se puede averiguar por fuerza bruta de forma relativamente sencilla.
- *Cipher-Block Chaining (CBC)* . En este caso, los bloques del mensaje antes de ser cifrados son transformados mediante una XOR con el bloque anterior cifrado, con lo que tienen dependencia uno de otro bloque en texto plano. Para su uso, se necesita un vector de inicialización aleatorio. Este modo puede ser vulnerable a los llamados "padding oracle attacks" [2], que explotan la adición de paddings (rellenos) al final del cifrado.
- *Cipher Feedback (CFB)* y *Output Feedback (OFB)* . Ambos funcionan con flujos de cifrado que realizan una operación XOR entre los bloques de texto para cifrarlos. Al igual que CBC , necesita un vector de inicialización que, en ambos casos, al cambiar este usando la misma entrada de texto, cambia la salida, lo cual puede ser muy útil para algunas aplicaciones.
- *Counter (CTR)* . Funciona también con un vector de inicialización único que va incremen-

tando para obtener un flujo pseudoaleatorio, usando siempre números únicos (nonces) para prevenir duplicaciones. Es parecido a OFB pero cuenta con la ventaja de que la encriptación y desencriptación pueden paralelizarse para ser más rápido, y además la transmisión de errores afecta solo a los bits erróneos y no se propagan.

- *Ciphertext Stealing (CTS)* . Añade algo de complejidad para solucionar la problemática que se da cuando los bloques de cifrado no se pueden dividir todos de la misma forma. Actúa igual que CBC para todos los bloques salvo para los dos últimos, donde utiliza parte del contenido del penúltimo bloque cifrado como padding del último, consiguiendo así que el tamaño del texto plano sea el mismo que el del texto cifrado.
- *Galois Counter Mode (GCM)* Parecido a CBC y CTR , utiliza también bloques de cifrado y funciones XOR, pero no cifra cada bloque con el anterior como CBC , por lo que puede ejecutarse en paralelo. Como su nombre indica, al igual que CTR utiliza un contador para crear un número pseudoaleatorio con el que encriptar y se basa en los campos de Galois y la función hash universal (universal hashing) de Carter y Wegman [21]. Tiene la ventaja de que asegura no solo la confidencialidad sino también la autenticidad mediante códigos de autenticación de los bloques cifrados.

2.2.3. Padding

En cuanto al padding o relleno, se usa para completar los bloques de cifrado en el caso de que un bloque no llegue a ocupar el tamaño requerido. Por ejemplo, si estamos usando AES de 128 bits (16 bytes) y tenemos que cifrar una palabra de 5 bytes, habrá que rellenar con 11 bytes extras el mensaje. También se recurre al uso del padding en algoritmos que pueden dar lugar a encriptaciones resultantes iguales al cifrar el mismo texto, por lo que al añadir el relleno se otorga algo más de aleatoriedad que dificulte el desciframiento por fuerza bruta de los textos. Los paddings disponibles en los algorimos simétricos de *javax/crypto* son *ISO10126Padding*, *NoPadding* o *PKCS5Padding* (el más común, que concretamente funciona añadiendo todos los bytes extras con el mismo valor cada uno: el número de bytes necesarios para completar el bloque). En algunos casos el uso del relleno puede ser útil para descubrir errores producidos por el *efecto avalancha* [3].

La desventaja del uso de relleno es que puede hacer vulnerable el texto a ataques de tipo “padding oracle attacks”. En los casos de los modos con contador como CTR , OFB o CFB , no se aplica padding, ya que el texto cifrado siempre tiene la misma longitud que el texto en plano, por lo que en nuestro caso, especificaremos en la función *Cipher* el tipo de relleno NoPadding.

[2] Ataques que se aprovechan de los sistemas de validación del relleno, de los que obtienen si el mensaje ha sido cifrado correctamente o no, para descifrar por fuerza bruta el mensaje, como se explica en [22].

[3] Efecto que se produce cuando al realizar un cambio pequeño en la entrada (por ejemplo, cambiar un bit) este cambio crece hasta modificar, al menos, un 50 % de la salida

2.2.4. Funciones hash

Como se ha explicado previamente, para cifrar es necesario aportar una clave de una determinada longitud que se convierte en la "llave" de todo lo que se cifre, pues es necesario conocerla para descifrar. La mejor manera de obtener esta clave es haciendo uso de las funciones hash [23], que son aquellas que cumplen como mínimo las dos propiedades siguientes:

1.– *Compresión*: la entrada puede tener un tamaño variable que siempre producirá una salida de tamaño fijo.

2.– *Eficiencia*: la función es relativamente sencilla de calcular sobre cualquier entrada dada.

En adición a estas propiedades básicas, se necesitan otras tres propiedades potenciales para tener una buena función criptográfica hash, que son:

1.– *Resistencia a pre-imagen*: para cualquier valor de hash obtenido h, es computacionalmente inviable encontrar una preimagen y tal que $H(y) = h$

2.– *Resistencia a segunda pre-imagen*: es conputacionalmente inviable encontrar una segunda entrada que tenga la misma salida que otra entrada especifica, es decir, dado x, encontrar una segunda pre-imagen $y \neq x$ tal que $H(x) = H(y)$

3.– *Resistencia a colisiones*: es conputacionalmente inviable encontrar dos entradas distintas x, y que den como resultado del hash el mismo resultado, de la forma $H(x) = H(y)$

Estas son las propiedades necesarias para tener una buena función hash, pero aún así, actualmente existen funciones hash para las que se han encontrado fallos en una o varias de estas propiedades, como colisiones intencionadas, que se siguen usando aunque su uso esté desaconsejado. Las colisiones suceden debido a la propia naturaleza del hash, que tiene un número limitado de bits de salida pero cuya entrada es ilimitada, por lo que es posible que exista un mismo hash para más de un objeto diferente.

Las funciones hash más conocidas que además existen en la librería *javax/crypto* de Android y que se podrían utilizar en el proyecto son las siguientes:

- *Message-Digest Algorithm 5 (MD5)* , desarrollado como sucesor del MD4 por Ronald Rivest en 1992, proporciona una salida de 128 bits a partir de una entrada, sobre la que se realizan 4 pasadas aplicando distintas operaciones lógicas bit a bit (suma, rotación, AND, OR, XOR). Al poco tiempo de su creación se descubrieron posibles debilidades y actualmente se ha demostrado que existen métodos relativamente sencillos para hallar colisiones de hash, por lo que su uso en seguridad está desaconsejado.
- *Secure Hash Algorithm 1 (SHA-1)* , fue creado en 1995 por la National Security Agency (NSA) como mejora de SHA-0 y realiza las mismas operaciones lógicas que MD5 en 5

pasadas (80 rondas) sobre el mensaje, produciendo una salida de tamaño fijo de 160 bits. En 2005 se demostró que solo eran necesarias 2^{69} operaciones en vez de 2^{80} y en 2017 Google anunció públicamente que habían conseguido producir una colisión hash. Actualmente su uso está desaconsejado e incluso algunos navegadores web bloquean las páginas que usan este cifrado para sus certificados.

- *Secure Hash Algorithm 2 (SHA-2)* , es una familia de funciones hash desarrolladas por la NSA y publicada en 2001 con tres tamaños de salida originalmente, SHA-256, SHA-384 y SHA-512, a la que luego se añadiría SHA-224 y versiones de SHA-512 (SHA-512/224 y SHA-512/256, que no están disponibles para Android). Tanto SHA-224 como SHA-256 tienen un tamaño de bloque de 512 bits mientras que el de las demás es de 1024 bits. Aunque comparte la misma estructura y operaciones matemáticas de SHA-1 , aún no se han encontrado colisiones.

3

Diseño

En este apartado se explicarán las decisiones que se han llevado a cabo a la hora de integrar el proyecto que se describe en este documento con la aplicación Taimun-Watch [1]. Esta aplicación ha sido desarrollada por el equipo del laboratorio AmILab [24] de la Escuela Politécnica Superior (EPS) de la Universidad Autónoma de Madrid (UAM) , en colaboración con el colegio de educación especial Alenta [25]. Se trata de una aplicación para smartphone y smartwatch orientada a entender mejor el comportamiento de personas con Trastornos del Espectro Autista (TEA) , que en algunas ocasiones pueden tener dificultades tanto para expresar cómo se sienten como para pedir ayuda. Para lograr este objetivo, la aplicación monitoriza las constantes del usuario (con TEA) mediante el smartwatch que, en base a los datos recogidos, es capaz de detectar situaciones de estrés e intervenir para calmar al usuario, mostrando una serie de estrategias de regulación que previamente ha programado una persona de apoyo, que puede ser un padre, un tutor o un profesor cualificado para ayudar a las personas con TEA .

Como se muestra en la figura 3.1, todos los datos que recoge el reloj inteligente actualmente son transferidos al teléfono mediante el uso de bluetooth, donde se pueden representar y analizar en la interfaz a la que la persona de apoyo del usuario del reloj tiene acceso. Esta interfaz sigue un modelo sencillo, orientado al usuario y con un diseño plenamente definido y totalmente funcional, al cual se debe adaptar la integración del proyecto que se quiere llevar a cabo, respetando en la medida de lo posible la estructura actual.

Se explicarán, así mismo, los razonamientos llevados a cabo para obtener la solución con mejor relación entre velocidad de ejecución y eficiencia para lograr el objetivo de convertir la insegura transmisión de los datos entre reloj y teléfono por bluetooth, en una transmisión de mayor velocidad y seguridad haciendo uso de la conexión wifi y la encriptación de ficheros, con un diseño como el que se puede ver en la figura 3.2.

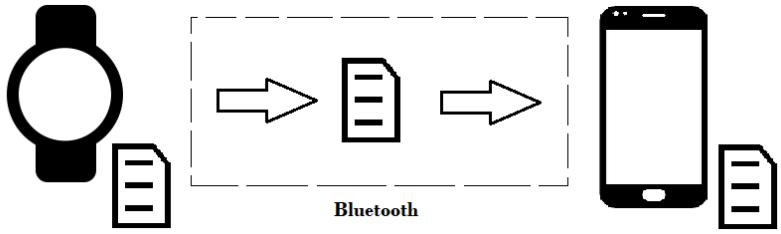

Figura 3.1: Estado actual de la aplicación: Transferencia de ficheros del smartwatch al smartphone mediante bluetooth.

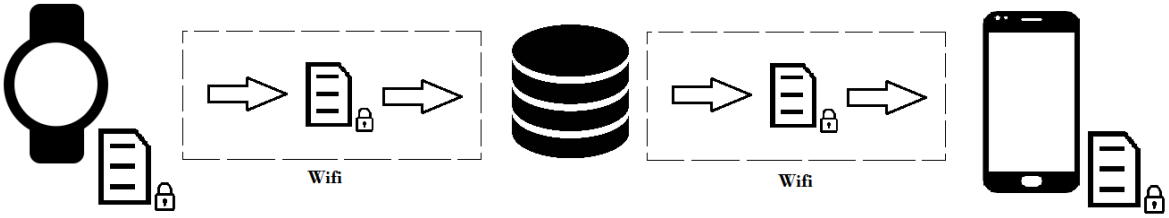

Figura 3.2: Propuesta: Transferencia de ficheros encriptados del smartwatch al smartphone mediante wifi, usando una base de datos intermediaria.

3.1. Explicación del diseño

A continuación se van a desarrollar e investigar las opciones que existen actualmente para cifrar los archivos que serán compartidos, haciendo uso de la información presentada en el capítulo 2.

Primera iteración: clave asimétrica

En primer lugar, es necesaria una clave simétrica con la que cifrar los archivos de datos (Figura 3.3). Esta clave se obtendrá a partir de un código PIN que introduce el usuario. La clave tendrá que subirse al repositorio junto con los archivos ya cifrados por la misma, y para mantener la privacidad de esta, se cifra con la clave pública del usuario antes de enviarla (Figura 3.4). Ahora, para poder leer el fichero en otro dispositivo se realiza el proceso inverso: se debe poseer la clave simétrica, obtenida desencriptando con la clave privada del usuario, y el fichero descifrado, que se obtiene con la clave simétrica que se acaba de descifrar.

Básicamente, de esta forma cada usuario puede subir sus datos usando su propia clave pública y obtenerlos con su clave privada.

El principal problema de este método surge cuando se quiere que más de un usuario tenga acceso a los archivos del servidor, que al estar encriptados con la clave pública, necesitan la clave privada correspondiente del usuario original. Una posible solución a este problema sería cifrar cada una de las claves simétricas con cada una de las claves públicas de todos los usuarios con acceso a estos ficheros, pero esto haría que hubiese tantos ficheros con la clave simétrica usada para la encriptación

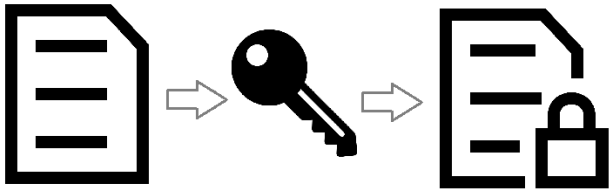

Figura 3.3: Con la clave simétrica se cifra el fichero.

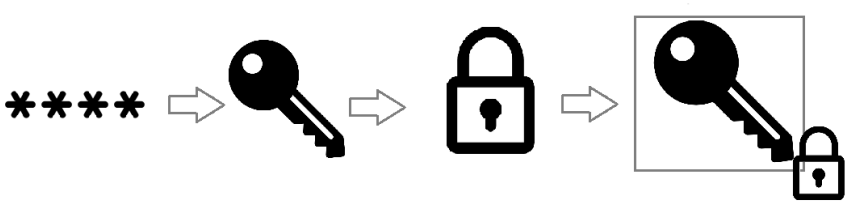

Figura 3.4: A partir del pin se obtiene la clave simétrica que luego se cifra con la clave pública.

y desencriptación del fichero como usuarios con acceso (n usuarios = 1 fichero de datos y n claves cifradas), y esto a su vez por cada archivo si usamos claves simétricas distintas para cada uno. De esta forma, en el momento en el que se tuviera un número grande de usuarios con acceso a los mismos archivos, el sistema sería insostenible.

Por otro lado, cualquier usuario que se *una* a un grupo que haya sido creado previamente y que ya cuente con archivos compartidos, no tendrá acceso a estos ficheros a menos que en el mismo momento de la unión se añada uno a uno el fichero con las claves simétricas cifradas con la clave pública de este nuevo miembro por cada fichero de datos.

Segunda iteración: clave simétrica

Descartado el uso de la criptografía asimétrica, se analiza el posible uso de claves simétricas. Con este tipo de claves, todos los usuarios encriptan y desencriptan usando una misma clave, acabando con el problema de tener un archivo de clave por cada usuario, y esta clave a su vez se comparte cifrada con un PIN que es necesario que el usuario conozca para obtener el texto en claro de los archivos.

La idea, desarrollada un poco más, consiste en que a partir de una clave alfanumérica generada se utiliza una función hash con la que obtenemos una clave simétrica, con la cual cifraremos y descifraremos los archivos de datos que se van a compartir. Esta clave a su vez está protegida por un PIN que solo conoce el grupo de usuarios).

De esta forma, para que un usuario tenga acceso a los archivos, necesita conocer el nombre del

grupo de archivos y el PIN para obtener la clave simétrica necesaria para la desencriptación (Figura 3.5).

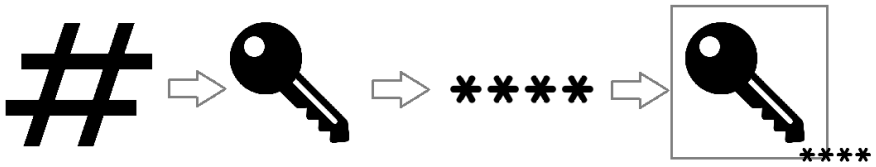

Figura 3.5: Se aplica la función hash con la cadena alfanumérica para obtener la clave simétrica, que es protegida con el PIN.

Esta segunda idea es la que resulta más adecuada y aconsejable para el diseño de la aplicación y para cubrir las necesidades del proyecto.

Proceso de cifrado final

Los datos del usuario se recogen gracias al smartwatch sincronizado con el smartphone, donde ambos dispositivos cuentan con la aplicación Taimun-Watch instalada. Para realizar la compartición de los archivos con el método explicado antes, es necesario que el usuario haya introducido previamente el nombre del grupo de archivos al que quiere añadir los ficheros, que es una clave alfanumérica autogenerada (para que sea única) la primera vez que un usuario quiere crear un grupo de archivos y que luego se comparte entre los usuarios a los que se quiera dar acceso al grupo, así como también es necesaria la introducción del PIN que comparten los archivos de ese grupo y con el que se descifra la clave simétrica. Los siguientes pasos para compartir los datos recogidos por el reloj se enumeran a continuación:

1.– La compartición de los ficheros se realiza de forma asíncrona cuando el usuario activa esta opción mediante un botón en la aplicación móvil.

2.– En este momento, el reloj recibe una señal y comienza el proceso de agrupación de las mediciones obtenidas desde la última sincronización, para obtener un fichero por día con datos registrados.

3.– Una vez los ficheros están preparados comienza su encriptación, para lo cual es necesario:

   3.1.– Obtener el fichero con la clave cifrada del servidor.

   3.2.– Desencriptarla mediante el PIN que el usuario introduce.

4.– Ya cifrados, se envían los ficheros de datos al servidor, donde los demás usuarios ya tienen acceso a ellos, y se eliminan del reloj.

5.– El teléfono descarga y almacena los ficheros encriptados sincronizados.

Cuando se añade un fichero con un nombre ya existente en la base de datos, normalmente es sobreescrito por el nuevo, por lo que los archivos que ya tengan datos al realizar la actualización deberán ser descargados, desencriptados, extendidos con los nuevos datos y encriptados de nuevo antes de cargar el archivo definitivo de vuelta en el servidor.

En el caso de que un nuevo usuario obtenga el acceso a los ficheros del grupo, al sincronizar por primera vez obtendría todos los ficheros almacenados. Como se ha mencionado, los ficheros obtenidos por el smarphone permanecerán encriptados en todo momento para aumentar la seguridad del sistema, y solo en el momento en el que sea necesario proceder a su lectura al acceder a la sección de estadísticas de los datos en la aplicación, se llevará a cabo la desencriptación de los ficheros necesarios, se leerán, se representarán y se volverán a encriptar.

En el diseño planteado de esta aplicación, el PIN que comparten los usuarios podría ser modificado en cualquier momento por un usuario autorizado. Con esto se busca, por un lado, tener la posibilidad de dejar fuera del acceso a los ficheros a los usuarios a los que no se les comunique el cambio de PIN, y por otro lado, poder cambiar el PIN en caso de olvido. El cambio se produciría en un smartphone (de un usuario con permisos para ello), que usaría el nuevo código para encriptar el fichero con la clave cifrada del servidor y actualizar el que hubiera previamente con el PIN anterior, así como el que tiene su smartwatch. Cuando los usuarios quisieran descargar nuevos ficheros, esto les actualizaría también el fichero con la clave y solo podrían leerlos utilizando el nuevo PIN.

4

Desarrollo

Como ya se ha explicado en capítulos anteriores, Taimun-Watch es una aplicación funcional, que se encuentra en uso en varios pilotos en diferentes centros de atención a personas con diversidad funcional intelectual, con un estilo ya definido que no debe ser modificado sustancialmente, por lo que para añadir toda la funcionalidad que se va a desarrollar (menús en la aplicación, ajustes adicionales, etc), se debe integrar siguiendo el diseño actual y haciendo uso en la medida de lo posible de funciones ya existentes. En este caso, será necesario modificar tanto código de la aplicación para móvil como de la aplicación para reloj.

En el capítulo 2 se habló de los algoritmos presentes en la librería *javax/crypto*, de interés para el proyecto, los cuales se utilizan concretamente dentro de Android, en la función *Cipher* (*javax.crypto.Cipher*), que es "el núcleo del framework *Java Cryptographic Extension* (JCE)" [26].

Una vez explicados los algoritmos y modos que se pueden utilizar, y ya decidido que se usará encriptación simétrica (desarrollado en el capítulo 3), queda decidir la especificación para la clase Cipher.

Entre los modelos de algoritmo simétrico de encriptación que ofrece, se utilizará el algoritmo AES, que es el más utilizado de este tipo ya que cuenta con una gran robustez sin ser descifrable por fuerza bruta a día de hoy; con una clave de 256 bits. El tamaño de la clave indica, como se explicó en el capítulo 2 en la sección 2.2.1, el número de iteraciones (rondas) de transformaciones que realiza el algoritmo AES, siempre usando bloques de 128 bits. En este caso, la clave que se utilizará para cifrar se generará mediante la función hash SHA-256 (de la que se habló en 2.2.4), perteneciente a la familia de algoritmos hash SHA-2 aún considerados seguros, que devuelve una salida de 256 bits y cuya entrada será una cadena aleatoria de 8 caracteres formada por números y letras.

Se podría plantear utilizar una clave de 128 bits, ya que actualmente este tamaño es suficiente para mantener la seguridad, pues aún no se ha conseguido sacar por fuerza bruta ninguna clave de AES 128, salvo en casos teóricos que comienzan suponiendo que se posee un ordenador cuántico, pero sin embargo se puede apreciar una migración generalizada en los sistemas de seguridad de claves de 128 a 256 bits, ya que es muy probable que en algún momento deje de ser seguro el uso de claves de 128 bits. De hecho, la propia NSA determinó que todos los archivos "Alto Secreto" debían ser

encriptados usando AES-256 en lugar de AES-128, tal y como señala Jeffrey Goldberg, uno de los principales expertos en seguridad de la aplicación de gestión de contraseñas *1Password* [27], en un artículo de 2013 [28] en el que ya se hablaba sobre esta migración de claves de algunas aplicaciones a 256 bits y donde detalla los motivos que les llevaron a ellos mismos (a *1Password*) a realizar el cambio de longitudes de claves. En cuanto a coste computacional, hace unos años, el uso de uno u otro tamaño de clave podía suponer bastante diferencia, pero con los sistemas actuales, incluyendo los smartwatches, esto ha dejado de ser una preocupación. En el capítulo 5 se realizarán una serie de pruebas para comprobar hasta qué punto puede afectar la longitud de la clave en la aplicación móvil de este proyecto.

En conclusión, se fijará el tamaño de la clave en 256 bits dado el nivel de seguridad que ofrece dicha longitud a día de hoy y en previsión de los futuros avances tecnológicos, y en consecuencia se usará el algoritmo hash SHA-256 para formar dicha clave.

En cuanto al modo de operación del algoritmo, de los que nos ofrece *Cipher* (explicados en los modos de encriptación de la librería *javax/crypto* de la sección 2.2), se opta finalmente por el modo *GCM* frente a los otros por varios motivos. En primer lugar, se descarta ECB, cuyo uso está desaconsejado más allá de fines educativos por su inseguridad; se descarta también el modo CBC, por su vulnerabilidad frente a ataques de padding, y de los restantes se descartan los modos de flujo de cifrado (CFB y OFB) y CTS, que no son tan veloces como sí pueden serlo CTR o GCM. En estos dos últimos modos, tenemos la ventaja de que cifrado y descifrado pueden ser paralelizables y por tanto necesitar menos tiempo de ejecución, y además evitan los problemas de padding (tanto ataques como complicación del código). GCM está basado en CTR, y por tanto ambos comparten la misma gran debilidad (al igual que otros modos que usan vectores de inicialización): no se puede repetir el mismo vector de inicialización (IV o nonce) en distintos bloques o su seguridad se verá comprometida, como se explica en [29], por ello está especificado en GCM que nunca será repetido un mismo IV. Por otro lado, es posible que si se intercepta el resultado de un cifrado (*ciphertext*), que va a ser la entrada del siguiente bloque de cifrado, y es alterado, el resultado final del cifrado y posterior descifrado no será el original. Esto se conoce como “ataques de bitflip”. Como defensa ante este ataque, es posible ampliar el algoritmo de CTR añadiendo un sistema que genera un código de autenticación de cada mensaje (Message Authentication Code (MAC) o tag) que verifica con los datos si siguen siendo los originales o han cambiado en algún momento de forma inesperada, caso en el que se lanzaría un aviso para tomar las medidas oportunas (reenvío del bloque, detención de la ejecución...). Sin embargo, el modo GCM ya realiza esta comprobación por sí mismo, por lo que tener que adaptar por ejemplo CTR para que cuente con este sistema sería un trabajo extra que GCM evita.

Por otro lado, es cierto que GCM (como cualquier otro modo de cifrado) tiene vulnerabilidades conocidas, al menos dos que explotan las características de su diseño, las cuales están descritas en el paper de Ferguson [30]. La primera es que es posible que si el tamaño de los “tags” de autenticación no es muy grande, se produzcan colisiones a propósito para violar las pruebas de autenticación. En

relación con esto, es posible obtener información sobre el hash usado en el cifrado a partir de las colisiones, con las consecuencias de seguridad que conocer dicha información supone, expuesto en [29]. Al saber acerca de estas debilidades, se pueden tomar medidas para contrarrestarlas, como asegurar un tamaño de “tags” adecuado. Por ello y por el resto de características ya mencionadas, se considera que este es el modo de cifrado con más ventajas y beneficios para ser aplicado en el caso descrito en este proyecto.

En conclusión, el modo que se utilizará con el algoritmo de encriptación AES será GCM debido a su mayor velocidad de cifrado y descifrado, su alta seguridad y su garantía de la autenticidad de los datos.

Además de la encriptación de los ficheros, hay otros aspectos de seguridad que deben ser definidos para el desarrollo del código, como la gestión de las claves en los dispositivos. Para ello, se utilizará el *KeyStore* de Android o *Android Keystore*. Se trata del sistema seguro de guardado de claves de Java en su adaptación para Android: un contenedor en el que se almacenan las claves criptográficas que dificulta su extracción del dispositivo para usos no autorizados. Las claves que se usarán en la aplicación serán gestionadas por este almacén y para obtenerlas será necesario conocer el “alias” y el PIN con el que se guardaron.

Finalmente, se debe decidir cómo y dónde se guardarán todos los archivos de los usuarios. Una opción podría ser almacenarlos en un servidor local en la UAM , pero ello conllevaría mantener a al menos una persona encargada de realizar comprobaciones, análisis y mantenimiento periódicamente para garantizar el correcto funcionamiento del mismo, disponible para realizar reparaciones en caso de fallo con la mayor brevedad posible para minimizar el tiempo de interrupción del servicio, etc. Por otro lado, debido a las leyes de protección de datos y al origen sensible y personal de los datos que se almacenarán, podrían surgir problemas legales si se descuidase la seguridad de este servidor propio, ya que toda la responsabilidad recaería sobre los propietarios del mismo.

La solución a estos y otros problemas reside en recurrir a un servidor externo fiable, como el servidor de Google, *Firebase* [31], que se describe a sí mismo como “Una plataforma integral de desarrollo para dispositivos móviles”. Entre sus muchas funciones, *Firebase* cuenta con el servicio *Cloud Storage*, que permite cargar y descargar archivos en una base de datos asignada al proyecto de la aplicación con una integración muy sencilla y de manera transparente al usuario. Además de esto, hasta cierto tamaño de tráfico de datos (aplicaciones con muchos usuarios), su uso es gratuito, por lo que para desarrollar la aplicación del proyecto no habrá que preocuparse por obtener ningún tipo de licencia. Otro motivo por el que escoger este servicio frente a otros, es que las nociones básicas de su uso y los primeros pasos para incluirla en una aplicación móvil se enseñan en la asignatura optativa de cuarto curso *Desarrollo de aplicaciones para dispositivos móviles* de la EPS , lo que aporta interés personal en cuanto a aprender a manejarla y aprovechar para poner en práctica lo visto en el aula.

4.1. Código

En este apartado se mostrarán secciones del código de la aplicación en los que se puede observar la forma final de uso de la función *Cipher*, con las características anteriormente comentadas, junto con otras funciones y métodos importantes en el proyecto que ya se han ido mencionando.

Los fragmentos de códigos mostrados a continuación han sido simplificados para resaltar la parte de la función que estamos tratando, por lo que se han omitido controles de excepciones, definiciones de clases y otros.

4.1.1. Creación de grupos

Cuando se crea un grupo nuevo, se genera su nombre de forma aleatoria con la función que se puede ver en el Código 4.1, que mezcla letras mayúsculas (de la *A* a la *Z*, 26 letras distintas) y minúsculas (26 posibilidades de la *a* a la *z*) con números del 0 al 9 (10 números diferentes) en una cadena de 6 caracteres. De esta forma se tienen 62 elementos ($26 + 26 + 10$) con repetición en grupos de 6 cifras, o $62^6 = 56\,800\,235\,584$ combinaciones posibles H, donde en un primer momento se podría pensar que la probabilidad de que se repita la misma cadena de nombre del grupo, utilizando la probabilidad de intersección de sucesos independientes, es de aproximadamente $\frac{1}{H} \cdot \frac{1}{H} = 3{,}1 \cdot 10^{-22}$. Sin embargo, debido al conocido como "paradoja del cumpleaños" o "problema del cumpleaños", (explicado y desarrollado en [32]) la probabilidad de que haya una colisión de nombres es menor, concretamente, si se considera que todas las combinaciones son igualmente probables y se aplica la siguiente fórmula obtenida a partir del desarrollo en series de Taylor desde la fórmula obtenida en el problema del cumpleaños, con una probabilidad p del 0.5 de que haya una coincidencia

$$n(p) \approx \sqrt{2 \cdot H \cdot \ln(\tfrac{1}{1-p})} = 280\,609{,}77$$

se calcula que quedan aproximadamente 280.609 nombres de grupos con una probabilidad de menos del 50 % de repetirse, por lo que se puede concluir que usando este método con las características descritas para nombrar los grupos hace poco probable que se produzca una colisión o que se lleguen a agotar todas las combinaciones únicas en el entorno del proyecto descrito.

El objeto de generar el nombre de esta forma es, por un lado, evitar la repetición de nombres de grupos, ya que deben ser únicos en el servidor, y por otro lado, añadir un elemento más de privacidad al sistema, ya que si en algún momento un intruso obtuviese acceso a la base de datos, no le resultaría fácil averiguar por dicho nombre a quién pertenecen los datos almacenados.

Este nombre generado será el que se asocie a la clave simétrica AES, obtenida utilizando la función hash *SHA-256* con otra secuencia generada de forma aleatoria de 8 caracteres como entrada, que será transformada en un hash único con un tamaño fijo de 256 bits para almacenarlo en el KeyStore como un objeto de tipo *SecretKeySpec*, usando el nombre del grupo como alias y el PIN del usuario como

clave de acceso a dicho *SecretKeySpec*. Esta clave es la que se debe usar para cifrar y descifrar los archivos.

Código 4.1: En este fragmento de código se genera una cadena aleatoria con una longitud de 6 caracteres a partir de letras mayúsculas y minúsculas y números.

```
1  String generatedString = GenerateRandomString.randomString(6);
2
3  public static class GenerateRandomString {
4      static final String upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
5      static final String lower = upper.toLowerCase(Locale.ROOT);
6      static final String digits = "0123456789";
7      static final String DATA = upper + lower + digits;
8      static Random RANDOM = new Random();
9
10     static String randomString(int len) {
11         StringBuilder sb = new StringBuilder(len);
12
13         for (int i = 0; i < len; i++) {
14             sb.append(DATA.charAt(RANDOM.nextInt(DATA.length())));
15         }
16         return sb.toString();
17     }
18 }
```

4.1.2. Encriptación

En el Código 4.2 se realiza la encriptación de un fichero del grupo, haciendo uso de la función local *getFromKeyStore* (explicada más adelante en el Código 4.5), donde usando el PIN compartido por los usuarios del grupo, se obtiene la clave AES que se estableció para el grupo originalmente, con la cual se inicializa el objeto *Cipher*, creado en modo encriptación indicando el algoritmo de cifrado y su modo; en este caso, AES en modo GCM. Para la inicialización, es necesario también aportar un vector de inicialización, o IV. Este vector es un bloque de bytes generado aleatoriamente que necesita el modo GCM (entre otros) para obtener aleatoriedad y producir siempre diferentes resultados de cifrado incluso si el texto introducido es el mismo, sin tener que usar distintas claves. Como ya se mencionó anteriormente, no se debe utilizar el mismo IV con la misma clave más de una vez. Este vector de inicialización será utilizado en la encriptación y posteriormente en la desencriptación, y para cada fichero se usará un IV distinto que se mandará con el texto cifrado.

Una vez inicializado el cifrador, se procede a leer por partes el fichero a encriptar, empleando un buffer de 4096 bytes (tamaño múltiplo de 256, que es el tamaño de la clave AES) que permitirá que incluso si el fichero a cifrar es muy grande no se produzca un déficit de memoria, y a cada bloque se le aplica el método *update* de *Cipher*, que es el que realiza la encriptación, a la vez que se va escribiendo el resultado de la misma en el fichero de salida. Por último, se llama al método *final* de *Cipher* que

termina de cifrar y ajusta los últimos bytes y se escribe este último bloque en el fichero de salida.

Código 4.2: En esta figura se presenta el código de encriptación con AES de un fichero “plainFile” cuyo resultado se guardará en “encryptedFile”. Se utiliza el PIN del usuario y el nombre del grupo obtenidos previamente para acceder a la clave almacenada en el KeyStore.

```
1  public File encryptAES(Context ctx, File plainFile, File encryptedFile) {
2      FileOutputStream fos = new FileOutputStream(encryptedFile);
3      FileInputStream fis = new FileInputStream(plainFile);
4
5      // A partir de las preferencias se obtienen el grupo (alias clave KS) y el pin para abrir el KS
6      String pin = Preferences.getPinCode(ctx);
7      SecretKeySpec sks = getFromKeyStore(Preferences.getGroupCode(ctx), pin.toCharArray());
8
9      // Generar aleatorio para el Vector de Inicialización (IV)
10     SecureRandom rng = new SecureRandom();
11     int readLen;
12
13     byte[] ivData = new byte[c.getBlockSize()];
14     rng.nextBytes(ivData); // IV aleatorio
15
16     Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
17     c.init(Cipher.ENCRYPT_MODE, sks, new IvParameterSpec(ivData));
18
19     // El IV se escribe al principio del fichero.
20     fos.write(ivData);
21
22     // Leer por bloques de 4096 bytes
23     byte[] inputBuffer = new byte[32 *128];
24     while ((readLen = fis.read(inputBuffer)) != -1){
25         byte[] enc = c.update(inputBuffer, 0, readLen);
26         fos.write(enc);
27     }
28
29     // Parte final
30     byte[] enc = c.doFinal();
31     fos.write(enc);
32
33     fos.close();
34     fis.close();
35     plainFile.delete(); // Ya no necesitamos el fichero en texto plano
36     return encryptedFile;
37 }
```

4.1.3. Desencriptación

Para descifrar el fichero es necesario tener en el KeyStore la clave AES que se usó para cifrarlo, a la que se accede con el PIN correcto compartido entre los usuarios. Por otro lado, es necesario conocer el vector de inicialización (IV), que se puede obtener del comienzo del fichero (los primeros 16 bytes). Con el *SecretKeySpec* y el IV, se inicializa el objeto *Cipher* en modo descifrador, indicando también el algoritmo de desencriptación y su modo. Seguidamente, se elige un tamaño de buffer de desencriptación de 4096 bytes al igual que en la encriptación, usando el método *update* de *Cipher* para ir desencriptando por bloques a la vez que se va escribiendo el resultado en el fichero de salida. Por último, se llama al método *final* de *Cipher* y se escribe el final del fichero (Código 4.3).

Código 4.3: En esta figura se presenta el código de desencriptación de un fichero cifrado “encFile” con AES que guarda en otro fichero “decryptedFile” el resultado en claro. Se utiliza el PIN del usuario y el nombre del grupo para acceder a la clave almacenada en el KeyStore.

```
 */
public int decryptAES(Context ctx, File encFile, File decryptedFile) {
    FileInputStream fis = new FileInputStream(encryptedFile);
    FileOutputStream fos = new FileOutputStream(decryptedFile);

    // Recuperar sks de KeyStore con las preferencias del PIN y el nombre del grupo
    String pin = Preferences.getPinCode(ctx);
    SecretKeySpec sks = getFromKeyStore(Preferences.getGroupCode(ctx), pin.toCharArray());

    int ptLength;
    byte[] iv = new byte[16];
    // Obtener IV del principio del fichero
    fis.read(iv);

    // Inicializar cipher con IV y usar la clave recuperada del KS
    Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
    c.init(Cipher.DECRYPT_MODE, sks, new IvParameterSpec(iv));

    // Leer por bloques de 4096 bytes
    byte[] inputBuffer = new byte[32 *128];
    while ((ptLength = fis.read(inputBuffer)) != -1){
        byte[] dec = c.update(inputBuffer, 0, ptLength);
        fos.write(dec);
    }

    // Parte final
    byte[] dec = c.doFinal();
    fos.write(dec);

    fos.close();
    fis.close();
}
```

4.1.4. Salvado de clave en el KeyStore

Esta función “saveToKeyStore” mostrada en el Código 4.4 recibe: un alias, o nombre para guardar la clave, que será el nombre del grupo; una contraseña, que en este caso es el PIN compartido entre los usuarios del grupo de ficheros y una clave de tipo *SecretKeySpec*, que será la que se quiere guardar en el KeyStore. A continuación, se inicializa el almacén de claves y se crea la nueva entrada con la clave a guardar. Para poder enviar la clave protegida al reloj y a los demás usuarios con acceso, se recurre al método *store*, que guarda el KeyStore creado en un fichero y protege la integridad del mismo con la contraseña aportada, haciendo posible su difusión de forma segura.

Código 4.4: En esta figura se presenta el código con el que se crea y almacena la clave de tipo SecretKeySpec en el almacén de claves KeyStore, asignando un alias y el PIN del usuario como contraseña para acceder a esta clave.

```
1  public void saveToKeyStore(String alias, char[] password, SecretKeySpec key, Context ctx) {
2      KeyStore ks = null;
3
4      // Inicializar KS vacío
5      ks = KeyStore.getInstance(KeyStore.getDefaultType());
6      ks.load(null);
7
8      // Crear entrada de la clave en el KS
9      KeyStore.ProtectionParameter protParam = new KeyStore.PasswordProtection(password);
10     ks.setEntry(alias, new KeyStore.SecretKeyEntry(key), protParam);
11
12     // Guardamos el fichero del ks en caché para guardar luego en KS
13     File file = new File(ctx.getCacheDir(), "keyStoreMobile");
14     FileOutputStream fos = new FileOutputStream(file);
15     ks.store(fos, password);
16 }
```

4.1.5. Obtención de la clave del KeyStore

A partir del alias de la clave guardada en el KeyStore y su contraseña, se puede acceder a la clave. Para ello, es necesario poseer previamente el fichero de KeyStore, que es cargado usando la contraseña correcta con el método *load*. Esta función, que se puede ver en el Código 4.5, termina con el retorno de la clave obtenida o “null” si algo no funcionó correctamente.

Código 4.5: En esta figura se presenta el código para la obtención de la clave del KeyStore una vez se ha descargado del servidor y se conoce el alias (en este caso el nombre del grupo) y el PIN.

```
1  public SecretKeySpec getFromKeyStore(String alias, char[] password) {
2      KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
3      File file = new File(getFilesDir(), "keyStore");
4      FileInputStream fis = new FileInputStream(file);
5      //Cargar keystore
6      ks.load(fis, password);
7
8      // Obtener clave
9      Key key = ks.getKey(alias, password);
10     if (key instanceof SecretKeySpec) {
11         return (SecretKeySpec) key;
12     }else{
13          return null;
14     }
15 }
```

4.1.6. Carga de ficheros a Firebase

La carga de ficheros se realiza en la parte del reloj cuando se actualizan los datos, y en el teléfono cuando se crea un nuevo grupo de usuarios. Si se están actualizando datos, se enviarán los ficheros locales al servidor y se borrarán del dispositivo una vez guardados. Si se trata de la formación de un grupo nuevo, se crea en la base de datos una carpeta con el nombre único de dicho grupo y se añade su clave encriptada correspondiente (el fichero del KeyStore) (Código 4.6).

4.1.7. Descarga de ficheros desde Firebase

La función “firebaseDownload” recibe la lista de ficheros que tiene que descargar, la ruta donde tiene que buscarlos y donde los guardará. Previamente se ha definido *storage* en la clase como:

```
private FirebaseStorage storage = FirebaseStorage.getInstance();
```

En la figura del Código 4.7 se puede ver el fragmento de código empleado en la descarga de ficheros en el reloj, que es muy parecido a la descarga en el teléfono, pero en este último caso se añade una comprobación para descargar solo los ficheros que no se posean ya.

Código 4.6: En esta figura se presenta el código que carga uno o varios ficheros en el servidor de Firebase y borra los archivos locales.

```
1  private void firebaseUpdate(ArrayList<File> files, String middlePath, Context ctx) {
2      for (File updFile : files) {
3          // Fichero que se va a enviar a Firebase. La ruta es del tipo "events/events-dd_mm_aaaa.txt"
4          File auxFile = new File(updFile.getParent(), updFile.getName());
5          // Se crea el URI del fichero.
6          Uri uriFile = Uri.fromFile(auxFile);
7          // Ruta en firebase
8          StorageReference storageReference = storage.getReference();
9          // El nombre de la carpeta en Firebase es el nombre del grupo
10         StorageReference fullStoRef = storageReference.child(Preferences.getGroupCode(ctx) + "/" +
               middlePath + "/" + uriFile.getLastPathSegment());
11
12         // iniciar subida
13         UploadTask uploadTask = fullStoRef.putFile(uriFile);
14         StorageTask status = uploadTask.addOnSuccessListener(new
               OnSuccessListener<UploadTask.TaskSnapshot>() {
15             @Override
16             public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
17                 //Obtener URL de descarga para mostrarla
18                 Uri url = taskSnapshot.getDownloadUrl();
19                 Log.i("WEAR", "URL fichero cargado: " + url);
20             }
21         });
22         //Una vez subidos los ficheros, se eliminan del dispositivo.
23         auxFile.delete();
24     }
25 }
```

Código 4.7: En esta figura se presenta el código que descarga uno o varios ficheros del servidor de Firebase y lo guarda localmente.

```
1  private Map<String, Boolean> firebaseDownload(ArrayList<File> files, String middlePath, File folder,
       Context ctx) {
2      Map<String, Boolean> resultadosDescargas = new HashMap<>();
3      for (File file : files) {
4          String grupo = Preferences.getGroupCode(ctx);
5          // Ruta al fichero en Firebase. Será del tipo "5g3Azy/events/events-dd_mm_aaaa.txt"
6          StorageReference folderRef = storage.getReference().child(grupo + "/" + middlePath + "/" +
               file.getName());
7
8          // Ruta del fichero descargado en el dispositivo.
9          final File tempName = new File(folder + "/" + "enc-" + file.getName());
10
11         // Guardar en carpeta sensor o en events. Se descargan en segundo plano.
12         StorageTask status = folderRef.getFile(tempName).addOnSuccessListener(new
               OnSuccessListener<FileDownloadTask.TaskSnapshot>() {
13             @Override
14             public void onSuccess(FileDownloadTask.TaskSnapshot taskSnapshot) {
15                 // Se crea el fichero temporal tempName
16                 tempName.setWritable(true);
17                 tempName.setReadable(true);
18             }
19         });
20         // Guardamos los resultados de la descarga para luego manejarlo.
21         resultadosDescargas.put("enc-" + file.getName(), status.isSuccessful());
22     }
23     return resultadosDescargas;
24 }
```

5

Pruebas

En este capítulo se describen las pruebas que se han realizado con el objetivo de conocer mejor el impacto que suponen los cambios realizados en la aplicación, comparando entre la versión actual que comparte los archivos vía bluetooth y la desarrollada en el proyecto, que encripta y comparte los archivos vía wifi. Por un lado se contrastarán los tamaños de ficheros cuando están en el reloj y cuando ya han sido compartidos y se encuentran en el teléfono. Por otro lado, se tomarán medidas de tiempo de paso de archivos con los dos métodos, donde se espera ver una gran mejora al utilizar el wifi como transporte de datos.

5.1. Recursos

Para realizar las pruebas, se ha usado el smartwatch LG Watch Urbane y el smartphone Xiaomi Redmi 4 Pro, cuyas características técnicas se pueden observar en la Tabla 5.1.

Características	Smartphone	Smartwatch
Modelo	Xiaomi Redmi 4 Pro	LG Watch Urbane
Almacenamiento	32GB	4GB
RAM	3GB	512MB
Procesador	Qualcomm Snapdragon 625 octa-core 2GHz	Qualcomm Snapdragon 400 quad-core 1.2GHz
Sistema Operativo	Android 6.0.1	Android 7.1.1
Bluetooth	4.1	4.0

Tabla 5.1: Especificaciones del hardware utilizado para la realización de las pruebas.

Además, se ha utilizado un grupo de ficheros de datos extraídos de experimentos reales anteriores, con una duración aproximada de 12 horas de información. Estos datos son recopilados por el reloj mientras el usuario lo lleva puesto y la aplicación se está ejecutando, e internamente son agrupados por día y tipo antes de ser enviados. Estos tipos se dividen en dos grupos: sensores y eventos, donde los sensores utilizados son: acelerómetro, detector de pasos, giroscopio y pulsómetro; y los eventos

son el registro de hitos claves de la aplicación en el reloj, como por ejemplo, el momento en el que se activa el proceso de una regulación. En la Tabla 5.2 se muestra el detalle de los ficheros de datos utilizados.

Tipo de fichero	Sensor	Nombre del fichero
Events	-	events-14_3_2018.txt
Sensors	Acelerómetro	acelerometro-14_3_2018.txt
	Detector de pasos	detectorPasos-14_3_2018.txt
	Giroscopio	giroscopio-14_3_2018.txt
	Pulsómetro	pulsometro-14_3_2018.txt

Tabla 5.2: Especificaciones de los ficheros utilizados para la realización de las pruebas.

5.2. Pruebas de tamaños

En la primera prueba, se ha querido comprobar si la encriptación de los ficheros conlleva un aumento de su tamaño respecto al que tenían en texto plano. Para ello, se ha pasado cada uno de los ficheros sin cifrar por la función de encriptación de la aplicación, obteniendo el archivo correspondiente cifrado con AES256 en modo GCM sin padding.

Como se aprecia en la Tabla 5.3, donde se puede ver el tamaño de cada fichero antes y después del proceso, los tamaños de los ficheros sin cifrar y cifrados solo varían en 16 bytes cada uno. Esto se debe a la adición al comienzo del archivo cifrado del valor del vector de inicialización (IV) que ocupa esos 16 bytes extras. Por tanto, encriptar todos los ficheros de datos solo requiere 80 bytes más que con la aplicación actual.

Fichero	Estado	Tamaño (bytes)
events	sin cifrar	63.228
	cifrado	63.244
acelerometro	sin cifrar	67.549.269
	cifrado	67.549.285
detectorPasos	sin cifrar	67.647
	cifrado	67.663
giroscopio	sin cifrar	11.258.534
	cifrado	11.258.550
pulsometro	sin cifrar	195.160
	cifrado	195.176

Tabla 5.3: Comparación de tamaños de ficheros originales y después de ser cifrados.

El tamaño total de todos los archivos cifrados es de 79.133.918 bytes.

5.3. Pruebas de tiempos

El objetivo de la siguiente prueba es la comparación de los tiempos de la aplicación con la versión actual y con la desarrollada. Para ello, se han tomado medidas de los tiempos que tarda la aplicación en realizar el traspaso de ficheros del reloj al teléfono con cada una de estas versiones, haciendo uso de la función *TimingLogger* de Android [33].

Con la aplicación actual, el grueso de trabajo lo realiza la transmisión de los ficheros por bluetooth, que va enviando por trozos los archivos que han generado los sensores y los eventos. Se han tomado medidas del tiempo que necesita el reloj para enviar todos los archivos mencionados anteriormente al móvil desde que este le envía la señal de que está listo para recibir hasta que el reloj indica que ha terminado, obteniendo un tiempo medio total de 3.669 segundos (aproximadamente una hora). En la Tabla 5.4 se puede ver el desglose por ficheros de los tiempos medios registrados.

Fichero	Tiempo
events	7 segundos
acelerometro	1.925 segundos
detectorPasos	768 segundos
giroscopio	668 segundos
pulsometro	311 segundos
Total	**3.669 segundos**

Tabla 5.4: Tiempo medio de intercambio de ficheros entre el reloj y el teléfono con la aplicación actual usando bluetooth.

En estos resultados, llama la atención que los ficheros de detector de pasos, que ocupan 67.647 bytes en conjunto, tarden más en enviarse que los del giroscopio, que suman 11.258.534 bytes. Al revisar el número de ficheros por sensor, se puede comprobar que, no obstante, el detector de pasos tiene más ficheros individuales que el giroscopio (Tabla 5.5).

Fichero	Número de ficheros
events	24
acelerometro	1.398
detectorPasos	1.398
giroscopio	1.280
pulsometro	1.041

Tabla 5.5: Número de ficheros fragmentados de cada tipo de sensor y eventos.

Sin embargo, al observarlos detenidamente, se comprueba que los ficheros del detector de pasos están, en su mayoría, vacíos (no se registraron pasos) pero aún así son enviados; mientras que los del acelerómetro contienen mucha información que incluso, en algunos casos, es necesario fragmentar para ser enviada en trozos más pequeños debido a las limitaciones de bluetooth, lo que hace que el envío tarde aún más.

Una vez analizada la situación actual, se procede a la obtención de los tiempos de ejecución de la nueva versión de la aplicación. Para medir estos tiempos, se han llevado a cabo dos tipos de pruebas: una utilizando una clave AES de 256 bits y otra con la clave de 128 bits. En ambas pruebas, la mayor parte del tiempo empleado en el paso de ficheros está en la encriptación previa al envío y en la subida de archivos a la plataforma de Firebase. A la hora de hacer estas pruebas, la velocidad de la red era de 100 Mb/s (por cable), llegando por wifi una media de 70Mb/s, variando entre 60 Mb/s de mínima y 86 Mb/s de máxima.

En la primera prueba con clave de 256 bits, el tiempo medio total de realización de la tarea por parte del reloj, desde que recibe la señal del teléfono hasta que le envía de vuelta la señal de finalización, es de 101,544 segundos. Esta tarea consiste en descargar ficheros del repositorio si los hubiera previamente (en el caso de la prueba no los hay) juntarlos con los archivos que tiene en ese momento el reloj, encriptarlo todo y subirlo a Firebase.

De ese tiempo total de la tarea, 61,04 segundos son de subida de ficheros, 14,709 segundos de encriptaciones y los 25,795 segundos restantes corresponden a otras operaciones secundarias, como el creado y borrado de ficheros. Este desglose de tiempos se puede ver en la gráfica de la Figura 5.1, donde se puede observar que donde más tiempo se consume es en el envío, que depende en gran parte de la red. El tiempo que se tarda en encriptar cada fichero individual haciendo uso de la clave de 256 bits se puede consultar en la gráfica de la Figura 5.2.

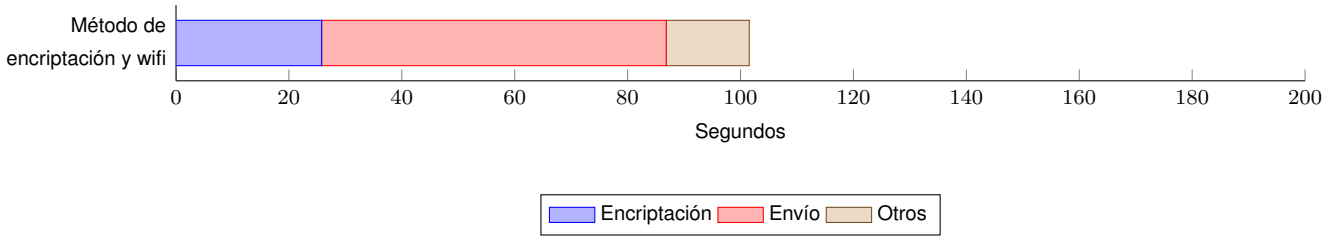

Figura 5.1: Desglose de tiempo de subida de ficheros encriptados por wifi.

En la segunda prueba se ha querido comprobar si el incremento de seguridad adquirido al aumentar el tamaño de la clave de 128 a 256 bits influye de forma sustancial en la duración de la fase de encriptación del traspaso de ficheros, y sopesar así el sacrificio realizado de tiempo frente a seguridad, en caso de que lo haya. En la gráfica 5.2 se pueden ver y comparar los tiempos medios de encriptación de los mismos ficheros que antes se encriptaban con la clave de 256 bits, ahora encriptados usando la clave de 128 bits, donde se aprecia una diferencia de 1,645 segundos más para el cifrado con la

clave de mayor tamaño. Como se puede observar en estos ficheros en concreto, excepto para los del detector de pasos, es mayor la duración del proceso de encriptación usando la clave de 256 bits frente a la de 128 bits.

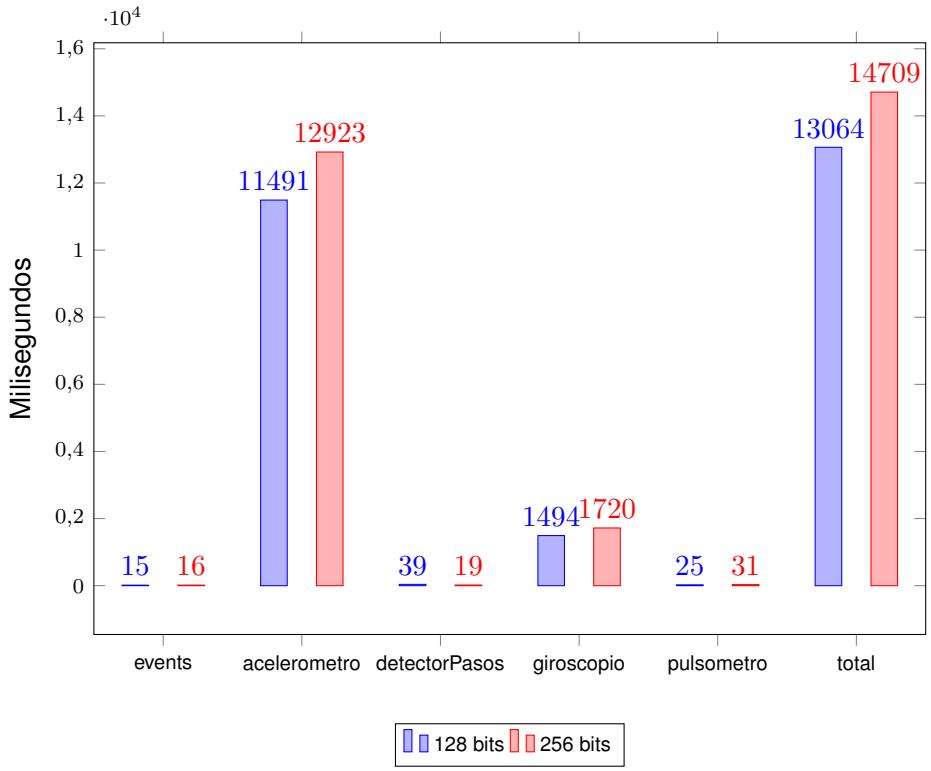

Figura 5.2: En esta figura se pueden ver los resultados de las mediciones de tiempos de encriptación de los diferentes ficheros usando una clave de 128 bits y otra de 256 bits.

En la aplicación, no siempre se cargarán archivos grandes, por lo que se ha realizado también el estudio de la diferencia de tiempos usando los dos tamaños de clave con ficheros más cortos, concretamente, de unas 2 horas de recopilación de datos, obteniendo los resultados representados en la gráfica de la Figura 5.3.

En este caso, la diferencia entre el uso de una u otra longitud de clave es de 0,576 segundos, siendo más rápida de nuevo la encriptación con clave de 128 bits.

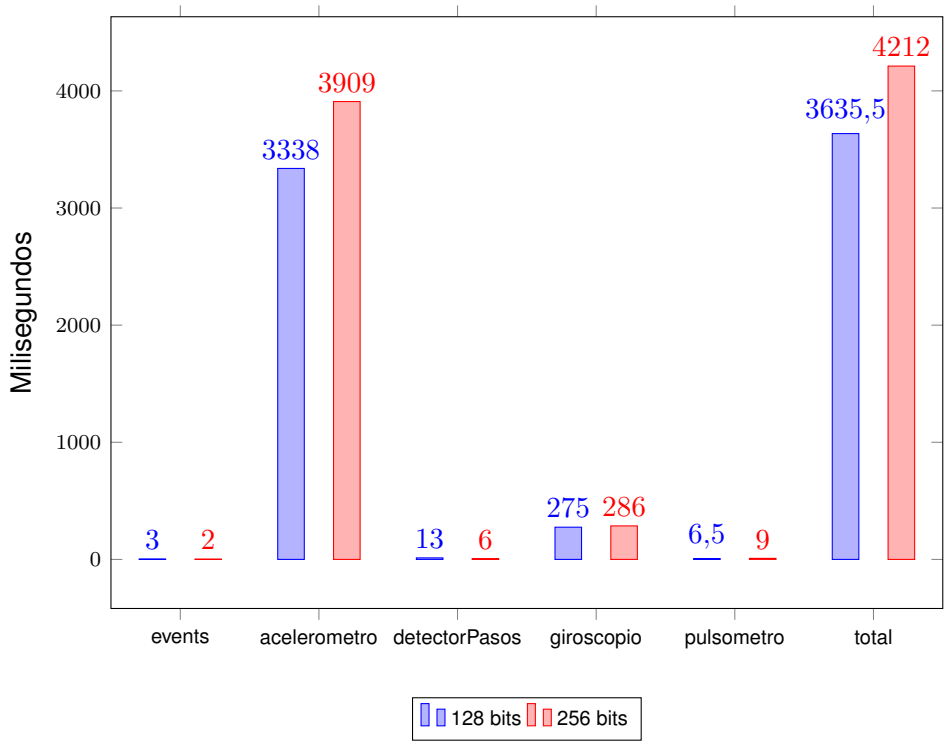

Figura 5.3: En esta figura se pueden ver los resultados de las mediciones de tiempos de encriptación con ficheros pequeños usando una clave de 128 bits y otra de 256 bits.

5.4. Conclusión de las pruebas

En conclusión, en las pruebas realizadas se ha comparado, por un lado la transferencia de archivos por bluetooth y por wifi a través del servidor de Firebase cifrando previamente, y por otro lado la diferencia de tiempos usando una clave de cifrado de AES de 128 bits o una de 256 bits.

En el primer caso, la diferencia de tiempos es de aproximadamente una hora con el diseño actual frente a 101,544 segundos obtenidos con las modificaciones del proyecto, es decir, que se consigue un ahorro muy significativo con el nuevo sistema de compartición de ficheros.

En cuanto a la longitud de la clave, se ha comprobado que efectivamente una clave más larga para el algoritmo de cifrado conlleva mayores tiempos de encriptación, que en el caso observado con ficheros de mayor tamaño supondría un 11,18 % más de tiempo, y en el caso de los ficheros más pequeños un 13,67 % más. Estos incrementos de tiempo, mínimos como se ha podido comprobar, apenas serán perceptibles para el usuario final y sin embargo obtendrá a cambio una mayor seguridad, por lo que se establece definitivamente el uso de la clave de 256 bits.

6

Conclusiones y trabajo futuro

6.1. Conclusiones

Tras realizar el análisis del estado actual de la aplicación Taimun-Watch, se hizo visible la falta de medidas de seguridad que garantizasen la confidencialidad de los datos sensibles que maneja, por lo que se llevó a cabo un estudio de las posibles soluciones, teniendo en cuenta las limitaciones de las librerías de seguridad de Android, así como la forma de actuar de otras aplicaciones ante el mismo problema. Como resultado, se aplicó un sistema de cifrado de claves simétricas que protege la confidencialidad de los ficheros de usuarios almacenados, concretamente, tras analizar todas las opciones disponibles, se decidió usar el algoritmo AES con clave de 256 bits y modo GCM .

Así mismo, en el análisis también se detectó el problema de la lentitud que sufre la aplicación en el envío de ficheros del smartwatch al smartphone, debido en gran parte a que hace uso de la tecnología bluetooth para ello, cuya velocidad de transmisión es muy reducida, además de ser muy insegura, como se concluyó en el estudio realizado. Por tanto, se propuso como solución realizar el envío de ficheros mediante wifi, haciendo uso de la plataforma Firebase de Google como servidor intermedio, donde se almacenan y descargan los datos cifrados.

El nuevo método de traspaso de ficheros descrito en el proyecto junto con los módulos necesarios para llevarlo a cabo, como la gestión de grupos de usuarios, se han implementado en la aplicación actual con una integración completa y de forma totalmente funcional y lista para su uso con usuarios.

Se ha comprobado que la reducción de tiempos frente al sistema anterior es un éxito, disminuyendo el tiempo de compartición de horas a minutos; así como el aumento de la seguridad que ofrece la encriptación frente al paso de archivos en texto plano mediante bluetooth.

Finalmente, este sistema podría ser fácilmente implementado en otras aplicaciones que tengan requisitos similares de traspaso de archivos de forma segura entre dispositivos.

6.2. Trabajo futuro

Aunque este trabajo supone un gran avance para la aplicación sobre la que se ha desarrollado y para la compartición de ficheros entre dispositivos móviles en general, se puede seguir mejorando y ampliando en muchos aspectos.

Uno de estos aspectos mejorables es la limitación que nos presenta el PIN que comparten los usuarios de un mismo grupo, ya que, aunque este PIN se puede modificar como se explicó en el capítulo 3, por ejemplo en caso de olvido de la clave, no funciona de forma infalible para casos en los que se quiera excluir del grupo a algún usuario. Al cambiar el PIN se actualizaría el fichero del KeyStore que se encuentra en el repositorio con la clave de cifrado de los archivos, para que solo se pudiera abrir con este nuevo PIN, pero la clave de cifrado sigue siendo la misma, por tanto, si un usuario guardase el fichero del KeyStore que posee (para el que sí conoce el PIN) antes del cambio, podría utilizarlo en lugar del nuevo fichero para sacar la clave de cifrado. Un usuario avanzado con conocimientos técnicos e información del funcionamiento interno de la aplicación podría realizar estos pasos, pero no es el perfil de usuario medio al que está dirigida la aplicación. Además, se considera de gran importancia tener una alternativa al posible olvido del PIN, ya que si esto sucede, se perderían todos los datos del grupo, ya sean de días, meses o incluso años de recopilación, que para este trabajo de investigación son de suma importancia. Por estos motivos el funcionamiento actual en este aspecto es el descrito, con la intención de mejorarlo en un futuro.

Otra de las tareas que quedan abiertas ante una posible mejora es el cambio de sistema de generación de nombres, diseñado en un primer momento para no ser demasiado largo y poderse compartir de forma sencilla, pero que podría ser, por ejemplo, un hash mucho más grande, para evitar problemas de colisiones, y ser compartido mediante códigos QR.

Por último, las pruebas han sido realizadas con unos ficheros de datos obtenidos en pruebas controladas, con poca actividad en determinados momentos de la recopilación, por lo que los resultados podrían variar si se obtuviesen de otros usuarios o en otro entorno de pruebas, que podría ser una prueba futura.

Bibliografía

[1] Juan C Torrado, Javier Gomez, and Germán Montoro. Emotional self-regulation of individuals with autism spectrum disorders: smartwatches for monitoring and interaction. *Sensors*, 17(6):1359, 2017.

[2] Informe de 2018 de Hootsuite y We Are Social. `https://www.slideshare.net/wearesocial/digital-in-2018-global-overview-86860338`. Último acceso: mayo 2019.

[3] Ke Wan Ching and Manmeet (Mandy) Mahinderjit Singh. Wearable technology devices security and privacy vulnerability analysis. *International Journal of Network Security and Its Applications*, 8:19–30, 05 2016.

[4] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. Is the data on your wearable device secure? an android wear smartwatch case study. *Software: Practice and Experience*, 47(3):391–403, 2017.

[5] watchOS by Apple. `https://www.apple.com/es/watchos/`. Último acceso: junio 2019.

[6] Wear OS by Google. `https://wearos.google.com/`. Último acceso: junio 2019.

[7] Web de análisis de tráfico web StatCounter. `http://gs.statcounter.com/os-market-share/mobile/worldwide/`. Último acceso: mayo 2019.

[8] Mike Ryan. Bluetooth: With low energy comes low security. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Washington, D.C., 2013. USENIX. (Descargar).

[9] Youngmin Shin. A smartphone-based secure transaction processing with smartwatches. 2015. (Descargar).

[10] Sleep as Android. `https://play.google.com/store/apps/details?id=com.urbandroid.sleep&hl=es`. Último acceso: abril 2019.

[11] API pública de Sleep as Android. `https://sleep.urbandroid.org/documentation/developer-api/`. Último acceso: abril 2019.

[12] Photo Gallery for Wear OS (Android Wear). `https://play.google.com/store/apps/details?id=com.appfour.wearphotos&hl=es`. Último acceso: junio 2019.

[13] Google Fit. `https://play.google.com/store/apps/details?id=com.google.android.apps.fitness`. Último acceso: junio 2019.

[14] William Stallings. Public-key cryptography and rsa. In *Cryptography and Network Security: Principles and Practice*, chapter 9, pages 266–291. Pearson, 2011.

[15] William Stallings. Block ciphers and the data encryption standard. In *Cryptography and Network Security: Principles and Practice*, chapter 3, pages 77–85. Pearson, 2011.

[16] J. Orlin Grabbe. The DES Algorithm Illustrated. *Laissez Faire City Times*, 2(28), 1997.

[17] Joshep Selvanayagam, Akash Singh, Joans Michael, and Jaya Jeswani. Secure file storage on cloud using cryptography. *International Research Journal of Engineering and Technology*, 5(3):2044–2047, 2018.

[18] William Stallings. Advanced encryption standard. In *Cryptography and Network Security: Principles and Practice*, chapter 5, pages 148–166. Pearson, 2011.

[19] Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. A novel parity bit scheme for sbox in aes circuits. In *2007 IEEE Design and Diagnostics of Electronic Circuits and Systems*, pages 1–5. IEEE, 2007.

[20] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. Block cipher modes. In *Cryptography Engineering: Design Principles and Practical Applications*, chapter 4. Wiley, 2010.

[21] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[22] Juliano Rizzo and Thai Duong. Practical padding oracle attacks. In *WOOT*, 2010.

[23] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[24] AmILab. `http://amilab.ii.uam.es`. Último acceso: junio 2019.

[25] Alenta. `https://www.alenta.org/`. Último acceso: junio 2019.

[26] Cipher documentation. `https://developer.android.com/reference/javax/crypto/Cipher`. Último acceso: junio 2019.

[27] 1Password. `https://1password.com/es/`. Último acceso: junio 2019.

[28] 1Password. `https://blog.1password.com/guess-why-were-moving-to-256-bit-aes-keys/`. Último acceso: junio 2019.

[29] Antoine Joux. Authentication failures in nist version of gcm. *NIST Comment*, page 3, 2006. (Descargar).

[30] Niels Ferguson. Authentication weaknesses in gcm. *Comments submitted to NIST Modes of Operation Process*, 2005. (Descargar).

[31] Firebase by Google. `https://firebase.google.com`. Último acceso: junio 2019.

[32] Trevor Fisher, Derek Funk, and Rachel Sams. THE BIRTHDAY PROBLEM AND GENERALIZATIONS. (Descargar).

[33] TimmingLogger de Android. `https://developer.android.com/reference/android/util/TimingLogger.html`. Último acceso: junio 2019.

Acrónimos

AES Advanced Encryption Standard.

AmILab Ambient Intelligence Laboratory.

API Application Programming Interface.

CBC Cipher-Block Chaining.

CFB Cipher Feedback.

CTR Counter.

CTS Ciphertext Stealing.

DES Data Encryption Standard.

ECB Electronic Codebook.

EPS Escuela Politécnica Superior.

GCM Galois Counter Mode.

MAC Message Authentication Code.

MD5 Message-Digest Algorithm 5.

NSA National Security Agency.

OFB Output Feedback.

RSA Rivest-Shamir-Adleman.

SHA-1 Secure Hash Algorithm 1.

SHA-2 Secure Hash Algorithm 2.

TDEA Triple Data Encryption Algorithm.

TEA Trastornos del Espectro Autista.

UAM Universidad Autónoma de Madrid.