UNIVERSIDAD DE GRANADA

TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO OFICIAL EN CIENCIA DE DATOS E INGENIERÍA DE COMPUTADORES

Seguridad de los Datos en Sistemas Empotrados

Autor
Cristian González Guerrero

Tutores
Antonio Francisco Díaz García
Diego Pedro Morales Santos

ESCUELA INTERNACIONAL DE POSGRADO

—

Granada, Septiembre 2018

UNIVERSIDAD
DE GRANADA

Seguridad de los Datos en Sistemas Empotrados

Autor
Cristian González Guerrero

Directores
Antonio Francisco Díaz García
Diego Pedro Morales Santos

Seguridad de los Datos en Sistemas Empotrados

Cristian González Guerrero

Palabras clave: Internet de las Cosas, IoT, Ciberseguridad, TLS, Microcontroladores, SmartHome

Resumen

La seguridad de los datos ya no se limita a las aplicaciones militares o industriales, sino que es muy relevante en la electrónica de consumo. Existen varios protocolos de seguridad cuyo propósito es proteger la privacidad y veracidad de la información transmitida. De entre estos protocolos, TLS (*Transport Layer Security*) es uno de los estándares de seguridad más importantes en Internet. Integrar protocolos de seguridad en sistemas empotrados tiene una importancia cada vez mayor, debido al surgimiento de tecnologías como *SmartHome*, *SmartCity*, y en resumen, al paradigma emergente conocido como Internet de las Cosas (IoT).

En este trabajo presentamos el proceso de integración de una biblioteca TLS en un sistema empotrado, cuya misión es intercomunicar una red local de sensores con la nube de Amazon Web Services (AWS IoT). Esta integración se completa con la adición de AWS IoT Device SDK al sistema, de forma que la comunicación sea efectiva.

Seguridad de los Datos en Sistemas Empotrados

Cristian González Guerrero

Keywords: Internet of Things, IoT, Cybersecurity, TLS, Microcontrollers, SmartHome

Abstract

Data security is critical in our days. Its great importance is not anymore limited to military or industrial applications. In contrast, data security is now an important issue in consumer electronics. There are several security protocols available, aimed to protect both privacy and veracity of transmitted information. Among these protocols, TLS (Transport Layer Security) is one of the most important Internet security standards. Integrating security protocols in embedded systems has an increasingly importance, due to the arising of technologies such as SmartHome, SmartCity, and briefly, an emergent paradigm known as Internet of Things (IoT).

This work presents the process of integration of a TLS library in an embedded system. This system is a gateway communicating a local sensor network with Amazon Web Services cloud for IoT (AWS IoT). The integration is completed with the addition of AWS IoT Device SDK to the system, making the communication effective.

Yo, **Cristian González Guerrero**, alumno de la titulación Máster Universitario Oficial en Ciencia de Datos e Ingeniería de Computadores de la **Escuela Internacional de Posgrado de la Universidad de Granada**, con DNI 53741253M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Máster en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Cristian González Guerrero

Granada a 12 de septiembre de 2018.

D. **Antonio Francisco Díaz García**, profesor del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

D. **Diego Pedro Morales Santos**, profesor del Departamento de Electrónica y Tecnología de Computadores de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Seguridad de los Datos en Sistemas Empotrados***, ha sido realizado bajo su supervisión por **Cristian González Guerrero**, y autorizan la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 12 de septiembre de 2018.

Los directores:

Antonio Francisco Díaz García **Diego Pedro Morales Santos**

Agradecimientos

A mis tutores, Antonio Díaz García y Diego Pedro Morales Santos, por la confianza y apoyo que me han brindado.

A eesy-innovation, por la oportunidad de realizar un trabajo académico con una aplicación industrial directa. En especial, a Antonio David Escobar Molero y a Francisco Cruz Ramírez, por hacer posible esta colaboración, por coordinar las entregas de este trabajo y por proporcionarme una experiencia en la industria altamente enriquecedora.

A todo el personal de la Universidad de Granada que ha trabajado en esta labor de transferencia del conocimiento. En especial, a Diego Pedro Morales Santos, que ha coordinado la relación con la empresa, y a Yolanda García Avilés, que me ha guiado en el laberinto de la burocracia.

A Miguel Damas Hermoso y a Gustavo Romero López, profesores de asignaturas relacionadas con este proyecto, por ofrecerme un marco sobre el que estructurar el conocimiento y por instarme a aplicar los contenidos de sus asignaturas en el proyecto.

A Javier Ruiz Riquelme y a Hilda Patricia Palencia, por sus comentarios sobre este trabajo pero sobre todo por su paciencia, que se extiende más allá de lo puramente académico o profesional.

A todas las personas que me inspiran a seguir avanzando y a ir siempre más allá, en especial a Francisco Javier Romero Maldonado, por contagiarme su afán de superación, a Carlos Sampedro Matarín, por sacarme de mi zona de comfort de forma inesperada, y a Francisco Valderas Jiménez, por mantenerme cuestionándolo todo y mirando desde nuevas perspectivas.

A los que de forma más o menos directa hacen que sea quien soy, dándome alas para hacer siempre algo nuevo y aprender cosas diferentes: Alfonso Salinas Castillo, Luis Parrilla Roure, Alberto Molina Cabrera, José María González Medina, y los que no menciono aquí por no seguir desgastando su nombre.

Por supuesto, a mi madre, a mi hermano, Érik, y a mi buen amigo Javier Aguilar Sánchez, por estar siempre ahí cuando los necesito y por acompañarme en el camino de la vida.

Índice general

Capítulo 1

Presentación del proyecto

1.1. Motivación

De entre todas las revoluciones de Internet, el Internet de las Cosas (*Internet of Things*, IoT) es probablemente la que tiene mayor impacto en la actualidad [1, 2]. Este nuevo paradigma, que supone la conexión masiva de dispositivos de uso cotidiano a Internet, posibilita la creación de nuevos servicios y propuestas de valor, como los *SmartHomes*, *SmartCities* o la Industria 4.0 [3]. Muchas empresas ven nuevas oportunidades de negocio fundamentadas en esta tecnología. Así, por ejemplo, Google, IBM y Amazon tienen disponibles sus plataformas de computación en la nube específicas para IoT [4]. Estas plataformas habilitan el desarrollo de productos y servicios que proporcionan valor al usuario final, como pueden ser las lámparas inteligentes de Phillips «hue» (https://www.meethue.com/) o el kit para SmartHome de eesy-innovation «H2» (https://www.h2-smart.com/).

Sin embargo, el hecho de que los dispositivos estén conectados a la Internet, hace que estos sean potencialmente vulnerables a los ciberataques, que podrían ser perpetrados por un atacante conectado desde cualquier lugar del mundo. Estas potenciales vulnerabilidades podrían afectar directamente a los servicios ofrecidos a los usuarios, pudiendo llegar a incurrir en daños materiales o incluso personales [5, 6]. Un escenario pesimista frecuentemente citado en la literatura es el uso de la nueva masa de dispositivos para llevar a cabo ataques de denegación de servicio distribuidos (*distributed denial-of-service*, DDoS), con las nefastas consecuencias que esto conllevaría. Por este motivo, la seguridad[1] juega un rol fundamental en todos los sistemas IoT, y deberá considerarse desde el diseño inicial del sistema.

Muchos productos IoT son pequeños dispositivos que se basan en sistemas empotrados. Los sistemas empotrados tienen unas importantes limitaciones computacionales y suelen funcionar de forma independiente [7]. Tal vez por esto tradicionalmente no se ha considerado la seguridad en términos de proteger el sistema de ataques externos. Sin embargo, con el nuevo paradigma IoT, muchas son las empresas que requieren este tipo de seguridad en sus dispositivos. En muchas ocasiones, esto viene impuesto por los proveedores de servi-

[1] Aunque el término seguridad es muy amplio y tiene diferentes acepciones cuando se usa en términos informáticos, en el presente trabajo lo usaremos para referirnos a la seguridad de los datos (ciberseguridad), a menos que se indique lo contrario.

cios de computación en la nube, como Amazon Web Services, que exige unos niveles de seguridad importantes en la conexión, pues solo de esta forma puede el sistema ser robusto y seguro. Por su parte, las empresas desarrolladoras de productos finales que deseen proporcionar un servicio IoT de calidad tendrán que ajustarse a los estándares impuestos por estos proveedores de servicio.

1.2. Objetivos

Este trabajo fin de máster ha sido realizado en relación a un contrato concedido por la empresa eesy-innovation[2]. eesy-innovation (https://www.eesy-innovation.com/) es una empresa alemana, con sede en Múnich, que ofrece servicios de ingeniería a sus numerosos socios en la industria. Gracias a su fuerte departamento de I+D y a su dilatada experiencia en electrónica y telecomunicaciones, eesy-innovation ha colaborado en proyectos nacionales y europeos, como TreuFunk y NexGen. En la actualidad, la empresa está desarrollando una solución tecnológica para la automatización del hogar (SmartHome). Su gama de productos H2 permitirá al usuario controlar, gestionar y monitorizar los dispositivos conectados en casa de forma remota, haciendo uso de tecnologías IoT. Desde un punto de vista tecnológico, los productos H2 son sistemas embebidos basados en los microcontroladores XMC™ de la multinacional alemana Infineon, con la que eesy-innovation colabora de forma continuada.

El objetivo principal del presente trabajo es interconectar los dispositivos H2 con la nube de Amazon (Amazon Web Services IoT, AWS IoT). Esto se llevará a cabo a través de un dispositivo especial que se incluye dentro de la gama de productos, denominado *H2 miniGateway* (o simplemente miniGW). Como su nombre indica, este dispositivo no es más que una puerta de enlace que interconecta la red de sensores y actuadores con la Nube. La elección de la nube de Amazon ha sido realizada por la empresa tras considerar una serie de factores estratégicos y tecnológicos, entre los que se encuentra la seguridad y la escalabilidad del sistema. Tras la consecución del proyecto, se espera que el miniGW integre las características necesarias para comunicarse con AWS IoT, con el nivel de seguridad que esta plataforma exige.

De forma más concreta, los objetivos perseguidos en este trabajo fin de máster son los siguientes:

1. Estudio de los conceptos de seguridad en redes, en especial los que aplican a sistemas empotrados y al nuevo paradigma de IoT. Breve presentación de TLS y de X.509.

2. Análisis de las implementaciones TLS para sistemas empotrados y elección de una de ellas.

3. Integración de la biblioteca TLS seleccionada en el sistema objetivo (miniGW).

4. Estudio y presentación de la nube AWS IoT.

[2]No usamos mayúsculas para ajustarnos a la imagen corporativa de la empresa.

5. Integración del AWS IoT Device SDK en el sistema objetivo. Realización de una prueba de concepto.

1.3. Estructura de la memoria

La memoria se estructura en tres grandes partes, que describimos a continuación.

Fundamentos teóricos y presentación de la tecnología

La primera parte está dedicada a la presentación de las bases teóricas que permitirán una mejor comprensión del desarrollo del proyecto.

En primer lugar, llevaremos a cabo una exposición de los principios básicos de seguridad (capítulo 2), prestando especial atención a los requisitos de seguridad en sistemas empotrados (capítulo 3).

A continuación presentaremos brevemente el protocolo TLS (capítulo 4), los certificados digitales X.509 (capítulo 5) y la estructura de la nube de Amazon AWS IoT (capítulo 6).

Trabajo realizado

En la segunda parte desarrollaremos con detalle el trabajo realizado durante la consecución de este proyecto.

Comenzaremos esta parte con una presentación pormenorizada del sistema objetivo (capítulo 7), a la que seguirá el análisis de las implementaciones TLS existentes y se justificará la elección de mbedTLS (capítulo 8).

También presentaremos aquí los detalles de la integración de TLS en el sistema, así como los problemas que hemos tenido que resolver durante la misma (capítulo 9). Pondremos fin a esta parte con los detalles de la integración del Kit de Desarrollo de Software para dispositivos embebidos programados en C de Amazon, *AWS IoT Device SDK for embedded C* (capítulo 10). En cada uno de estos capítulos explicaremos las verificaciones y tests llevados a cabo, exponiendo los resultados obtenidos y planteando los problemas que hemos enfrentado.

Conclusiones y trabajo futuro

Para terminar, presentaremos las conclusiones obtenidas de la realización de este proyecto y disertaremos acerca de las posibles mejoras que podemos llevar a cabo en nuestro sistema (capítulo 11).

Parte I

Fundamentos teóricos y presentación de la tecnología

Capítulo 2

Principios básicos de criptografía

El propósito de la criptografía es convertir un mensaje o fichero (*texto plano*) en algo que a priori carece de sentido (*criptograma*), de forma que solo las personas autorizadas pueden realizar la transformación inversa para acceder al contenido original. La operación de transformar el texto plano en un criptograma es conocida como *encriptación* o *cifrado*, mientras que la operación inversa se denomina *desencriptación* o *descifrado*[1]. Al contrario de lo que nos dice la intuición, uno de los fundamentos de la criptografía moderna, el principio de Kerckhoff, afirma que los algoritmos de cifrado y descifrado deben ser públicos. El uso de algoritmos secretos es una táctica conocida como *seguridad por oscuridad*, y suele acarrear severos problemas de seguridad [7]. En su lugar, el único elemento que debe permanecer en secreto es la *clave*, que se usa como parámetro de entrada en los algoritmos de cifrado y descifrado. De este modo, aunque los algoritmo de cifrado y descifrado sean conocidos, un potencial atacante que tenga acceso al criptograma no podrá descifrar el mensaje original si no posee la clave.

A continuación formalizaremos los procesos de encriptación y desencriptación. Esto será útil para discurrir acerca de las diferentes técnicas de cifrado de las que haremos uso en este trabajo. Llamemos P al texto plano, K_E a la clave de encriptación, C al criptograma y E al algoritmo o función de encriptación. De esta forma, podemos definir el proceso de encriptación con la ecuación $C = E(P, K_E)$. Esto sugiere que para obtener el criptograma es necesario aplicar el algoritmo de encriptación sobre el texto plano y la clave de encriptación, que son tomados como parámetros. De forma similar, podemos definir el proceso de desencriptación con la ecuación $P = D(C, K_D)$, donde D es el algoritmo de desencriptación y K_D es la clave de desencriptación, que puede sr distinta a la clave de encriptación. Esto indica que la obtención del texto plano a partir del criptograma pasa por aplicar la función de desencriptación a este criptograma, tomando como segundo parámetro la clave de desencriptación. La relación entre estas operaciones queda ilustrada en la figura 2.1.

Aunque existe una multitud de algoritmos de encriptación y desencriptación disponibles, todos ellos caen en alguna de estas dos categorías: criptografía de clave simétrica y criptografía de clave pública. Estos dos tipos de cifrado se expondrán con detalle en los

[1] Por su similitud al idioma anglosajón, preferiremos los términos encriptación (*encryption*) y desencriptación (*decryption*), a pesar de la disonancia de los mismos.

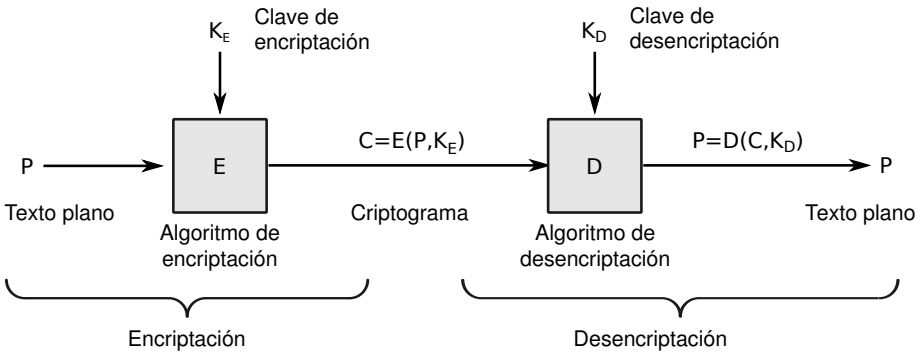


Figura 2.1: Relación entre el texto plano y el texto cifrado.

siguientes epígrafes.

2.1. Criptografía de clave simétrica

Los algoritmos de cifrado *de clave simétrica* son llamados así porque usan la misma clave en la encriptación y en la desencriptación. La tabla 2.1 describe algunas características de los algoritmos criptográficos de clave simétrica más conocidos, entre los que se encuentran DES (*Data Encription Standard*) y AES (*Advanced Encription Standard*). A continuación presentaremos los fundamentos de los algoritmos de *cifrado por bloques*, que consiste en encriptar el texto plano dividiéndolo en bloques de *n* bits, produciendo así el criptograma en bloques de la misma longitud.

Algoritmo de cifrado	Autor	Longitud de clave	Comentarios
DES	IBM	56 bits	Vulnerable
RC4	Ronald Rivest	1-2048 bits	Algunas claves vulnerables
RC5	Ronald Rivest	128-256 bits	Bueno, pero patentado
AES (Rijndael)	Daemen and Rijmen	128-256 bits	Mejor elección
Triple DES	IBM	168 bits	Bueno, pero antiguo
Twofish	Bruce Schneier	128-256 bits	Muy robusto y usado

Cuadro 2.1: Algunos de los algoritmos de clave simétrica más comunes [8].

Para ilustrar los principios del cifrado por bloques, recurriremos a algunos aspectos de su implementación en hardware. De este modo, podemos decir que que cualquier método de cifrado por bloques puede descomponerse en una sucesión de dos elementos hardware muy sencillos: las *cajas de permutación* (P-box) y las *cajas de sustitución* (S-box), que se ilustran en la figura 2.2.

Las cajas de permutación producen una reordenación entre los bits de su entrada y de su salida, sin modificar el número de ceros o de unos presentes. Esto puede conseguirse a través de un cableado arbitrario entre cada uno de los bits de entrada y de salida. De esta forma, la operación de permutación solo requiere la propagación de la señal, que se lleva

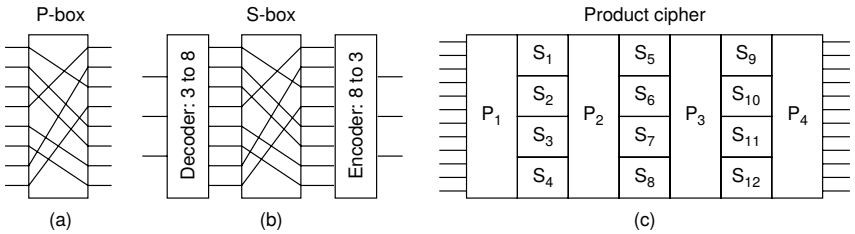


Figura 2.2: Elementos básicos del cifrado por bloques [8]. (a) P-box. (b) S-box. (c) Producto.

a cabo sin coste computacional alguno. Una caja de permutación correctamente diseñada será capaz de efectuar cualquier permutación deseada reconfigurando su cableado. Como puede comprobarse, las cajas de permutación obedecen al principio de Kerckhoff: el método utilizado (la permutación de bits) es público, lo que se reserva es el orden en que se realiza este cableado, que hará las veces de clave.

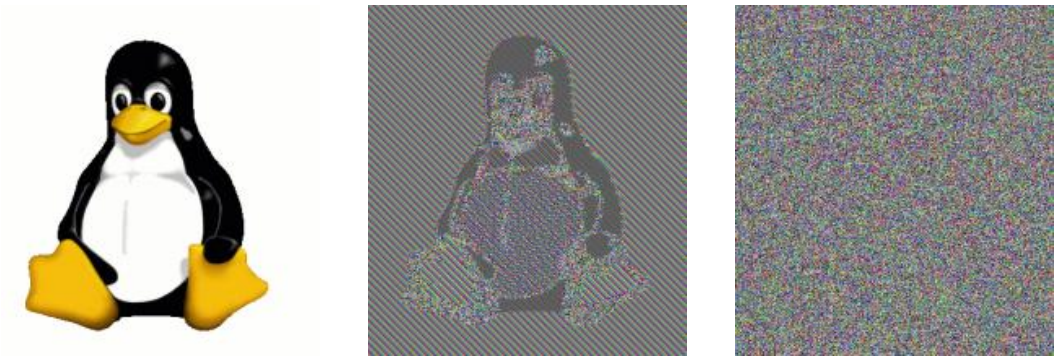
Por otro lado, las cajas de sustitución cuentan con tres etapas. La primera de ellas es un decodificador, que en este ejemplo es de 3 bits, por lo que se seleccionará una de las 8 líneas, estableciéndose el valor de esta a 1 y permaneciendo el resto a 0. La segunda etapa es una P-box. La tercera etapa codifica la línea de entrada seleccionada nuevamente a binario. Con el cableado mostrado, la secuencia de entrada compuesta por los números octales 01234567 producirá la secuencia de salida 24506713. De nuevo, mediante el cableado apropiado de la P-box dentro de la S-box, se puede realizar cualquier sustitución. Además, la implementación hardware de este dispositivo requiere muy pocas puertas lógicas, por lo que es posible lograr una gran velocidad, con un tiempo de propagación muy inferior al nanosegundo.

2.1.1. Modos de cifrado

Modo Electronic Code Book (ECB)

Cualquier método de cifrado por bloques, sin importar lo complejo que sea, puede verse como un diccionario que relaciona de forma única todas las posibles combinaciones de n bits de texto plano con otra combinación (normalmente distinta) del mismo número de bits en el criptograma. De este modo, podemos decir que cualquier cifrado por bloques actúa como un «diccionario» que permite traducir, en función de la clave utilizada, cualquier bloque de texto plano a otro bloque de la misma longitud del criptograma.

Cuando el algoritmo de cifrado actúa del modo descrito diremos que se está usando el modo de cifrado conocido como *Electronic Code Book*. Este modo es el más directo y el más eficiente en su implementación. Sin embargo, presenta una vulnerabilidad, que se hace más importante cuanto menor es tamaño del bloque. Puesto que un bloque de texto plano determinado siempre dará como resultado el mismo bloque de criptograma, la confidencialidad puede verse comprometida por ataques basados en criptoanálisis, como se muestra en la figura 2.3. Además de la confidencialidad, este modo de cifrado puede sufrir de ataques de reproducción (*replay attacks*) si no se toman medidas para evitarlos.



(a) **(b)** **(c)**

Figura 2.3: Comparación entre los modos de cifrado por bloques [9]. (a) Imagen sin cifrar (texto plano). (b) Imagen encriptada usando el modo ECB (vulnerable a ataques de criptoanálisis). (c) Imagen encriptada usando otro modo de cifrado (se consigue un criptograma pseudo-aleatorio).

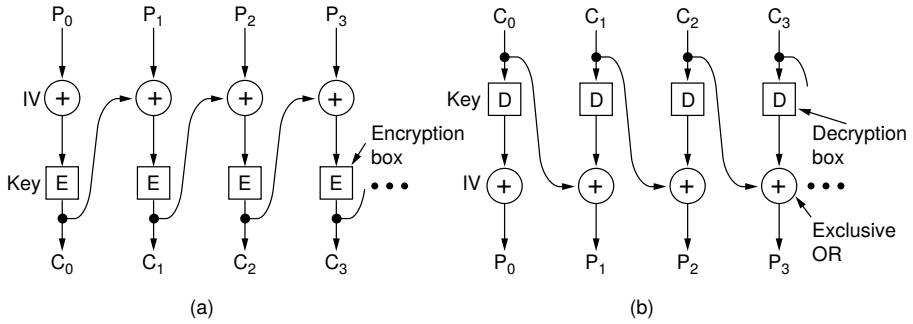


Figura 2.4: Modo de encriptación Cipher Block Chaining [8]. (a) Encriptación. (b) Desencriptación.

Modo Cipher Block Chaining (CBC)

Para evitar este tipo de ataques, todos los cifrados por bloque se pueden encadenar de varias formas. Una de estas formas es la conocida como *Cipher Block Chaining* (CBC). Con este método, mostrado en la figura 2.4, cada bloque de texto plano es mezclado con el bloque de criptograma anterior antes de ser encriptado. Esto se consigue gracias a la función XOR y sus interesantes propiedades. Por consiguiente, un mismo bloque de texto plano ya no se traduce al mismo bloque de criptograma, y el cifrado ya no es un diccionario de sustitución monoalfabético. El primer bloque se mezcla (XOR) con un vector de inicialización (*initialization vector*, IV) elegido al azar, que se transmite, en texto plano, junto con el texto cifrado.

Podemos ver cómo funciona el modo CBC examinando el ejemplo de la figura 2.4. Comenzamos calculando $C_0 = E(P_0 \,\mathrm{XOR}\, IV)$. Luego calculamos $C_1 = E(P_1 \,\mathrm{XOR}\, C_0)$, y así sucesivamente. El proceso de desencriptación también usa XOR para revertir el proceso, con $P_0 = IV \,\mathrm{XOR}\, D(C_0)$, y así sucesivamente. Cabe destacar que el cifrado del bloque i es una función de todo el texto plano en los bloques 0 a $i-1$, por lo que el mismo bloque de texto plano genera un criptograma diferente dependiendo de dónde se encuentre en el mensaje.

En este caso, un ataque que consista en modificar un bloque del criptograma producirá un mensaje sin sentido a partir del bloque modificado cuando el mensaje se desencripte. El modo CBC también tiene la ventaja de que el mismo bloque de texto plano no dará como resultado el mismo bloque de criptograma, lo que hace que el criptoanálisis sea mucho más difícil.

Otros modos de cifrado

Existen otros modos de cifrado además de los presentados aquí. Todos ellos tienen la ventaja, al igual que CBC, de producir una salida pseudo-aleatoria, que complica el criptoanálisis. Sin embargo, existen diferencias significativas entre ellos. Por ejemplo, el modo de cifrado por contador (CTR) no encadena los bloques de texto plano mezclándolos con el criptograma anterior, sino que usa un IV que se incrementa con cada bloque. De esta forma, CTR produce una salida pseudo-aleatoria, permitiendo el acceso no secuencial a la información, siempre y cuando se disponga de la clave, el IV inicial y el número de secuencia.

2.2. Criptografía de clave pública

La criptografía *de clave pública*, también conocida como criptografía asimétrica, parte de la idea de tener dos claves distintas para encriptar y para desencriptar. De este modo, tendremos un algoritmo de encriptación E, parametrizado con la clave K_E, y un algoritmo de desencriptación D, parametrizado con la clave K_D. Los algoritmos criptográficos de clave pública deben cumplir ciertas propiedades para garantizar la confidencialidad de los mensajes enviados, y son las que siguen [8]:

1. $D(E(P)) = P$.
2. Es extremadamente difícil deducir D a partir de E.
3. E debe resistir ataques de texto plano escogido (*chosen-plaintext attack*).

Veamos cómo funciona la criptografía de clave pública con un ejemplo sencillo. En este ejemplo tendremos dos nodos que nunca han establecido comunicación entre sí, pero que desean comunicarse de forma segura. Como suele hacerse en criptografía, llamaremos *Alice* al primer nodo, y *Bob* al segundo. Utilizaremos la notación E_A para referirnos al algoritmo de encriptación parametrizado por la clave pública de Alice. De manera similar, el algoritmo de desencriptación (secreto) parametrizado por la clave privada de Alice será denominado D_A. Bob hace lo mismo, publicando E_B pero manteniendo D_B en secreto.

Se supone que tanto la clave de encriptación de Alice, E_A, como la clave de cifrado de Bob, E_B, se encuentran en archivos de lectura pública. Ahora Alice toma su primer mensaje, P, calcula $E_B(P)$ y lo envía a Bob. Bob lo descifra aplicando su clave secreta D_B (es decir, calcula $D_B(E_B(P)) = P$). Nadie más puede leer el mensaje encriptado, $E_B(P)$, porque se supone que el sistema de cifrado es robusto y porque es demasiado difícil derivar D_B a partir

de E_B, que es la única clave de Bob conocida públicamente. Para enviar una respuesta, R, Bob transmite $E_A(R)$. Alice y Bob ahora pueden comunicarse de forma segura.

Como vemos, la criptografía de clave pública requiere que cada usuario tenga dos claves: una clave pública, utilizada por todo el mundo para encriptar los mensajes que se enviarán a ese usuario, y una clave privada, que el usuario necesita para descifrar los mensajes. Constantemente nos referiremos a estas claves como las claves pública y privada, respectivamente, y las distinguiremos de las claves secretas utilizadas para la criptografía de clave simétrica convencional [8].

Encontrar algoritmos que satisfagan los tres requisitos mencionados anteriormente no es fácil, pero existen algunos algoritmos publicados con estas propiedades. Uno de los más conocidos y usados hoy en día es RSA, descubierto en el MIT en los años 1970, y que toma el nombre de las iniciales de sus tres descubridores: Rivest, Shamir y Adleman [10]. RSA basa su seguridad en la dificultad de factorizar números muy grandes, en especial cuando estos son el producto de números primos elevados. A pesar de todos los intentos de romper su seguridad, RSA ha demostrado ser un algoritmo seguro y robusto durante más de 30 años. Su mayor desventaja es que requiere claves de al menos 1024 bits para una buena seguridad (frente a 128 bits para algoritmos de clave simétrica), lo que lo hace bastante lento. Por este motivo, muchos protocolos de seguridad solo usan criptografía de clave pública para realizar el intercambio seguro de una clave privada, pasando luego a comunicarse con criptografía simétrica, a través de un algoritmo parametrizado con esta clave.

Otros esquemas de clave pública se basan en la dificultad de calcular logaritmos discretos [11, 12]. Existen algunos otros esquemas, como los basados en curvas elípticas [13], que están disfrutando de una gran popularidad en nuestros días gracias a su eficiencia relativamente alta, pues son capaces de obtener niveles de seguridad equivalentes con claves más cortas.

2.3. Firmas digitales

Las firmas tradicionales se llevan usando mucho tiempo para establecer compromisos legales entre dos partes. En un esquema de comunicación, esto se traduce en garantizar una serie de situaciones [8]:

1. El receptor puede verificar la identidad del remitente.
2. El remitente no puede repudiar el contenido del mensaje más adelante.
3. El receptor no puede haber elaborado el mensaje por sí mismo.

Si pensamos en transacciones bancarias, por ejemplo, garantizar estas situaciones es de gran importancia. Sin embargo, en el dominio digital, no tenemos la posibilidad de firmar un documento en el sentido tradicional, ya que todos los documentos y mensajes están compuestos por un puñado de ceros y unos, todos iguales entre sí. Para verificar la autenticidad del documento y dar solución a las situaciones listadas anteriormente tendremos que hacer uso de las firmas digitales, de las que hablaremos a continuación.

Firmas digitales de clave pública

El problema de las firmas digitales puede resolverse de forma relativamente sencilla haciendo uso de la criptografía de clave pública. De hecho, este enfoque es el más usado hoy en día.

Supongamos que los algoritmos de encriptación y desencriptación de clave pública tienen la propiedad de que $E(D(P)) = P$ (además, por supuesto, de la propiedad habitual de que $D(E(P)) = P$). En este caso, Alice puede enviar un mensaje P firmado a Bob transmitiendo $E_B(D_A(P))$. Puesto que Alice conoce su propia clave privada, D_A y la clave pública de Bob, E_B, construir este mensaje es algo que Alice puede hacer. Cuando Bob recibe el mensaje, lo transforma usando su clave privada, como de costumbre, produciendo $D_A(P)$, al que aplica E_A para obtener el texto original [8].

Alice no podrá negar haber enviado el mensaje P a Bob, ya que su clave pública está disponible de forma abierta y por tanto es posible demostrar que $D_A(P)$ solo puede haber sido generado por ella. Sin embargo, este esquema trae consigo algunos problemas relacionados con la gestión de claves, como son el robo de la clave privada, o el cambio de clave. Algunos de estos problemas pueden ser resueltos con una infraestructura de clave pública (*public key infrastructure*, PKI). Hablaremos de infraestructuras de clave pública y de certificados digitales en el capítulo 5.

2.4. Funciones hash criptográficas

El proceso de firma explicado también puede llevarse a cabo empleando un hash del mensaje, en lugar del mensaje completo. Este método tiene dos ventajas. Por un lado, el hash será más corto que el mensaje, por lo que el proceso de firma será más eficiente, tanto en la generación como en la verificación. Por otro lado, separamos las funciones de autenticación y confidencialidad, por lo que no será necesario encriptar un mensaje si lo único que necesitamos es garantizar la autenticidad del remitente y evitar el repudio de la transacción.

El esquema que exponemos a continuación se basa en la idea de una función hash unidireccional que toma un texto plano de longitud arbitraria calcula una cadena de bits de longitud fija a partir de este. Esta función hash, a menudo denominada *message digest* (MD), debe cumplir cuatro propiedades importantes [8]:

1. Dado P, es fácil calcular $MD(P)$.
2. Dado $MD(P)$, es efectivamente imposible encontrar P.
3. Dado P, es extremadamente difícil encontrar P' tal que $MD(P') = MD(P)$.
4. Un cambio en la entrada, por pequeño que sea, producirá una salida muy diferente.

Computar el hash de un mensaje es mucho más eficiente que encriptarlo con un algoritmo de clave pública, por lo que los *message digests* pueden usarse para acelerar algoritmos de firma digital.

El funcionamiento de este tipo de firmas queda ilustrado en la figura 2.5. Como pue-

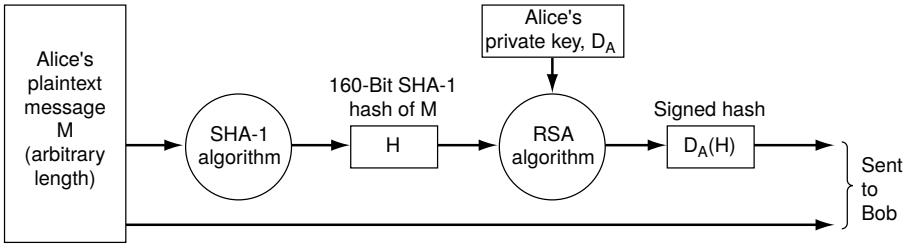


Figura 2.5: Uso del algoritmo seguro de hash SHA-1 y del algoritmo de clave pública RSA para la firma de un documento no confidencial [8].

de verse en la misma, el mensaje de Alice M se envía sin encriptar, junto a un hash firmado $D_A(H)$, donde $H = MD(M)$ usando el algoritmo de hash seguro SHA-1 [14]. Gracias a las propiedades de las funciones de hash, este esquema permite a Bob verificar la identidad de Alice y la integridad del mensaje en un paso, simplemente comprobando que $E_A(D_A(H)) = MD(M)$. En caso de que esto no pueda comprobarse, el mensaje puede haber sido modificado (o haber sido firmado por otra entidad), por lo que Bob deberá desconfiar del mismo. En cambio, si el resultado coincide, podremos estar seguros de que el mensaje no ha sido modificado en su transmisión, y que ha sido firmado, en este caso, por Alice.

Capítulo 3

Seguridad en sistemas empotrados

Como hemos visto en el capítulo 2, una parte importante de la ciberseguridad depende del uso de funciones y algoritmos complejos, que suelen estar relacionados con operaciones computacionalmente muy costosas. Sin embargo, los sistemas empotrados, como los usados en los nodos finales y en las pasarelas de las infraestructuras IoT, suelen disponer de recursos muy limitados. Esta escasez de recursos impondrá una serie de limitaciones, que van desde limitaciones computacionales (baja frecuencia de la CPU, escasa memoria) hasta limitaciones energéticas (sistemas autónomos alimentados por baterías). Por otro lado, los sistemas embebidos son responsables, en numerosas ocasiones, de llevar a cabo funciones críticas, que podrían poner en peligro la integridad física del usuario en caso de fallo. Debido a estos factores, la seguridad en sistemas empotrados debe ser tratada de un modo especial.

Aunque a continuación hablaremos de requisitos generales de seguridad, conviene mencionar que son muchas las entidades involucradas en la cadena de diseño, fabricación y uso de un sistema embebido típico. Los requisitos de seguridad dependerán de la perspectiva que consideremos, tal y como ilustra la figura 3.1, con el ejemplo de un teléfono móvil. En el ejemplo, el usuario estará interesado en mantener la confidencialidad y la integridad de los datos que almacena o transmite haciendo uso del sistema. El proveedor de hardware, por su parte, tratará de proteger la propiedad intelectual de los módulos provistos al diseñador del dispositivo. El proveedor de contenidos, en cambio, tratará de evitar la copia ilegal de los contenidos multimedia ofrecidos al usuario. De esta forma, las entidades no confiables (potencialmente maliciosas) también variarán dependiendo de la perspectiva considerada [15].

En las siguientes secciones daremos una definición para los sistemas empotrados y estudiaremos sus características más relevantes. También presentaremos algunos de los requisitos de seguridad fundamentales en cualquier sistema informático. Finalmente, introduciremos las soluciones de seguridad con las que se abordan estos requisitos. Con esto habremos dado las primeras pinceladas en la definición de los requisitos de seguridad de nuestro sistema.

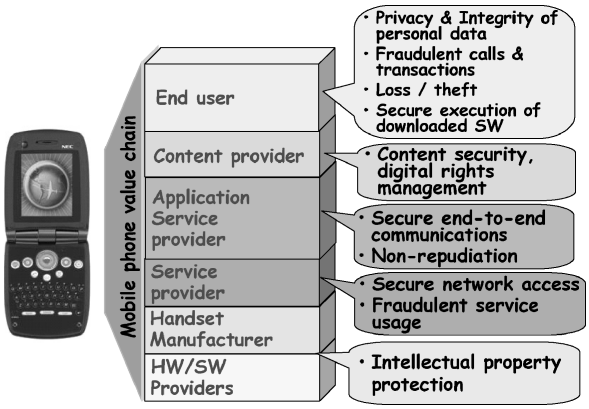


Figura 3.1: Actores implicados y requisitos de seguridad de un terminal móvil [15].

3.1. Particularidades de los sistemas empotrados

Los sistemas empotrados (también conocidos como sistemas embebidos o *embedded systems*) son sistemas basados en computador y que están diseñados para realizar una función concreta. Muchos sistemas empotrados llevan a cabo tareas a tiempo real, siendo frecuente su uso en el control de sistemas mecánicos, como ascensores o cadenas de montaje [16].

Además, estos sistemas están presentes en muchos electrodomésticos, como hornos microondas, lavadoras o reproductores de música. En este contexto, también los encontramos en el núcleo de muchos dispositivos cotidianos, como la calculadora, la cámara de fotos o el marcapasos [7].

Podría discutirse si algunos dispositivos como los *SmartPhones* o las videoconsolas de última generación son también sistemas empotrados. No obstante, estos dispositivos cuentan con un sistema operativo (*operating system*, OS) que les permite ejecutar aplicaciones de terceros bajo demanda del usuario. Por este motivo, estos dispositivos serán considerados en este trabajo como computadores de propósito general, a pesar de compartir algunas características con los sistemas empotrados.

Estas son las principales características que consideraremos para decir que un sistema es empotrado:

- Se trata de un sistema electrónico basado en microprocesador (normalmente microcontrolador).
- El sistema está diseñado para llevar a cabo una aplicación o tarea específica, ejecutando un software concreto que es instalado en el proceso de producción. El sistema no está diseñado para ejecutar software de terceros.
- El sistema cuenta con los periféricos necesarios para llevar a cabo su función. El sistema está diseñado para controlar este hardware específico, no permitiendo que se añadan nuevos periféricos.

- El sistema puede ser el núcleo de un dispositivo autónomo (como un mando a distancia), o puede formar parte de un sistema mayor (como las distintas unidades de control electrónico de un automóvil).

En aras de la exhaustividad, y aunque no es un problema que afecte directamente a este proyecto, destacaremos los requisitos temporales de algunos sistemas empotrados. En numerosas ocasiones, las tareas deben ser realizadas en tiempo real, proporcionando respuestas a ciertos eventos en un intervalo de tiempo controlado y reducido. Esto tiene gran relación con la tendencia a usar sistemas empotrados en la implementación de sistemas críticos. Los sistemas críticos deben ser altamente fiables, llevando a cabo su función de forma rigurosa y precisa. En caso de error, un sistema crítico puede desencadenar un fallo general del sistema, incurrir en enormes pérdidas económicas o incluso poner en riesgo la vida de personas.

3.2. Requisitos generales de seguridad

Como hemos mencionado anteriormente, los recursos computacionales de los sistemas empotrados suelen ser muy limitados. Esto supone un abaratamiento de costes que se hace muy relevante en la producción en masa de los dispositivos. Sin embargo, esta escasez de recursos, junto al *propósito no general* de estos sistemas, hace que estos carezcan de un sistema operativo propiamente dicho, con todas las consecuencias que esto conlleva.

En los sistemas empotrados más complejos es frecuente el uso de un planificador de tareas (*scheduler*) sencillo, o de un sistema operativo de tiempo real (*real-time operating system*, RTOS), con los que se facilita el desarrollo y ejecución de procesos concurrentes. Estas herramientas suelen permitir una buena gestión de las tareas de tiempo real, sin embargo no cuentan con los servicios proporcionados por un sistema operativo, como una pila de red, un sistema de ficheros, o la posibilidad de ejecutar software de terceros. Por este motivo, tendremos que estudiar cuidadosamente los requisitos de seguridad del sistema, ya que todos ellos deberán ser implementados por el equipo de desarrollo del mismo.

A continuación presentamos los requisitos de seguridad más relevantes en cualquier sistema informático [15].

Identificación de usuarios

Se trata del proceso de validar a los usuarios antes de permitirles usar el sistema. Este requisito es fundamental cuando el sistema proporciona el acceso a datos sensibles (como los valores de sensores en un *SmartHome*), a servicios restringidos (como llamadas telefónicas) o permite el control de otro sistema (como una cerradura electrónica o la calefacción).

La identificación de usuarios es especialmente importante cuando el sistema empotrado puede ser accedido con facilidad por personas no autorizadas, como son aquellos que están conectados a Internet.

Acceso seguro a la red

Consiste en proporcionar acceso a una conexión de red o a un servicio solo si el dispositivo está autorizado. Por ejemplo, un router doméstico no debería permitir el acceso a ningún dispositivo que no haya sido autorizado a usar la red. Del mismo modo, una pasarela IoT no deberá retransmitir datos de una red de sensores ajena.

Comunicación segura

La comunicación segura incluye varias funciones, que se listan a continuación. También proporcionamos explicaciones con ejemplos concernientes al presente proyecto.

1. Autenticar los extremos de la comunicación, ya sean personas o sistemas informáticos.

   En el caso que nos ocupa, necesitaremos un mecanismo para verificar la identidad de los nodos de la red. Solo así podremos asegurarnos que los datos mostrados en la terminal son, efectivamente, los medidos por nuestros dispositivos.

2. Garantizar la confidencialidad de los datos comunicados, evitando que puedan ser leídos por entidades no autorizadas.

   En un *SmartHome*, la disponibilidad de los datos de temperatura o de presencia de personas podrían ser usados para atentar contra la propiedad privada.

3. Garantizar la integridad de los datos comunicados, evitando que puedan ser modificados por entidades no autorizadas.

   Un posible atacante podría causar severas molestias, e incluso sembrar el terror en el hogar, enviando comandos aleatorios a las lámparas o a la caldera. A pesar de que el atacante no entendería el contenido de los mensajes, los habitantes de la casa podrían creerse víctimas de un *poltergeist*.

4. Evitar el repudio de transacciones ejecutadas de forma efectiva, como puede ser una transferencia bancaria.

   Como puede ser, también, el pedido de víveres efectuado de forma autónoma por un frigorífico inteligente correctamente configurado.

Almacenamiento seguro de datos

Consiste en asegurar la confidencialidad y la integridad de la información sensible almacenada en el sistema. Esta información solo podrá ser leída por usuarios o dispositivos autorizados a ello. Del mismo modo, solo los usuarios o dispositivos autorizados podrán modificar determinados datos. Por ejemplo, en el caso de una tarjeta bancaria, solo el usuario debería poder modificar la clave, y solo el banco debería poder modificar el saldo. En el caso que nos ocupa, solo el usuario del sistema podrá leer la información de los sensores almacenada en la Nube, y solo los sensores podrán actualizar dicha información.

Disponibilidad

Debe garantizarse que el sistema pueda realizar la función para la que fue diseñado y dar servicio a los usuarios legítimos en todo momento, no interrumpiéndose a causa de ataques de denegación de servicio (*denial of service*, DoS).

Resistencia a la manipulación *(Tamper resistance)*

Los ataques perpetrados contra los sistemas empotrados pueden explotar vulnerabilidades a distintos niveles, incluyendo aquellas presentes en el sistema operativo y las aplicaciones. Estos ataques pueden manipular datos o procesos sensibles (ataques a la integridad), revelar información confidencial (ataques a la privacidad) o denegar el acceso a los recursos del sistema (ataques a la disponibilidad). Por tanto, es necesario desarrollar y desplegar diversas medidas contra estos ataques, tanto a nivel software como a nivel hardware.

Fiabilidad del sistema

Para afirmar que un sistema es realmente fiable, no basta con que este funcione de forma correcta la mayor parte del tiempo, sino que debe verificarse que su funcionamiento es el esperado en todo momento. Si además el sistema ha de ser seguro, este debe funcionar incluso cuando es sometido a ataques de seguridad. La fiabilidad es un requisito fundamental cuando se trata de *sistemas críticos*.

3.3. Soluciones de seguridad

Existen múltiples técnicas con las que se trata de satisfacer los requisitos de seguridad definidos en la sección anterior. A continuación presentamos algunos de ellos, prestando especial atención a los que tienen más relevancia en la realización de este trabajo.

Protocolos de seguridad

Los protocolos de seguridad proporcionan formas de establecer canales de comunicación seguros entre dispositivos. Esto se consigue estableciendo mecanismos para asegurar la confidencialidad y la integridad de los mensajes, y autenticando cada uno de los extremos de la comunicación. La confidencialidad suele implicar el cifrado de los datos previa transmisión, normalmente con técnicas de criptografía simétrica. Por su parte, la integridad de los mensajes, que es esencial para evitar la manipulación de los mismos por un posible atacante, se consigue añadiendo un código de autenticación de mensaje (*message authentication aode*, MAC) a cada mensaje transmitido. Los códigos MAC se calculan mediante la aplicación de una función hash criptográfica al mensaje, y cifrando el resultado con una clave privada. La verificación de la autenticidad de los dispositivos puede conseguirse con el uso de certificados digitales. De esta forma, dos dispositivos pueden comunicarse de forma segura a través de una red a priori no segura.

Tal vez los protocolos de seguridad más conocidos sean *Internet Protocol Security* (IPSec) y *Transport Layer Security* (TLS), este último sucesor de *Secure Sockets Layer* (SSL). El uso de estos protocolos está muy extendido: el primero, en el despliegue de redes privadas virtuales (*virtual private networks*, VPN), el segundo, en la realización de transacciones de aplicación seguras en Internet. Estos protocolos son prácticamente transparentes para las capas de comunicación superiores. De este modo, una aplicación que se comunica en red de forma no segura puede pasar a establecer una comunicación segura con pocas modificaciones, sin necesidad de ser rediseñada desde cero. No obstante, los requisitos computacionales que supone el uso de estos protocolos pueden comprometer la capacidad computacional del sistema con gran facilidad, en especial cuando se trata de sistemas empotrados.

En el capítulo 4 presentamos las características principales del protocolo TLS, por su relevancia en el presente proyecto.

Certificados digitales de clave pública

Estos certificados digitales, de los que hablaremos más a fondo en el capítulo 5, proporcionan formas de asociar una identidad digital con una entidad física, basándose en criptografía asimétrica. Esto se consigue a través de un fichero, con el que es posible validar que la entidad física posee una determinada clave pública, gracias a la confianza depositada en una *autoridad certificadora* y en su *cadena de confianza*. El certificado incluye información sobre la clave, la identidad de su propietario (sujeto del certificado) y sobre la firma digital de la entidad.

Gestión de derechos digitales

Las herramientas de gestión de derechos digitales (*digital right management*, DRM) proporcionan entornos seguros para proteger contenidos digitales de un uso no autorizado. De esta forma, es posible evitar la copia de música con derechos de autor, así como la retransmisión pública de películas protegidas por las leyes de propiedad intelectual. Algunas herramientas de gestión de derechos digitales son los protocolos OpenIPMP, MPEG, ISMA, y MOSES [15].

Almacenamiento y ejecución seguros

Estas soluciones requieren que el sistema haya sido diseñado teniendo en cuenta las consideraciones de seguridad. Algunos ejemplos sencillos son el uso de hardware para monitorizar las transacciones en un bus, bloqueando acceso ilegales a zonas protegidas de la memoria, la autenticación del firmware que se ejecuta en el sistema, o el aislamiento de aplicaciones para preservar la privacidad e integridad tanto del código como de los datos asociados a cada proceso [15]. Destacaremos aquí el hecho de que algunos procesadores comerciales están diseñados para la ejecución segura, como es el caso de algunos ARM [17], y que existen sistemas operativos de tiempo real diseñados para este fin.

Verificación del software

Existe un gran abanico de verificaciones a las que se puede someter un software para comprobar que lleva a cabo la función para la que fue diseñado en todo momento. Esto puede hacerse de diversas formas. La verificación formal, por su parte, supone la comprobación (matemática) de que el software realiza su función sin importar las condiciones a las que es sometido. Este método de verificación es el más estricto de todos, siendo también el más fiable. Sin embargo, la verificación formal es muy costosa, por su dificultad y por el gran tiempo que conlleva. Otro método es el análisis estático, que supone la revisión del código fuente (o del código objeto) sin llevar a cabo la ejecución del programa. De esta forma es posible detectar muchos errores y algunas prácticas comunes que suponen un riesgo para la integridad de la ejecución. Por último, mencionaremos el análisis dinámico, que requiere la ejecución del programa en ambientes diversos, idealmente con todos los posibles valores en los parámetros de entrada de cada función.

Existen diversos estándares que especifican requisitos en la verificación de software. También podemos encontrar directrices que nos ayudan a escribir código libre de errores. Los siguientes son ejemplos de estos estándares y directrices.

FIPS-140 estándares de seguridad informática del gobierno de los Estados Unidos, que especifican requisitos para módulos criptográficos.

Evaluation Assurance Level (EAL) certificaciones para seguridad informática por niveles definidos por el estándar *Common Criteria for Information Technology Security Evaluation* (ISO/IEC 15408).

MISRA C conjunto de directrices para el desarrollo de software en el lenguaje de programación C, desarrollado por la *Motor Industry Software Reliability Association* (MISRA) y usada ampliamente en la industria de la automoción.

Puesto que la verificación de un programa completo suele ser muy costosa, es común que se verifiquen solo algunas partes del mismo, por ejemplo, algún módulo crítico.

Capítulo 4

El protocolo TLS

Transport Layer Security (TLS) es un protocolo de red cuyo principal objetivo es establecer una comunicación segura entre dos nodos de una red que no tiene por qué ser segura. Es el sucesor de *Secure Sockets Layer* (SSL), por lo que comparte muchas características con el mismo.

El protocolo, cuya versión más reciente es TLS 1.2, se compone de dos capas principales, denominadas *TLS Record Protocol* y *TLS Handshake Protocol*. En el nivel más bajo, justo por encima de algún protocolo de transmisión confiable[1] (como TCP), se encuentra el *TLS Record Protocol*. Este protocolo proporciona la seguridad de la conexión, que tiene dos propiedades básicas:

- **La conexión es privada**. Los datos son encriptados con algoritmos de criptografía simétrica (como AES, RC4, etc). Las claves usadas en esta encriptación simétrica se generan de forma única para cada conexión, a través de una negociación secreta que lleva a cabo otro protocolo (el *TLS Handshake Protocol*). Aunque no es común, el *TLS Record Protocol* también puede ser usado sin encriptación.
- **La conexión es confiable**. El transporte de mensajes incluye una comprobación de la integridad del mensaje, que se basa en un código MAC. El cálculo del código MAC se lleva a cabo haciendo uso de funciones de hash seguras (como SHA-1). Aunque no es común, el *TLS Record Protocol* también puede operar sin el uso de códigos MAC.

El *TLS Record Protocol* es usado para encapsular otros protocolos de más alto nivel. Uno de estos protocolos es el *TLS Handshake Protocol*, que permite la autenticación mutua de servidor y cliente y les permite negociar los algoritmos de encriptación y las claves antes de que comience la transmisión de los datos de aplicación. El *TLS Handshake Protocol* proporciona seguridad en la conexión que tiene tres características principales:

- Permite autenticar la identidad de los extremos, usando criptografía asimétrica[2] (co-

[1] Un protocolo de red confiable es aquel que garantiza que los datos serán entregados en su destino sin errores y en el mismo orden en que se transmitieron.

[2] Existe una modalidad que permite usar criptografía simétrica en la autenticación, a través de un sistema

mo RSA, DSA, etc.). Esta autenticación puede ser requerida por la aplicación, o alternativamente, hacerse de forma opcional.

- Garantiza la seguridad en la negociación de una clave secreta. La clave simétrica negociada no puede ser interceptada, ni siquiera por un atacante que se coloque entre los nodos y escuche toda la comunicación.

- Garantiza la integridad de la negociación, que será confiable, de forma que ningún atacante puede modificar la negociación sin ser detectado por los nodos.

TLS se sitúa en la capa de aplicación del modelo OSI. Esto quiere decir que son las aplicaciones las que deben incorporar una biblioteca que les habilite las capacidades de encriptación y de comunicación segura que proporciona este protocolo. Esto tiene varias implicaciones. Por un lado, los datos viajarán seguros desde la aplicación, sin que exista la posibilidad de que estos sean interceptados en una capa de red inferior. Por otro lado, el establecimiento de la conexión TLS deberá ser gestionado por la misma aplicación, no pudiéndose delegar al sistema operativo. Una ventaja de TLS es que es un protocolo independiente de la aplicación. Esto quiere decir que sobre él puede situarse cualquier protocolo de nivel superior de forma transparente.

4.1. Criterios de diseño del protocolo

De acuerdo a las especificaciones del protocolo TLS v1.2, los objetivos del mismo, en orden de prioridad, son los siguientes [20]:

1. Seguridad criptográfica: TLS debería usarse para establecer una conexión segura entre dos partes

2. Interoperabilidad: Programadores independientes deberían poder desarrollar aplicaciones usando TLS que puedan intercambiar parámetros criptográficos sin tener conocimiento del código de la otra parte.

3. Extensibilidad: TLS busca proveer una infraestructura en la que puedan incorporarse nuevos métodos de encriptación según sea necesario. De esta forma, se evita tener que definir nuevos protocolos de seguridad, con las consecuencias que esto tiene: la posibilidad de encontrar brechas de seguridad en el nuevo protocolo y la necesidad de implementarlo como una nueva biblioteca.

4. Eficiencia: Las operaciones criptográficas tienden a ser computacionalmente costosas, sobre todo cuando se trata de encriptación de clave pública. Por este motivo, el protocolo TLS trata de reducir el número de conexiones que se establecen desde cero, así como reducir la actividad de red.

de claves compartidas (PSK) [18, 19]. Esto puede ser útil en determinadas circunstancias, por ejemplo, si se requiere autenticación y los recursos computacionales son muy limitados.

4.2. El *TLS Record Protocol*

El *TLS Record Protocol* es la capa más baja del protocolo TLS, y garantiza la comunicación segura entre los extremos. Los mensajes que van a ser transmitidos son procesados por el *TLS Record Protocol* en los siguientes pasos: los mensajes son fragmentados en bloques de un tamaño manejable, luego estos bloques pueden ser comprimidos, al resultado se le aplica un código MAC y finalmente es encriptado, quedando listo para la transmisión. En el otro extremo, los datos recibidos son desencriptados, verificados, descomprimidos (si procede), reagrupados en los bloques originales y entregados a la capa superior.

Existen cuatro protocolos que hacen uso inmediato del *TLS Record Protocol*, y se presentan a continuación. Destacaremos que aunque estos son los protocolos definidos en la actualidad, el *TLS Record Protocol* puede dar soporte a otros tipos de mensaje (*content types*).

Handshake Protocol

Los parámetros criptográficos que se usarán en la sesión son acordados mediante el *TLS Handshake Protocol*. Este protocolo implica el intercambio de varios mensajes entre cliente y servidor, en los que se establece un secreto compartido, se lleva a cabo la autenticación de los nodos y finalmente, si se verifica que no se ha producido ningún intento de ataque, se configuran los parámetros del *TLS Record Protocol* para que pueda comenzar la transmisión de los datos de aplicación.

Puesto que este protocolo está íntimamente relacionado con los parámetros criptográficos que se usarán en la sesión, y dado que el uso de estos parámetros requiere que estén disponibles en la implementación TLS a usar, dedicaremos una sección completa al *TLS Handshake Protocol* más adelante en este mismo capítulo.

Alert Protocol

Los mensajes de alerta comunican la severidad de una alerta (*warning* o *fatal*) y su descripción. Un mensaje de alerta fatal resulta en la conclusión inmediata de la conexión. Como con el resto de protocolos, los mensajes de alerta son encriptados y comprimidos siguiendo las especificaciones del estado actual de la conexión.

Change Cipher Spec Protocol

El *TLS Change Cipher Spec Protocol* es usado para notificar un cambio en las estrategias de cifrado. El protocolo consiste en un mensaje único, que se comprime y encripta haciendo uso de los parámetros de cifrado actuales.

El mensaje *ChangeCipherSpec* es enviado tanto por el cliente como por el servidor para notificar que los mensajes enviados a continuación estarán protegidos atendiendo a las nuevas especificaciones de cifrado (*CipherSpec*) y claves establecidas.

Para ser más específicos, si un handshake tiene lugar durante una transmisión de datos, las partes pueden continuar comunicándose usando los parámetros establecidos en la anterior negociación. Es el mensaje *ChangeCipherSpec* el que señala el momento en el que las nuevas especificaciones serán usadas, por lo que cualquier mensaje recibido después de este deberá atender a los nuevos parámetros y claves de cifrado.

Application Protocol

Los datos de aplicación son transportados directamente por la *TLS Record Layer*, donde son fragmentados, comprimidos y encriptados atendiendo al estado de la conexión y a sus parámetros. De esta manera, los mensajes provenientes de la capa de aplicación son tratados de forma transparente por la *TLS Record Layer*.

4.3. El *TLS Handshake Protocol*

Como se ha visto anteriormente, TLS tiene tres subprotocolos con los que los extremos de la conexión pueden acordar los parámetros de seguridad para la *TLS Record Layer*, autenticarse e informarse sobre errores producidos en la conexión.

El *TLS Handshake Protocol* es responsable de negociar la sesión, cuyos principales elementos son:

session identifier secuencia de bytes arbitrarios elegidos por el servidor para identificar la sesión activa.

peer certificate certificado del cliente o del servidor, definido por el estándar X.509 V3. Este elemento puede no estar presente.

compression method algoritmo utilizado para comprimir datos antes de la encriptación.

cipher spec especifica la función pseudo-aleatoria (*pseudo-random function*, PRF) que se usará para generar las claves, el algoritmo de encriptación simétrica (como AES, etc.) y el algoritmo de MAC (como HMAC-SHA1). También define los atributos criptográficos como la longitud de MAC, entre otros.

master secret clave de 48 bytes compartida entre el cliente y el servidor.

Estos elementos son usados para crear los parámetros de seguridad que la *TLS Record Layer* usará para proteger los datos de aplicación. Distintas conexiones pueden ser instanciadas usando la misma sesión, gracias a la característica de reanudación del protocolo *TLS Handshake Protocol*.

Descripción funcional del *TLS Handshake Protocol*

Los parámetros criptográficos del estado de la sesión son establecidos por el *TLS Handshake Protocol*, el cuál opera sobre la capa *TLS Record Layer*. Cuando un cliente y un

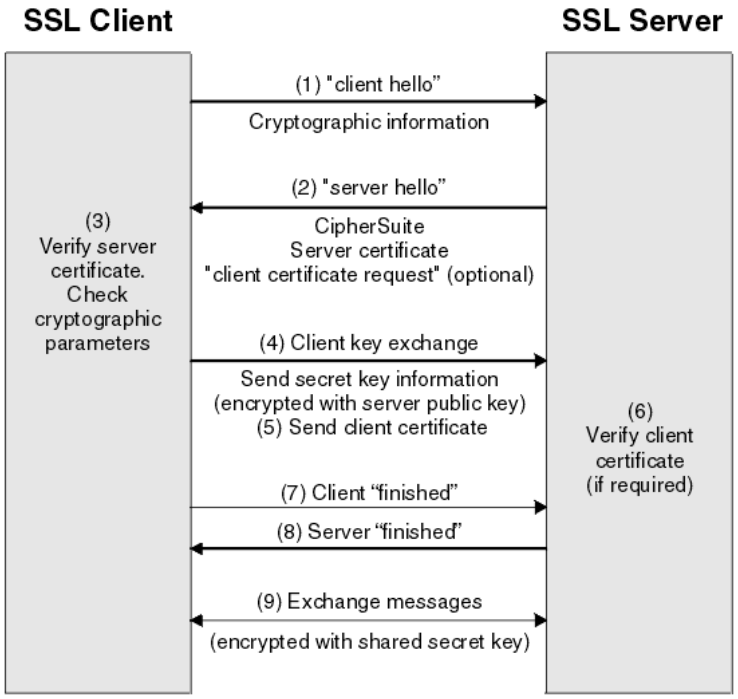


Figura 4.1: Establecimiento de la conexión por el *TLS Handshake Protocol* [21].

servidor TLS comienzan a comunicarse por primera vez, negocian una versión del protocolo, eligen los algoritmos criptográficos, se autentican mutuamente (de forma opcional), y usan técnicas de criptografía asimétrica para generar claves compartidas.

El *TLS Handshake Protocol* implica los siguientes pasos:

- Intercambio de mensajes *hello* para negociar los algoritmos, intercambiar valores aleatorios y comprobar si debe reanudarse una sesión anterior.
- Intercambio de los parámetros criptográficos necesarios para permitir que cliente y servidor puedan compartir una clave *pre-maestra* (*pre-master secret*).
- Intercambio de los certificados e información criptográfica para permitir que cliente y servidor puedan autenticarse mutuamente.
- Generación de una clave maestra (*master secret*) a partir de la clave pre-maestra e intercambiar valores aleatorios.
- Proporcionar parámetros de seguridad para la *TLS Record Layer*.
- Permitir que cliente y servidor verifiquen que el otro extremo ha calculado los mismos parámetros de seguridad y que por tanto el handshake se ha llevado a cabo sin la manipulación de un atacante.

La figura 4.1 muestra los pasos llevados a cabo en el establecimiento de la conexión TLS, de la que se encarga el *TLS Handshake Protocol*.

4.4. Resumen

TLS permite conexiones seguras a nivel de la capa de aplicación. El establecimiento de la conexión se lleva a cabo de forma segura, usando criptografía asimétrica para el intercambio de las claves compartidas, que serán usadas en el cifrado de los mensajes de aplicación. Durante el establecimiento de la conexión, conocido como handshake, los nodos pueden autenticarse mutuamente usando certificados digitales X.509.

El siguiente esquema resume las funcionalidades básicas de los distintos subprotocolos de TLS.

1. Handshake

   Se encarga del establecimiento de la conexión entre dos nodos (cliente y servidor). El handshake conlleva:

   a) Definición de los parámetros de seguridad a usar en la sesión (algoritmos de cifrado, clave, compresión, etc.), que son negociados entre cliente y servidor, de acuerdo con las funciones criptográficas implementadas en cada uno.

   b) Autenticación mutua de los nodos cliente y servidor (opcional, puede llevarse a cabo en una única dirección).

   El establecimiento de la conexión implica un esquema de criptografía asimétrica, usando claves relativamente largas generadas exclusivamente para la conexión a partir de una fuente de entropía. Este esquema es computacionalmente muy costoso, pero también muy seguro.

   a) Gracias al esquema de criptografía asimétrica, el contenido del propio handshake queda oculto para un observador externo, haciendo imposible deducir los parámetros de la sesión.

   b) El handshake cuenta con mecanismos de detección de *tampering*, por lo que un ataque sería detectado.

2. Intercambio de datos de aplicación

   Los datos de la capa superior son transmitidos de forma transparente, usando los parámetros de seguridad establecidos.

   a) Estos datos son cifrados usando criptografía simétrica, haciendo uso de la clave acordada durante el handshake.

   b) Los datos de aplicación son comprimidos (de forma opcional) y fragmentados para optimizar el uso de la red.

3. Cambios de especificación y alertas

   Los nodos pueden cambiar los parámetros de seguridad en cualquier momento. Además, pueden intercambiar avisos o incluso cerrar la sesión si se detecta un problema serio de seguridad.

TLS se ha convertido en una referencia en cuanto a protocolos de seguridad, llegando a inspirar a otros protocolos, como *Datagram Transport Layer Security* (DTLS) [22, 23]. DTLS tiene la ventaja de que puede operar a pesar de que se pierdan algunos paquetes en la conexión, por lo que no requiere un canal de comunicación confiable. Esta característica lo hace ideal para proporcionar seguridad a las aplicaciones que hacen uso del protocolo UDP.

En el capítulo 8 hablaremos de distintas implementaciones de TLS. Cabe destacar aquí que una buena implementación del estándar debe cumplir con todos los requisitos definidos en el mismo. Sin embargo, no todas las implementaciones de TLS abarcan el amplio abanico de parámetros de seguridad disponibles, tales como funciones de cifrado simétrico, algoritmos de clave pública, funciones de hash, etc. Por este motivo, hablaremos de los conjuntos de funciones criptográficas, o *ciphersuites*, para referirnos al conjunto de parámetros de seguridad disponibles en una implementación determinada[3].

[3] Algunas implementaciones de TLS disponen de un gran conjunto de funciones criptográficas, pero permiten elegir un subconjunto de las mismas en tiempo de compilación. Esta forma de elegir el *ciphersuite* en tiempo de compilación será muy conveniente a la hora de integrar una implementación de TLS en un sistema empotrado, pues reduce el uso de recursos computacionales sin comprometer sin comprometer la versatilidad del protocolo.

Capítulo 5

Certificados digitales X.509

La criptografía de clave pública hace posible que las personas que no comparten una clave común de antemano puedan comunicarse de forma segura. También hace que la firma de mensajes sea posible sin la presencia de un tercero confiable. Finalmente, la firma de los *message digests* hace posible que el destinatario verifique la integridad de los mensajes recibidos de forma fácil y segura. Sin embargo, hay un problema que no hemos resuelto aún. Se trata de la verificación de la identidad y de la obtención de la clave pública asociada a una entidad. Si fuéramos a intercambiar una clave privada con cualquier persona o entidad, lo primero que necesitaremos es su clave pública. Sin embargo, si esta clave se hace disponible en una web sin más, corremos el riesgo de que un posible atacante intercepte la comunicación con la web y nos haga creer que la clave es otra (la del propio atacante, por ejemplo). De esta forma, el atacante podría interceptar todo el intercambio de claves, quebrando la confidencialidad (y posiblemente la integridad) de la comunicación. Claramente, necesitamos algún mecanismo para garantizar que las claves públicas puedan intercambiarse de forma segura [8].

Los certificados digitales de clave pública se fundamentan en la confianza depositada en una organización, la autoridad certificadora (*certification authority*, CA), de la que conocemos su identidad y clave pública. La CA certificará las claves públicas que pertenecen a personas, empresas y otras organizaciones, de forma que podamos verificar la correspondencia entre entidad y clave pública. El papel de la CA será verificar la identidad de una persona a través de un documento físico oficial (como el pasaporte) y emitir un documento que contendrá los datos del solicitante (principalmente su nombre), su clave pública, y un hash SHA firmado con la clave privada de la CA. Este documento es denominado certificado digital, y como vemos su trabajo fundamental es vincular una clave pública con el nombre de un sujeto (individuo, empresa, etc.).

Los certificados en sí no son secretos ni están protegidos, pudiendo ser publicados sin riesgo alguno. Si un atacante tratara de modificar un certificado durante su transmisión, nos daríamos cuenta de que la firma no se corresponde con el hash de los campos del certificado. De igual forma, si el atacante introdujera un certificado válido, podríamos comprobar que este no contiene el nombre de la entidad que intentamos autenticar. Una ventaja de este esquema es que la CA no tiene por qué estar en línea permanentemente. En cambio, su

función se limita a proporcionar la firma de los certificados digitales, lo cuál se lleva a cabo con cada solicitud, tras la verificación de la identidad física de la persona o empresa en cuestión.

5.1. Contenido de los certificados

El estándar X.509 aprobado por la ITU (*International Telecommunication Union*) define el formato en el que deben presentarse los certificados digitales. Este estándar, que es usado ampliamente en Internet, ha pasado por tres versiones desde la estandarización inicial en 1988. Discutiremos la versión más reciente del mismo (X.509 V3), tal y como está definido por la IETF [24, 25]. Los campos principales en un certificado se enumeran en la figura 5.1. A continuación se describe la función general de cada campo [8].

Version versión de X.509 con la que se ha generado el certificado.

Serial number este número, junto con el nombre de la CA, identifica el certificado de forma unívoca.

Signature algorithm el algoritmo usado para firmar el certificado.

Issuer nombre de la CA (en formado X.500).

Validity period fecha y hora del comienzo y fin de la validez del certificado.

Subject name la entidad cuya clave está siendo certificada (en formado X.500).

Public key la clave pública del sujeto y la ID del algoritmo que hace uso de la misma.

Issuer ID un ID opcional que identifica de forma unívoca al *issuer* del certificado.

Subject ID un ID opcional que identifica de forma unívoca al sujeto del certificado.

Extensions extensiones opcionales (existe una gran variedad de ellas).

Signature firma del certificado (firmado con la clave privada de la CA).

5.2. Cadena de confianza

Tener una única CA para emitir todos los certificados del mundo obviamente no funcionaría. Se colapsaría bajo la carga y también sería un punto crítico central. Una posible solución podría ser tener múltiples CA, todas gestionadas por la misma organización y todas usando la misma clave privada para firmar certificados. Si bien esto resolvería los problemas de carga y error, introduce un nuevo problema: la pérdida de la clave. Si hubiera docenas de servidores repartidos por todo el mundo, todos con la clave privada de la CA, la probabilidad de que la clave privada sea robada o se escape sería mucho mayor. Dado que el compromiso de esta clave arruinaría la infraestructura de ciberseguridad mundial, tener

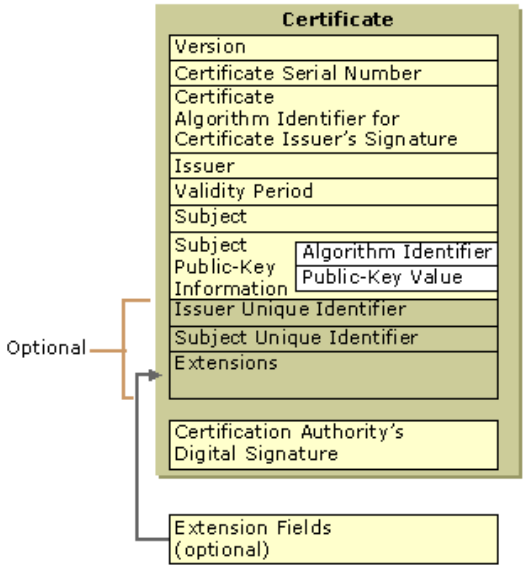


Figura 5.1: Campos principales de los certificados X.509 V3 [26].

una CA central única es muy arriesgado. Además, es difícil imaginar una autoridad que sea aceptada en todo el mundo como legítima y confiable. En algunos países, insistirían en que esta autoridad fuera un gobierno, mientras que en otros países insistirían en que no sea un gobierno.

Por estos motivos, se ha desarrollado una forma diferente de certificar claves públicas, que se conoce con el nombre general de infraestructura de clave pública (*public key infrastructure*, PKI). Una PKI tiene múltiples componentes, en los que se incluyen usuarios, CA, certificados y directorios. Lo que la PKI hace es proporcionar una forma de estructurar estos componentes y definir estándares para los diversos documentos y protocolos. Una forma particularmente simple de PKI es una jerarquía de CA, como la representada en la figura 5.2. En este ejemplo, se muestran tres niveles, pero en la práctica puede haber menos o más. La CA de nivel superior, denominada raíz (*root CA*), certifica las CA de segundo nivel, que aquí llamamos autoridades regionales (*regional authorities*, RA) porque podrían cubrir alguna región geográfica. Estos a su vez certifican las CA reales, que emiten los certificados X.509 a organizaciones y particulares. Cuando la CA raíz autoriza una nueva RA, genera un certificado X.509, en el cual la CA declara que ha aprobado la RA, incluye la nueva clave pública de la RA, la firma y se la entrega a la RA. De forma similar, cuando un RA aprueba una nueva CA, produce y firma un certificado que indica su aprobación y que contiene la clave pública de la CA. A continuación resumimos el funcionamiento de la PKI del ejemplo.

Cuando queremos comunicarnos de forma segura con una organización concreta O, es posible que esta tenga un certificado firmado por, digamos, CA 5. Pero como es muy probable que no tengamos información sobre CA 5, deberemos dirigirnos hacia esta autoridad y solicitar que nos demuestre su legitimidad. La CA 5 responderá con el certificado que recibió de la RA 2, que contiene la clave pública de CA 5. Ahora podremos verificar que

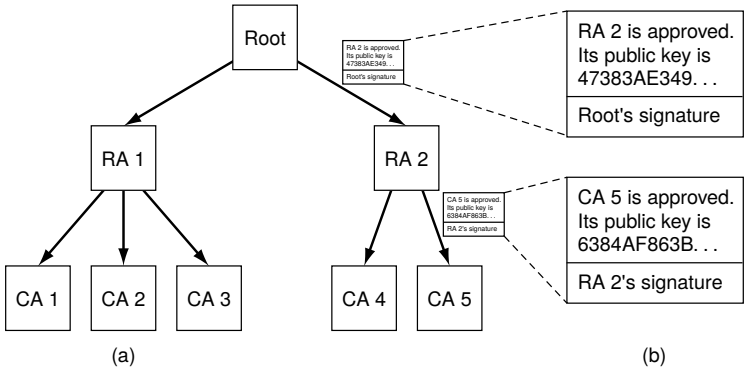


Figura 5.2: (a) Ejemplo de infraestructura de clave pública jerárquica. (b) Cadena de certificados en esta PKI [8].

el certificado de la organización O gracias a la clave pública de CA 5. Sin embargo, si no tenemos información previa sobre CA 5, nos será difícil confiar en la legitimidad de O, por lo que el siguiente paso será solicitar a la RA 2 que demuestre que CA 5 es legítimo. La respuesta esta consulta es un certificado firmado por la raíz y que contiene la clave pública de RA 2. Ahora podremos estar seguros de que O tiene un certificado legítimo, por lo que podremos confiar en que la comunicación se llevará a cabo con la organización en cuestión y no con un atacante.

En este ejemplo solo queda una pregunta por responder, y es de dónde obtenemos la clave pública de la CA raíz. Esta es una pregunta difícil de responder, ya que se supone que todos conocen la clave pública de la CA raíz. La forma habitual, cuando se trata de páginas web, es que el navegador incorpore la clave pública de la CA raíz por defecto.

Una forma que tienen las organizaciones de agilizar el proceso de verificación es aportar todos (o alguno) de los certificados necesarios para la misma junto al certificado propio. De esta forma podremos verificar el certificado de la organización sin tener que establecer una nueva conexión con cada una de las CA intermediarias, ya que dispondremos de todos los certificados de la cadena. Además, como todos los certificados están firmados, será posible detectar fácilmente cualquier intento de manipular sus contenidos.

Una cadena de certificados que retrocede hasta la CA raíz, como la del ejemplo, se denomina cadena de confianza (*trust chain*) o ruta de certificación (*certification path*). Esta técnica es ampliamente utilizada en la práctica. Por supuesto, todavía tenemos el problema de quién va a gestionar la CA raíz. La solución no es tener una CA raíz única, sino tener muchas CA raíces, cada una con sus propias RA y CA. De hecho, los navegadores modernos vienen preconfigurados con las claves públicas para más de 100 raíces. De esta forma, se puede evitar tener una única autoridad de confianza en todo el mundo. Además, la mayoría de los navegadores permiten a los usuarios inspeccionar las claves raíz, pudiendo eliminar las que parecen oscuras o añadiendo claves nuevas.

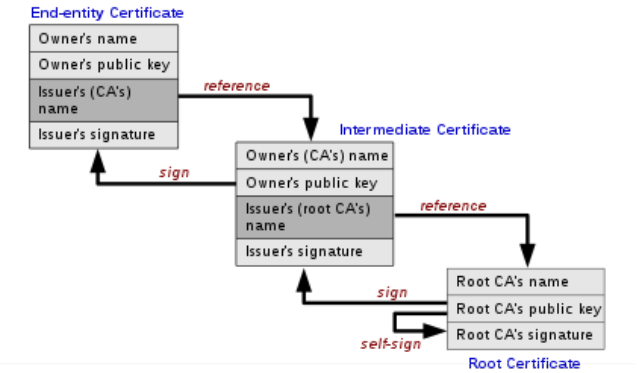


Figura 5.3: Cadena de confianza con certificados X.509 [27].

5.3. Estructura y tipos de fichero

Los certificados digitales X.509 usan la codificación denominada *Abstract Syntax Notation* 1 (ASN.1), definida por OSI. Este tipo de codificación es una especie de struct de C, salvo porque tiene una notación mucho más verbosa y peculiar. Haciendo uso de esta notación los campos de los certificados se encapsulan en forma de ficheros. Existen dos extensiones comúnmente usadas para estos ficheros, dependiendo de si este se encuentra en formato binario (DER) o con una codificación ASCII adicional (PEM). Esta codificación adicional usa un conjunto de 64 caracteres ASCII con los que se codifica el contenido del binario usando palabras de 6 bits [28]. La idea detrás de este tipo de codificación es maximizar la compatibilidad en el almacenamiento y la transmisión.

Capítulo 6

Amazon Web Services IoT

En este capítulo describiremos la arquitectura de la plataforma de Amazon para Internet de las Cosas (AWS IoT), explorando su potencial y las particularidades que presenta.

Como se describe en la documentación oficial, AWS IoT proporciona una *comunicación bidireccional segura* entre dispositivos conectados a Internet (ya sean sensores, actuadores, microcontroladores, electrodomésticos inteligentes, etc.) y la nube de Amazon (*AWS Cloud*) [29]. Esto nos permite recoger, almacenar y analizar datos de una gran variedad de dispositivos. Del mismo modo, nos permite crear aplicaciones para controlar estos dispositivos a través de un *smartphone* o *tablet*.

En este sentido, AWS IoT actúa como una pasarela a la cuál pueden conectarse tanto dispositivos como aplicaciones de terceros. Además, ofrece una interfaz sencilla para procesar los datos recogidos a través de los demás servicios de AWS. Este modo de operación hace que Amazon haya concentrado sus esfuerzos en una plataforma robusta y segura, dando soporte a empresas que desean construir aplicaciones complejas y escalables.

6.1. Componentes de AWS IoT

La plataforma AWS IoT está formada por diversos módulos que trabajan orquestados para proporcionar la funcionalidad completa ofrecida por este servicio. Con el fin de hacer un uso eficaz de la plataforma, conviene tener una imagen global de su arquitectura.

La figura 6.1 muestra los componentes principales de AWS IoT. Los elementos que quedan dentro del recuadro amarillo constituyen la plataforma propiamente dicha. Fuera del recuadro quedan los *dispositivos*, las *aplicaciones* y los *servicios*, tanto los de AWS como los de terceras partes. A continuación describimos los principales componentes de AWS IoT [29].

Pasarela para dispositivos (*Device gateway*) se encarga de establecer la comunicación entre la nube de Amazon y cada dispositivo. El *Device gateway* es el responsable de que esta comunicación se lleve a cabo de forma eficiente y segura.

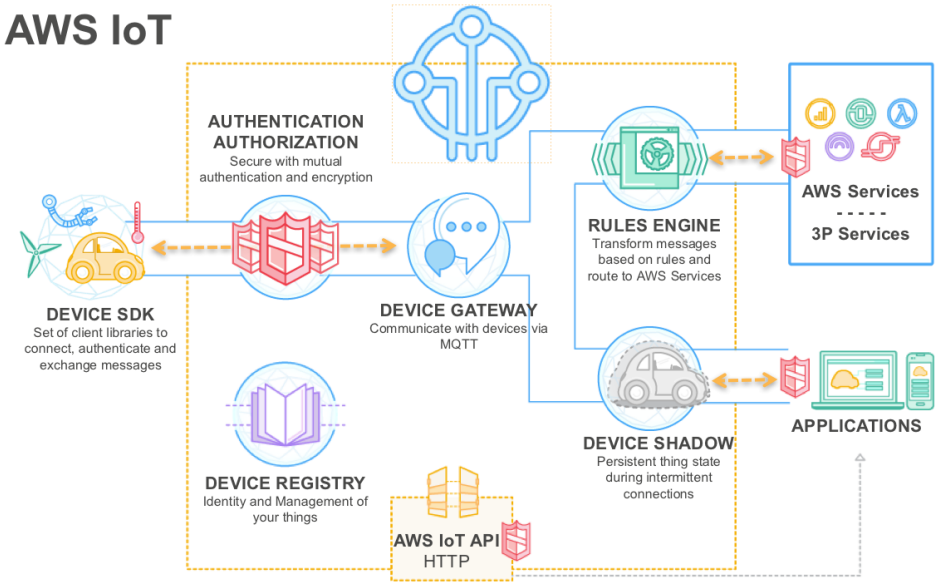


Figura 6.1: Principales componentes de AWS IoT [30].

Agente de mensajes (*Message broker*) proporciona un mecanismo seguro para que los dispositivos y las aplicaciones de AWS IoT publiquen y reciban mensajes entre sí, basándose en el protocolo MQTT [31] o en MQTT sobre WebSocket.

Motor de reglas (*Rules engine*) proporciona funciones de procesamiento de mensajes y de integración con otros servicios de la nube de Amazon. También permite usar el *Message broker* para volver a publicar mensajes para otros suscriptores.

Registro (*Registry*) organiza los recursos asociados a cada dispositivo en la nube de Amazon. Es necesario registrar los dispositivos y asociar hasta tres atributos personalizados a cada uno. Este componente también es responsable de asociar los certificados digitales y los IDs de clientes MQTT a cada dispositivo.

Registro de grupos (*Group registry*) permite administrar varios dispositivos a la vez clasificándolos en grupos.

Sombra del dispositivo (*Device shadow*) documento JSON utilizado para representar el estado de un dispositivo. También es usado para solicitar una acción al dispositivo.

AWS IoT dispone de otros componentes y servicios con los que el desarrollador puede interactuar para realizar algunas tareas de interés, como actualizar el firmware de sus dispositivos de forma remota, por ejemplo. No obstante, su uso es más avanzado y quedan fuera de los objetivos de este proyecto.

Cabe mencionar que tanto el agente de mensajes como el motor de reglas utilizan las características de seguridad de AWS. De este modo, solo los dispositivos y aplicaciones convenientemente autorizados pueden establecer una comunicación con AWS IoT.

6.2. Descripción funcional

En esta sección describiremos el esquema general de funcionamiento de AWS IoT. Será aquí donde desarrollaremos la relación entre los distintos elementos definidos anteriormente, así como su interacción con el mundo exterior.

Por un lado, tendremos dispositivos físicos ejecutando el *kit de desarrollo de software del dispositivo* (*AWS IoT Device SDK*). Cada uno de estos dispositivos debe disponer de unas credenciales con las que se identificará y se autenticará ante la Nube. En general, estas credenciales vienen dadas por un certificado digital único para cada dispositivo. De forma adicional, el dispositivo debe incluir el certificado raíz de autoridad certificadora (*CA root certificate*), de modo que pueda autenticar a los servidores de Amazon.

Por otro lado, tendremos que definir estos dispositivos en la plataforma, usando el Servicio de registro (*Registry*) anteriormente descrito. Cada uno de estos dispositivos virtuales dispondrá de un *nombre único*, que servirá para identificarlo, y de unos *atributos estáticos* a modo de descripción. Estos atributos estáticos pueden ser compartidos por dispositivos del mismo *tipo*, por lo que AWS IoT proporciona esta unidad de organización a tal efecto. De forma adicional, el dispositivo puede tener uno o varios certificados asociados. En general, cada dispositivo virtual tendrá asociado el mismo certificado digital que instalemos en el dispositivo físico correspondiente, de forma que AWS IoT pueda autenticarlo. Los certificados digitales definidos en la consola pueden tener asociadas una o varias políticas de uso (*policies*). Gracias a estas políticas de uso, distintos dispositivos tendrán permisos para realizar diferentes acciones, como puede ser publicar o suscribirse en determinados *topics*, actualizar su sombra, etc.

Cabe destacar que los *Atributos estáticos* sirven para describir al dispositivo en sí, por lo que su función no debe confundirse con aquélla de la *Sombra del dispositivo*. Mientras que estos atributos se almacenan de forma estática en el Registro, la sombra es un documento dinámico que contiene los valores de los sensores y el estado de los actuadores. Ejemplos de atributos estáticos son el número de serie, la versión de firmware, una breve descripción del dispositivo, etc. El siguiente epígrafe está dedicado a la sombra del dispositivo.

6.2.1. Sombra del dispositivo

El estado del dispositivo se guarda en un fichero JSON especial denominado sombra del dispositivo (*device shadow*). La sombra del dispositivo actúa como un intermediario entre dispositivos y aplicaciones, permitiendo que ambas partes tengan acceso al estado del dispositivo, pudiendo actualizarlo en ambos sentidos. Para que el estado del dispositivo pueda ser modificado por las aplicaciones, cada parámetro definido en la sombra podrá estar bajo una de las categorías *reported* y *desired* [32].

Bajo la categoría *reported* tendremos un informe del estado confeccionado por el dispositivo. De este modo, podemos decir que los parámetros encontrados en esta categoría tienen un significado informativo del estado del dispositivo, y solo el dispositivo debería escribir en ellos. Es en esta sección donde encontraremos los valores de los sensores y el estado de los actuadores reportado por el dispositivo.

Algoritmo 6.1 Ejemplo de sombra de dispositivo [33].

```
{
    "state" : {
        "desired" : {
          "color" : "RED",
          "sequence" : [ "RED", "GREEN", "BLUE" ]
        },
        "reported" : {
          "color" : "GREEN"
        }
    },
    "metadata" : {
        "desired" : {
            "color" : {
                "timestamp" : 12345
            },
            "sequence" : {
                "timestamp" : 12345
            }
        },
        "reported" : {
            "color" : {
                "timestamp" : 12345
            }
        }
    },
    "version" : 10,
    "clientToken" : "UniqueClientToken",
    "timestamp": 123456789
}
```

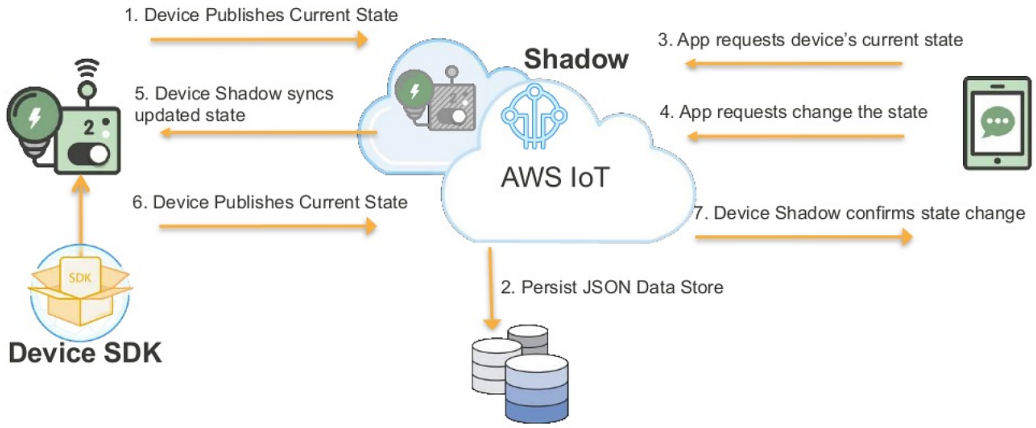


Figura 6.2: Flujo de mensajes en la interacción entre el dispositivo y una aplicación a través de la sombra .

En la categoría *desired*, en cambio, tendremos instrucciones para el dispositivo, como el cambio de color de un LED o el accionamiento de un motor. Como puede intuirse, esta categoría es confeccionada por la aplicación, debiendo ser revisada por el dispositivo para ejecutar las acciones solicitadas. Una vez completada la acción, el dispositivo actualizará el estado del actuador en el apartado *reported*.

La figura 6.2 muestra el flujo de mensajes intercambiados entre el dispositivo y la aplicación, usando la sombra de intermediario. Como puede comprobarse, el estado publicado por el dispositivo (1) se almacena en un fichero JSON, la sombra del dispositivo, en la nube de Amazon (2). Este fichero será proporcionado a la aplicación cuando ésta realice una petición del estado del dispositivo (3). Cada vez que el usuario solicite un cambio de estado a través de la aplicación (4), estos cambios se almacenarán en el fichero JSON, hasta que el dispositivo se conecte a la Nube y ejecute una actualización de su estado (5). Será entonces cuando el dispositivo recibirá la acción solicitada bajo el apartado *desired* de la sombra. Una vez realizada la acción, con éxito o no, el dispositivo reportará su nuevo estado en el apartado *reported* (6). La aplicación será informada de inmediato, siempre y cuando esté en línea (7). El Algoritmo 6.1 muestra un ejemplo sencillo de sombra de dispositivo, en el que la aplicación requiere un cambio de color en el LED del dispositivo.

Una ventaja de la existencia de la sombra del dispositivo con respecto a otras implementaciones es que puede conocerse el estado del dispositivo aunque este se encuentre sin conexión. Evidentemente, en este caso no contaremos con el estado actualizado, pero podremos visualizar el último estado registrado. Además, la sombra cuenta con una serie de metadatos, en los que se incluye un sello temporal (*timestamp*) con la última actualización en cada uno de los sentidos de la comunicación. Del mismo modo, será posible enviar instrucciones a nuestro dispositivo independientemente de su conectividad.

6.2.2. Esquema de conexión

Como en cualquier esquema IoT, el dispositivo se comunicará con la plataforma a través de una conexión a Internet. En el caso de AWS IoT, la comunicación se lleva a cabo a través del protocolo MQTT, sobre un canal seguro TLS. Esta conexión se produce en varios pasos.

En primer lugar, el dispositivo, que tendrá el rol de cliente, inicia una comunicación segura TLS con el servidor de AWS IoT. En esta primera comunicación servidor y cliente intercambian y verifican el certificado digital de la otra parte. De esta forma, la plataforma identifica y autentica el dispositivo, y el dispositivo se asegura de la autenticidad del servidor. En este proceso se establece una clave con la que se encriptarán todos los mensajes.

Una vez establecida la comunicación segura, el dispositivo puede publicar mensajes siguiendo el protocolo MQTT. Cuando el dispositivo trata de publicar un mensaje en un *topic*, la plataforma verifica si este tiene permiso para publicar en dicho *topic*. En caso afirmativo, el *message broker* retransmite este mensaje a cualquier elemento de la red suscrito a este *topic*. Del mismo modo, el dispositivo puede suscribirse a cualquier *topic* siempre y cuando tenga permisos para ello.

Actualización de la sombra

La actualización de la sombra se lleva a cabo publicando mensajes en unos *topics* especiales, destinados a este fin. En concreto, están definidos los siguientes *topics*, donde el dispositivo queda identificado por la variable `thingName` [33].

`$aws/things/thingName/shadow/update` Tanto las aplicaciones como los dispositivos pueden publicar mensajes a este *topic* para actualizar la sombra.

`$aws/things/thingName/update/accepted` AWS IoT publica un mensaje en este *topic* para confirmar la actualización de la sombra.

`$aws/things/thingName/update/rejected` AWS IoT publica un mensaje en este *topic* para indicar que la actualización de la sombra no tuvo éxito.

`$aws/things/thingName/get` Tanto las aplicaciones como los dispositivos pueden publicar un mensaje vacío a este *topic* para obtener la sombra.

`$aws/things/thingName/get/accepted` AWS IoT publica un mensaje en este *topic* para con la sombra actual.

`$aws/things/thingName/get/rejected` AWS IoT publica un mensaje de error en este *topic* para indicar que no se pudo recuperar la sombra.

De forma adicional, existen otros *topics* especiales relacionados con la sombra, cuyo uso es más avanzado.

RESTful API

AWS IoT dispone de una API RESTful para permitir que las aplicaciones puedan conectarse e interactuar con la sombra a través del protocolo HTTPS. En este caso, la seguridad es similar a la llevada a cabo en el caso del dispositivo. La diferencia es que la aplicación puede tener una credencial basada en una tupla (ID, clave) en vez de un certificado [34].

6.3. Disparadores y reglas

Aunque no se han usado en la consecución de este proyecto, destacamos que AWS IoT permite definir reglas con las que pueden realizarse diversas acciones al detectar eventos determinados. De este modo, es posible redireccionar mensajes entre distintos *topics* MQTT, detectar la superación de un umbral en un determinado parámetro, detectar la aparición de un mensaje concreto en un *topic*, etc. Las acciones que pueden asociarse a un evento son muy variopintas, pero siempre implican una interacción con otros servicios de la nube de Amazon. Así, por ejemplo, es posible insertar un nuevo registro en una base de datos *AWS DynamoDB*, invocar una función *AWS Lambda* o enviar mensajes específicos a *AWS Salesforce* .

6.4. Opciones de facturación

Amazon Web Services ofrece un amplio abanico de servicios de computación en la nube. Para cada servicio, el usuario debe abonar el precio correspondiente a los recursos consumidos. Aunque existen varias modalidades de facturación, todas ellas requieren que el usuario proporcione un método de pago. Sin embargo, existen dos alternativas gratuitas.

AWS Free Tier

Amazon establece un período de prueba gratuito para atraer nuevos clientes y dar a conocer su tecnología. En este período, es posible utilizar la mayoría de los servicios ofrecidos por AWS sin incurrir en gastos. No obstante, es necesario configurar un método de pago para comenzar a usar los servicios. Todos los detalles sobre *AWS Free Tier* están disponibles en `https://aws.amazon.com/free/` [35].

AWS Educate

AWS Educate proporciona a profesores y alumnos acceso a muchos de los servicios de AWS. Además, incluye ventajas como programas de capacitación y rutas profesionales en la nube de Amazon. Esta modalidad, que está disponible en 47 países, requiere que el usuario esté vinculado a alguna institución académica y procese una solicitud que será tratada de forma personalizada por Amazon[1]. Los detalles de esta modalidad están disponibles en

[1] Deberá usarse el correo institucional en la solicitud.

https://aws.amazon.com/education/ [36].

Parte II

Trabajo realizado

Capítulo 7

Presentación del sistema

Una vez revisados los fundamentos teóricos y las tecnologías usadas en este proyecto, pasaremos a describir el trabajo realizado durante la consecución del mismo. En los próximos cuatro capítulos presentaremos el sistema con el que trabajaremos y los requisitos que deberá satisfacer, analizaremos las diferentes implementaciones de TLS disponibles, y finalmente repasaremos el proceso de integración de la implementación escogida, así como el proceso de integración del AWS IoT Device SDK.

Este capítulo está dedicado a la descripción detallada del sistema, reparando en el funcionamiento general del mismo y en cada uno de los elementos que lo forman. En primer lugar, presentaremos la estructura general del sistema, profundizando en la función llevada a cabo por cada uno de los elementos que lo forman. Pasaremos entonces a describir estos elementos en mayor profundidad, a fin de comprender mejor los detalles de la posterior integración.

Comenzaremos por introducir DAVE™, el entorno de desarrollo integrado (*integrated development environment*, IDE) proporcionado por Infineon para trabajar con sus microcontroladores XMC™ [37]. Será este IDE el que usaremos para editar el código fuente, compilar el proyecto y cargar el firmware en el dispositivo.

7.1. El entorno de trabajo DAVE™

Trabajar con microcontroladores supone editar código fuente, compilarlo a la plataforma de destino, cargar el firmware en el dispositivo y depurar el programa en tiempo de ejecución. Esto puede hacerse de diferentes formas, usando distintos flujos de trabajo. Una de las más populares es usar un entorno de desarrollo integrado que reúna las herramientas necesarias para llevar a cabo estas tareas. En la realización de este trabajo, hemos usado DAVE™ en su versión 4.3.2.

DAVE™ es un IDE profesional y gratuito con el cuál podemos llevar a cabo todo el proceso de desarrollo, desde la evaluación hasta la producción (*evaluation to production*, E2P) [37]. DAVE™ está basado en el conocido entorno de desarrollo de código abierto Eclipse IDE, e incluye el compilador de C de GNU (*GNU C compiler*, GCC), interfaz de

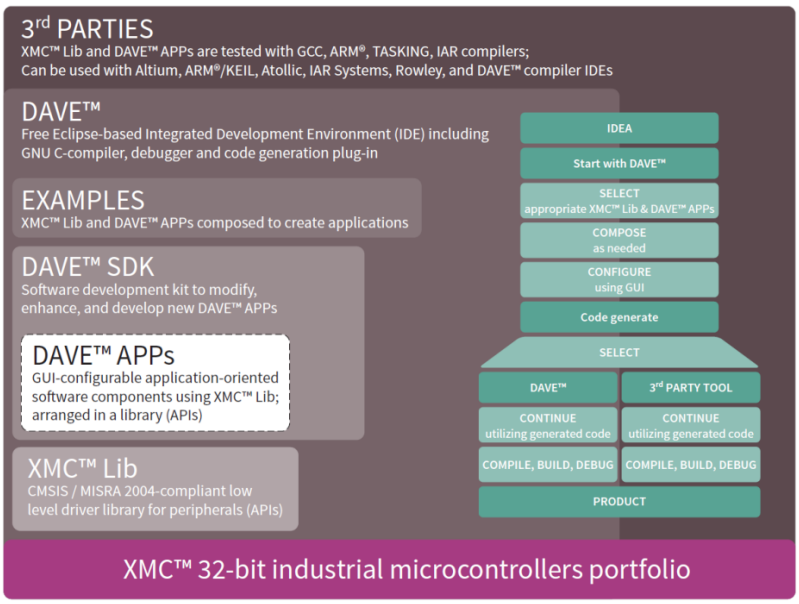


Figura 7.1: Componentes de DAVE™ y flujo de trabajo. [37]

depuración, repositorio de código completo, plug-in de generación de código y administración de recursos de hardware.

Como peculiaridad, resaltaremos que esta IDE incluye las denominadas DAVE™ APPs, que son componentes software orientados a aplicación que pueden configurarse a través de una interfaz gráfica de usuario (*graphical user interface*, GUI). Gracias a las mismas, es posible usar muchos de los módulos del microcontrolador de forma sencilla y eficiente. Al añadir una de estas APPs al proyecto, DAVE™ importa el código necesario para usar la biblioteca o módulo hardware proporcionados por la misma, colocando los ficheros de la biblioteca en un directorio reservado a tal efecto. Asímismo, las APPs proporcionan una API homogénea para acceder a los distintos recursos del microcontrolador, y están ampliamente documentadas.

La figura 7.1 muestra los distintos componentes de DAVE™, situando las DAVE™ APPs en el entorno del IDE, además del flujo de trabajo E2P que puede obtenerse con el mismo. Como puede comprobarse en la misma, las DAVE™ APPs se apoyan en la XMC™ Lib, que proporciona acceso a los distintos elementos del microcontrolador en un nivel más cercano a la máquina.

DAVE™ se encarga de gestionar el proceso de compilación, usando GCC con los parámetros definidos en la configuración del proyecto. Como Eclipse IDE, permite excluir algunos directorios o ficheros del proceso de compilación. Al final de este proceso, se genera un archivo binario con la extensión `.bin` cuyo contenido se ajusta a las regiones de memoria definidas en el *linker script*. Para concluir el proceso, DAVE™ permite cargar el binario en los microcontroladores XMC™ conectándose al mismo por USB, a través de un programador JTAG. El firmware cargado puede depurarse haciendo uso de la interfaz gráfica de depuración de DAVE™, la cuál se muestra por defecto después de cada programación.

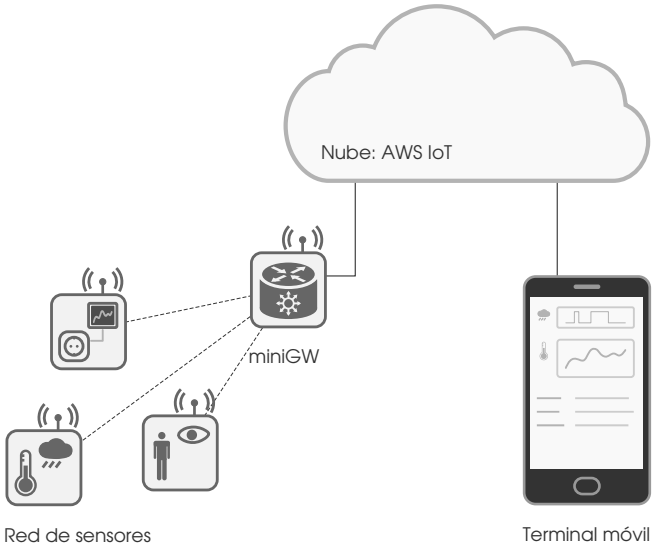


Figura 7.2: Arquitectura general del sistema. Las líneas discontinuas representan la conexión inalámbrica a través del protocolo propietario de eesy-innovation. Las líneas continuas representan conexión IP.

7.2. Objetivo perseguido y arquitectura propuesta

Como hemos introducido en el capítulo 1, el sistema con el que trabajaremos es una pasarela (miniGW) que permitirá la interconexión de una red de sensores, instalada en un *SmartHome*, con la nube de Amazon ASW IoT. En su versión actual, el miniGW es capaz de comunicarse con los sensores, almacenar los datos recibidos en memoria, mostrar los datos a través de una interfaz web sencilla, y actualizar la hora del sistema a través del protocolo NTP. Todos estos procesos están implementados como tareas en un sistema operativo de tiempo real denominado CMSIS-RTOS. El primer objetivo de este trabajo será integrar una biblioteca TLS con un *ciphersuite* que permita la conexión con AWS IoT. Seguidamente, tendremos que integrar el AWS IoT Device SDK, que proporcionará los mecanismos necesarios para establecer una conexión segura con la Nube y permitir la comunicación bidireccional. Esta comunicación, que en este caso se llevará a cabo a través del protocolo MQTT, permitirá enviar los datos de los sensores y recibir comandos para los actuadores.

A continuación presentaremos los distintos elementos que componen la arquitectura básica del sistema, la cuál se ilustra en la figura 7.2. Describiremos estos elementos con mayor profundidad en las siguientes secciones.

miniGW pasarela IoT, cuya función es comunicar la red de sensores con la nube de forma confiable, segura y eficiente. También está encargada de almacenar algunos elementos de configuración.

Red de sensores es el conjunto de sensores y actuadores que proporcionan información sobre variables de interés en el hogar y permiten controlarlo. Por simplicidad, denominaremos a este conjunto de dispositivos simplimente por el término *red de sensores*.

Nube se trata de un servicio on-line que permite el almacenamiento y gestión de los datos producidos por la red de sensores. En el presente proyecto, usaremos el servicio AWS IoT para tal fin.

App móvil interfaz a través la cuál el usuario interacciona con su SmartHome. La App muestra los datos recogidos por los sensores y permite modificar el estado de los actuadores, manualmente o a través de reglas.

7.3. La pasarela *miniGW*

El dispositivo con el que trabajaremos será el miniGW introducido en la sección anterior. Como explicamos anteriormente, el miniGW es la pasarela entre la red de sensores y la nube de Amazon, por lo que organiza y traduce los mensajes de la red local, para transmitirlos a través de la pila de protocolos de Internet.

Como puede comprobarse en la figura 7.3, el miniGW no es más que un pequeño circuito impreso, cuyas dimensiones no exceden los 5x5x2 cm. Se trata de un dispositivo basado en microcontrolador, en el que un XMC4500 hace las veces de cerebro, encargándose de la lógica y de la coordinación de las distintas interfaces. Es en este circuito donde se encuentra el programa que gestiona las distintas tareas de las que el miniGW tiene que hacerse cargo, así como la pila de protocolos de Internet y los controladores de las distintas interfaces. Entre estas interfaces, cabe destacar el TDA5340, un transceptor de muy bajo consumo encargado de mantener la comunicación con los sensores.

Además de estos circuitos principales, que describiremos a continuación, el miniGW cuenta con las siguientes interfaces físicas. Estas interfaces, representadas en la figura 7.4, permiten que el miniGW se comunique con el mundo exterior:

Puerto micro USB proporciona alimentación al circuito. Además, puede usarse para varios propósitos, como establecer una comunicación serial o actualizar el firmware del XMC4500.

Conector RJ45 proporciona conexión a Internet a través de Ethernet.

Ranura micro SD proporciona acceso a una memoria no volátil, sirviendo de almacenamiento para los datos de sensores, la interfaz web, etc. También puede usarse para cargar un nuevo firmware del XMC4500.

Antena recoge las señales de radio enviadas por los sensores, a la vez que permite transmitir señales a los actuadores. Está conectada directamente al TDA5340.

Acceso a pines se trata de un acceso a la interfaz de programación y depuración del microcontrolador JTAG. Este conector solo se encuentra en la versión de desarrollo del miniGW, no soldándose en producción.

A continuación expondremos algunos detalles de interés sobre los dos módulos principales del sistema: el microcontrolador XMC4500 y el transceptor TDA5340. Posteriormente, introduciremos la estructura del firmware presente en el miniGW.

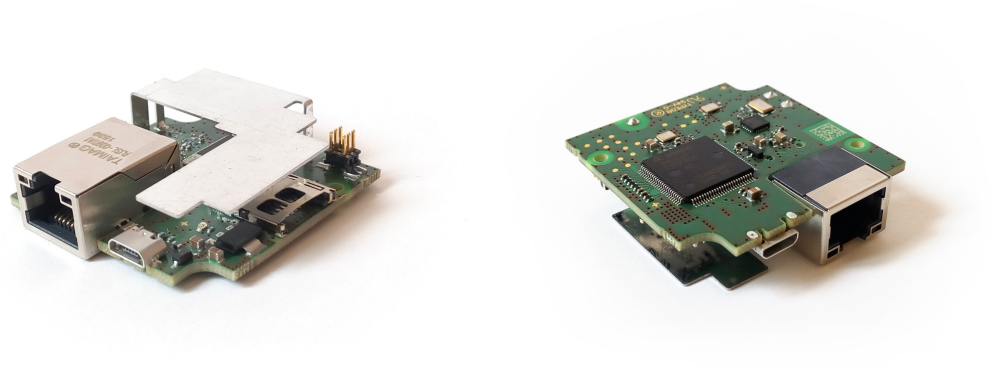


Figura 7.3: Fotografía del hardware del miniGW. Pueden observarse distintos componentes electrónicos e interfaces.

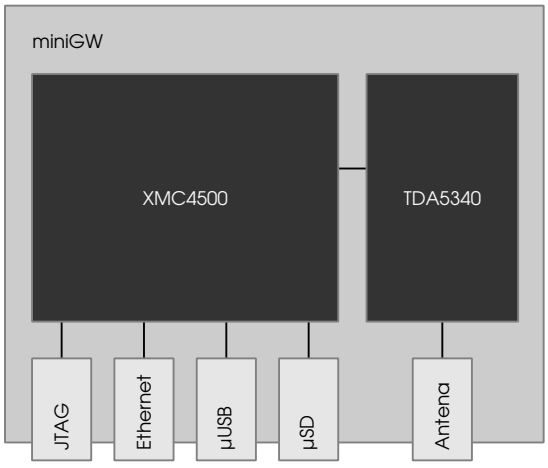


Figura 7.4: Arquitectura hardware del miniGW. Pueden distinguirse los módulos principales y las diferentes interfaces.

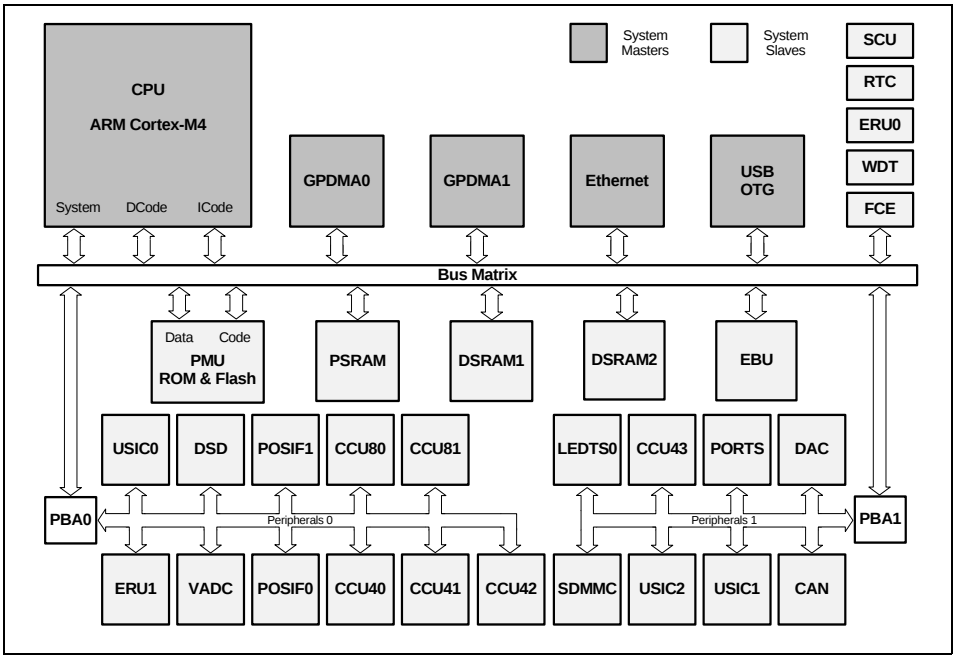


Figura 7.5: Diagrama de bloques del microcontrolador XMC4500 [38].

XMC4500

XMC4500 es un microcontrolador de la familia de microcontroladores de alto rendimiento XMC4000 de Infineon. Esta familia, basada en el procesador ARM Cortex-M4, está optimizada para tareas tales como el control industrial, conversión de potencia y sensorización y control.

La figura 7.5 muestra el diagrama de bloques de este circuito, que cuenta con numerosos módulos de aceleración hardware. A continuación presentamos una lista con las características principales de este microcontrolador [38]. Esta lista no es exhaustiva, nos centraremos exclusivamente en aquellas características que serán más relevantes en la consecución del presente proyecto.

- Núcleo de procesamiento
  - Núcleo ARM Cortex-M4. CPU de 32 bits de alto rendimiento, con instrucciones para procesamiento digital de señales (DSP).
  - Temporizador del sistema (SysTick) para dar soporte a sistemas operativos.
  - Unidades de acceso directo a memoria (DMA) y de protección de memoria (MPU).
- Bancos de memoria integrados
  - 16 KB *on-chip boot ROM.*
  - 64 KB *on-chip high-speed program memory.*

  - 64 KB *on-chip high speed data memory.*
  - 32 KB *on-chip high-speed communication.*
  - 1024 KB *on-chip Flash Memory* con *4 KB instruction cache.*

- Periféficos de comunicación
  - Módulo Ethernet MAC compatible con las tasas de transmisión 10/100 Mbit/s.
  - Universal Serial Bus, USB 2.0 *host*.
  - Canales de interfaz serial universal (USIC), configurables como interfaces UART, double-SPI, quad-SPI, IIC, IIS y LIN.
  - Interfaces SD y Multi-Media Card (SDMMC) para tarjetas de almacenamiento de datos.
  - Unidad de interfaz de bus externo (EBU), permitiendo la comunicación con memorias externas y otros periféricos *off-chip*.
- Periféricos analógicos
  - Conversores analógico-digital (VADC) con 12 bits de resolución.
  - Conversores digital-analógico (DAC) con 12 bits de resolución. Estos conversores tienen distintos modos de operación, con los que pueden generar distintas formas de onda, incluyendo ruido [39].
- Periféricos de control industrial
  - Perro guardián (WDT).
  - Reloj de tiempo real (RTC).
- Soporte para depuración del firmware
  - Interfaces ARM-JTAG y SWD.
  - Soporte para *breakpoints*.

TDA5340

El TDA5340 es un transceptor de bajo consumo diseñado y producido por Infineon para aplicaciones en la industria del automóvil. Este circuito integrado permite distintos tipos de modulación, como ASK, FSK y GFSK para las bandas de frecuencia 300-320, 415-495 y 863-960 MHz [40]. El tipo de modulación, la frecuencia y otros parámetros de transmisión son seleccionados a través de un pequeño firmware que se ejecuta en este circuito. Este firmware permite además llevar a cabo algunas operaciones básicas como la entrada en el modo de suspensión.

Este módulo será el encargado de comunicar el miniGW con la red de sensores, recibiendo los datos de los distintos sensores para proporcionárselos al XMC4500 a través de un bus SPI. De forma análoga, el TDA enviará comandos a los actuadores a petición del microcontrolador.

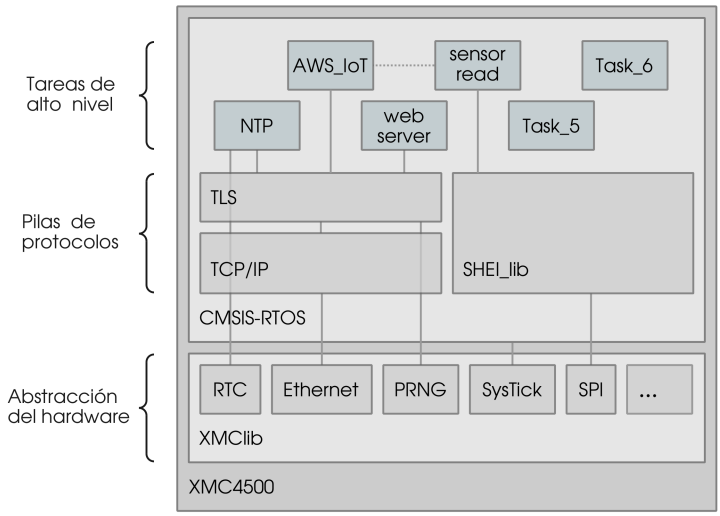


Figura 7.6: Arquitectura del firmware del miniGW. Las líneas continuas indican dependencias entre módulos, las líneas discontinuas indican comunicación entre tareas.

La comunicación con los sensores y actuadores de la red de sensores se lleva a cabo a través de un protocolo propietario, que usa cifrado por bloques AES en modo contador (AES-CTR) para garantizar la seguridad.

Firmware del XMC4500

La lógica del XMC4500 debe garantizar que el miniGW realiza todas las funciones para las que fue diseñado. En este firmware se incluyen diversas tareas que deberán ejecutarse en paralelo, como la recepción de paquetes de datos de los sensores o la continua actualización de la fecha y hora a través del protocolo NTP. La arquitectura del firmware puede consultarse en la figura 7.6. Cabe destacar que la finalidad de este proyecto no es añadir nuevas funcionalidades al firmware desarrollado, sino demostrar el uso de AWS IoT Device SDK para que eesy-innovation lleve a cabo la integración final. Como hemos mencionado ya, este SDK requerirá una capa de seguridad TLS, que deberá ser compatible con el resto de elementos del firmware. Por este motivo, necesitamos revisar cuáles son los elementos fundamentales que estarán incluidos en el sistema final, y que supondrán el marco en el cuál deberemos integrar la biblioteca TLS y el AWS IoT SDK. A continuación revisamos los elementos software principales.

CMSIS-RTOS Las distintas tareas que debe realizar el miniGW está construidas sobre un sistema operativo de tiempo real denominado CMSIS-RTOS. Este RTOS, diseñado especialmente para dispositivos Cortex-M, permite que las distintas tareas se ejecuten de forma concurrente y proporcionan herramientas para la gestión de recursos y de la comunicación entre tareas.

lightweight IP (lwIP) La pila de protocolos de Internet está proporcionada por lwIP. Esta biblioteca está diseñada para reducir el uso de recursos, conservando todas las funcionalidades de TCP/IP. Proporciona una interfaz de sockets tipo BSD estándar.

Tanto CMSIS-RTOS como lwIP están disponibles en el formato de DAVE™ APPs, por lo que han sido añadidas al proyecto usando esta herramienta.

7.4. Red de sensores

La red de sensores se encarga de monitorizar variables de interés para el usuario, transmitiéndolos al miniGW de forma inalámbrica. La gama de productos H2 de eesy-innovation incluye una multitud de sensores y actuadores, como sensores de presencia, estación meteorológica, enchufe inteligente (*SmartPlug*), etc. Todos estos nodos se comunican con el miniGW de forma segura a través de un protocolo propietario. Además, cada uno de estos sensores y actuadores tiene un número de identificación único (*unique identity*, UID) que lo identifica de forma única, no solo en la red local, sino en toda la Internet.

Una diferencia a destacar entre los sensores y los actuadores es que los primeros usan un transmisor (comunicación unidireccional) mientras que los segundos hacen uso de un transceptor (comunicación bidireccional). Esto fuerza que la topología de red sea en estrella, con el miniGW como nodo central.

7.5. La nube de Amazon *AWS IoT*

El servicio AWS IoT se encarga de almacenar los datos de los sensores, haciendo que permanezcan disponibles para que el usuario pueda consultarlos cuando desee. Esto se consigue gracias a la sombra del dispositivo, descrita en la sección 6.2.

La nube alojará dispositivos virtuales, uno por cada dispositivo físico (sensor o actuador), que tendrá que asociar al usuario. La definición de estos dispositivos en producción está fuera del alcance de este proyecto. En la sección 10.3 se definirá un dispositivo de prueba para la propia pasarela haciendo uso de la consola de AWS IoT, de forma que podamos verificar la comunicación entre el miniGW y la Nube.

7.6. App móvil

La App se conecta a la nube de Amazon a través de las APIs pertinentes, permitiendo al usuario visualizar los datos. Esta App queda fuera del alcance de este proyecto, por lo que usaremos la consola de AWS IoT para verificar el intercambio de mensajes.

Capítulo 8

Análisis de diferentes bibliotecas TLS

Como vimos en el capítulo 3, las dos características principales de un sistema empotrado son sus limitaciones de recursos y la ausencia de un sistema operativo real. Estas dos características derivan directamente de la definición de sistema empotrado (sistema que realiza una tarea específica), y condicionan por completo el desarrollo de su software.

En el tema que nos ocupa, las limitaciones en los recursos computacionales nos llevarán a buscar implementaciones del protocolo TLS ligeras, cuya configuración se realice, preferiblemente, en tiempo de compilación. También tendremos que asegurarnos de que las implementaciones analizadas puedan funcionar con los servicios mínimos proporcionados por un RTOS.

Dedicaremos el presente capítulo al análisis de diversas implementaciones TLS, llevado a cabo tras un ejercicio de búsqueda cuyo objetivo ha sido conocer las implementaciones TLS más destacadas en el ámbito de los sistemas empotrados. Realizaremos este análisis mediante la evaluación de una lista de características, que incluye, pero no se limita a: requisitos computacionales, funcionalidades, portabilidad y coste de la implementación. Elegiremos la biblioteca TLS atendiendo al análisis presentado a continuación, que también contará, de forma adicional, con algunos aspectos económicos y de licencia.

8.1. Criterios de búsqueda y evaluación

La búsqueda de bibliotecas TLS se ha llevado a cabo usando diversas herramientas de búsqueda, entre las que se inclyen el motor genérico de Google, artículos de Wikipedia [41], los repositorios de Github y el foro de desarrolladores Stack Exchange. Esta búsqueda, en cambio, ha prescindido de bases de datos académicas, ya que se solo hemos considerado implementaciones comerciales o que se encuentren en producción.

En nuestra búsqueda, solo hemos considerado aquellas implementaciones de TLS que se ajustaran a sistemas embebidos. Este criterio supone que las bibliotecas deben estar programadas en lenguaje C estándar, y que deben presentar el mínimo número de dependencias. Idealmente, la biblioteca seleccionada presentará ciertas facilidades para la integración con el sistema objetivo, como la compilación nativa en CMSIS-RTOS o compatibilidad con la

pila de lwIP.

Además de la compatibilidad con el sistema, será necesario que la implementación seleccionada cumpla con los siguientes criterios:

- Biblioteca en continuo desarrollo: se trata de un requisito de seguridad, ya que es frecuente encontrar vulnerabilidades en las implementaciones, y estas deben solventarse cuanto antes.

- Ciphersuite *compatible con AWS IoT. Se requiere TLS 1.2, con alguno de los ciphersuites listados en [42]. En especial, se recomiendan:*

  - ECDHE-ECDSA-AES128-GCM-SHA256
  - ECDHE-RSA-AES128-GCM-SHA256

Los criterios de búsqueda establecidos nos hacen descartar la conocida biblioteca de seguridad OpenSSL. A pesar de que esta biblioteca cuenta con una larga tradición y sigue siendo un marco de referencia para muchas de las aplicaciones que implementan TLS, OpenSSL no fue diseñada para sistemas embebidos, por lo que su tamaño excede las limitaciones de nuestro sistema. Además, presenta dependencias que serían difícilmente salvables, y sus desarrolladores no recomiendan compilar OpenSSL en un entorno para el que no está preparado [43].

Además de los requisitos mencionados anteriormente, se evaluarán positivamente las siguientes características:

- Disponibilidad de documentación: será indispensable una buena documentación para comprobar la disponibilidad de características y el uso de las distintas funciones definidas en la API.

- Soporte para ingenieros: será conveniente en el caso en que se necesite soporte durante la implementación.

- Disponibilidad del código fuente: se valorará positivamente la disponibilidad del código fuente previa adquisición de la licencia, de forma que podamos indagar en el mismo para evaluar la implementación. Además, tener el código fuente abierto a la comunidad disminuye el riesgo de que contenga errores o puertas traseras.

- Licencias compatibles con la actividad a realizar, con precios razonables.

8.2. Bibliotecas analizadas

En esta sección presentaremos brevemente cada una de las bibliotecas analizadas, revisando algunos datos generales o de especial relevancia.

mbedTLS

Esta implementación, lanzada en 2006 bajo la denominación de PolarSSL, da soporte a los protocolos y configuraciones de TLS más usados. Se centra en la claridad del código y modularidad y está ampliamente documentada. Se distribuye bajo la licencia de código abierto Apache 2.0, por lo que puede usarse de forma gratuita para propósitos comerciales, siempre que se respeten los términos de la licencia.

Además, mbedTLS es la biblioteca de seguridad del proyecto mbed de ARM. Este proyecto trata de poner en conjunto una serie de herramientas para el desarrollo ágil de sistemas empotrados, incluyendo IoT. Con mbedTLS, la compañía apuesta por un mundo de dispositivos interconectados de forma segura.

WolfSSL

Lanzada en 2006 bajo la denominación de CyaSSL, da soporte a los protocolos y configuraciones de TLS más usados. Es altamente modular y personalizable, gracias a los diversos Makefiles que incluye consigo. Se distribuye siguiendo un modelo de doble licencia, incluyendo una licencia *Open Source* y otra comercial, que tiene un precio de license USD $5000.

Esta implementación es conocida por su uso en MySQL, por lo que ha adquirido una presencia considerable. Por último, una de las versiones de la biblioteca criptográfica wolfCrypt dispone de un certificado de verificación FIPS 140-2 (Certificate #2425).

MatrixSSL

Lanzada en 2004, da soporte a los protocolos y configuraciones de TLS más usados. De acuerdo con la web oficial, su *footprint* puede ser reducido hasta ~50KB, incluyendo biblioteca de criptografía y certificados, o hasta ~10KB en su *Tiny version*, que solo soporta PSK. Se distribuye bajo la licencia de software libre GPLv2, aunque también existe una licencia comercial (precio no disponible en la web oficial).

cryptlib

Lanzada inicialmente en 1995, da soporte a los protocolos y configuraciones de TLS más usados. Tiene un modelo de doble licencia, incluyendo una licencia Open Source y otra comercial, que puede adquirirse por USD $5000.

SharkSSL

Esta implementación es una propuesta de Real Time Logic, una firma estadounidense dedicada a IoT. La compañía también distribuye soluciones para HTTPS, MQTT y SMQ, entre otras. De acuerdo con la web, SharkSSL puede compilarse en menos de 38KB ROM,

necesitando solo 13KB RAM en tiempo de ejecución. Se distribuye bajo una licencia privada, cuyo precio no está disponible en la web.

8.3. Análisis comparativo

La tabla 8.1 muestra algunos datos generales de las bibliotecas mencionadas anteriormente, en los que se incluye la licencia y la compatibilidad con los distintos estándares TLS.

Debemos destacar aquí que a pesar de las diferencias mostradas, todas las bibliotecas consideradas cumplen con las siguientes características.

- Lenguaje de programación C estándar (y ensamblador, en algunos casos).
- Soporte para TLS 1.2.
- Soporte para la Suite B definida por la agencia nacional de seguridad de los EEUU (National Security Agency, NSA), que incluye [44]:
  - Algoritmos de criptografía simétrica: Advanced Encryption Standard – AES-128 y AES-256, con los modos CTR o GCM.
  - Algoritmos de criptografía asimétrica: *Elliptic Curve Digital Signature Algorithm* – ECDSA (firmas digitales) y *Elliptic Curve Diffie-Hellman* – ECDH (intercambio de claves).
  - Algoritmos de hash: Secure Hash Algorithm 2 – SHA-256 y SHA-384 (*message digest*).
- *Footprint* reducida, necesaria para la integración en sistemas embebidos.
- Compatible con la arquitectura ARM.

Compatibilidad con el sistema objetivo

Como desarrollamos en el capítulo 7, el sistema objetivo está basado en un procesador de la familia ARM Cortex-M4, en el que se ejecuta CMSIS-RTOS como sistema operativo de tiempo real y se usa lwIP para implementar la pila de protocolos TCP/IP y UDP/IP.

Estos aspectos deben considerarse cuidadosamente a la hora de elegir la biblioteca TLS a integrar. En esta sección trataremos los aspectos de compatibilidad con el sistema objetivo, así como la disponibilidad de documentación y de otros recursos.

Además de los criterios mencionados al comienzo de este capítulo, evaluaremos el soporte para CMSIS-RTOS y para lwIP que ofrecen estas bibliotecas, ya que pueden reducir dramáticamente el tiempo de desarrollo.

La tabla 8.3 muestra un resumen de la información obtenida a partir de fuentes oficiales de cada una de las implementaciones al respecto.

Implementación	WolfSSL	mbedTLS	MatrixSSL	cryptlyb	SharkSSL
Desarrollador	WolfSSL Inc.	Arm Limited	PeerSec Networks	Peter Gutmann	Real Time Logic
License	GPLv2 / comercial	Apache Licences 2.0 ‖ GPLv2 / comercial	GPLv2 / comercial	Licencia Sleepycat / comercial	Privativa (distribución comercial)
Precio para uso comercial	USD $5000	Sin costo (con condiciones)	Información no disponible en la web	USD $5000	Información no disponible en la web
SSL 2.0 (no seguro)	No	No	No	No	No especificado
SSL 3.0 (no seguro)	Desactivado por defecto	Desactivado por defecto	Desactivado por defecto	Desactivado por defecto	No especificado
TLS 1.0	Sí	Sí	Sí	Sí	Sí
TLS 1.1	Sí	Sí	Sí	Sí	Sí
TLS 1.2	Sí	Sí	Sí	Sí	Sí
DTLS 1.0	Sí	Sí	Sí	No	No
DTLS 1.2	Sí	Sí	Sí	No	No
NSA Suite B	Sí	Sí	Sí	Sí	Sí
Verificaciones FIPS-140 de nivel 1	wolfCrypt FIPS Module: 3.6.0 (#2425)	No	SafeZone FIPS Cryptographic Mudule: 11 (#2389)	No	No
Conocido por	Usado en MySQL	Parte del proyecto mbed de ARM	No aplica	No aplica	No aplica

Cuadro 8.1: Comparación de las bibliotecas analizadas: datos generales, licencia y compatibilidad con los distintos estándares TLS.

Implementación	WolfSSL	mbedTLS	MatrixSSL	cryptlyb	SharkSSL
Disponibilidad de documentación	Disponible de forma abierta en la web oficial. Exhaustiva y bien organizada. Numerosos recursos en línea	Disponible de forma abierta en la web oficial. Exhaustiva y bien organizada. Numerosos recursos en línea	Disponible de forma abierta en forma de PDF, descargable desde el repositorio oficial. Recursos en línea limitados	Disponible de forma abierta en forma de manual PDF. Recursos en línea limitados	No disponible de forma abierta (es necesario registrarse para probar ejemplos). Sin recursos en línea
Comunidad	Muy activa	Muy activa	Poco activa	Poco activa	Inexistente
Soporte para ingenieros	Foro para Q&A. Soporte por e-mail. GitHub Issues	Foro para Q&A. Soporte por e-mail. GitHub Issues	Soporte por e-mail. GitHub Issues	Soporte por e-mail.	Soporte por e-mail.
Disponibilidad del código fuente	Disponible en GitHub. Disponible el código de la última versión estable en la web oficial	Disponible en GitHub. Disponibles varias versiones estables en la web oficial	Disponible en GitHub. Disponibles 46+ versiones estables en la web oficial	Bajo demanda, tras rellenar un formulario	No disponible antes de la adquisición de la licencia
Presencia en las redes sociales	Web clara, GitHub, Twitter, Facebook, Linkedin	Web clara, GitHub, Twitter, Facebook, Linkedin	Web plana, GitHub, blog	Web poco clara, blog	Web clara
Soporte para CMSIS-RTOS	Sí, característica incluida	Sí, característica incluida	No se mencina en la web	Sí, característica incluida.	Con soporte para MDK-ARM y mbed
Soporte para lwIP	Sí, característica incluida	Sí, característica incluida	No se mencina en la web	No se mencina en la web ni en la documentación	Sí

Cuadro 8.3: Comparación de las bibliotecas analizadas: disponibilidad de documentación y soporte y compatibilidad con el sistema objetivo.

Implementación	WolfSSL	mbedTLS
Compilación en MS Windows	Sí, con el proyecto de Visual Studio incluido	Sí, con el proyecto de Visual Studio incluido y con la IDE Eclipse
Compilación en GNU/Linux	Sí, usando Make	Sí, usando Make y CMake
Configuración de la biblioteca	Sí, en la forma de un fichero de configuración que modifica el comportamiento de Make	Sí, modificando el fichero config.h
Soporte para otros sistemas operativos de tiempo real (RTOS)	Sí, incluyendo Linux embebido, FreeRTOS, TinyOS y μC/OS.	Sí, incluyendo FreeRTOS, SEGGER, embedOS y eCOS.
Tests funcionales y comprobaciones de integridad	Sí, conjunto de tests incluido	Sí, conjunto de tests incluido
Compatible con OpenSSL	Sí	Sí
Disponibilidad de una versión con funcionalidades mínimas	Sí	Sí

Cuadro 8.4: Comparación de WolfSSL y mbedTLS: características de compilación y configuración de la biblioteca

8.4. Elección de la biblioteca

Al contemplar las tablas 8.1 y 8.3 es evidente que WolfSSL y mbedTLS son las opciones más adecuadas. Por esta razón las hemos analizado más a fondo. La tabla 8.4 muestra una comparación en mayor profundidad de estas dos bibliotecas. Como podemos comprobar, ambas tienen características muy parecidas. Finalmente, nos hemos decantado por mbedTLS por los motivos siguientes:

- AWS IoT Device SDK funciona con mbedTLS de forma nativa. Este es un argumento de peso, ya que facilitará enormemente la integración del SDK de AWS IoT.
- La licencia parece más apropiada para el proyecto.

Con esta elección ponemos fin a este capítulo, en el que hemos analizado y comparado una serie de implementaciones TLS, tanto de código abierto como privativas. Destacaremos que de entre las alternativas comparadas, WolfSSL y mbedTLS parecen ser las más desarrolladas, tanto desde el punto de vista tecnológico como por la disponibilidad de documenta-

ción, presencia en las redes sociales y comunidad. Tras revisar la información disponible en los sitios web de WolfSSL y mbedTLS, resultan tener características muy similares. El principal motivo de la elección de mbedTLS ha sido su excelente compatibilidad con AWS IoT Device SDK, puesto que este último la integra por defecto.

Capítulo 9

Integración de mbedTLS

Tras elegir la implementación TLS que integraremos en el sistema, hemos llevado a cabo una evaluación en profundidad de la misma. Esta se ha basado principalmente en la documentación oficial de mbedTLS, disponible en su página web [45]. También hemos estudiado su código fuente y su documentación para aclarar algunos detalles [46].

Dedicaremos este capítulo a la integración de mbedTLS en el XMC4500 presente en el miniGW. Presentaremos aquí algunos detalles de la biblioteca, que serán de interés para integrarla al sistema. A continuación, discutiremos los detalles de esta integración, recorriendo cada uno de los elementos portados.

9.1. Detalles de la biblioteca

mbedTLS ofrece APIs tanto para la parte del cliente como para la del servidor, con soporte para todos los estándares SSL y TLS actuales. Además, la biblioteca incluye un amplio rango de algoritmos criptográficos. Así mismo, incluye una variedad de métodos de intercambio de claves, algoritmos de cifrado simétrico, algoritmos de hash, diversas curvas elípticas y herramientas para la generación de números aleatorios [47]. Además, mbedTLS dispone de todos los algoritmos criptográficos definidos en la *Suite B* de la NSA [44].

9.1.1. Estructura de ficheros

El código fuente de la biblioteca está estructurado de la forma que se presenta a continuación.

`/` en el directorio raíz se encuentran los ficheros que podemos encontrar en la mayoría de los proyectos, como `Makefile`, `README`, `LICENSE`, etc.

- **`configs`** en este directorio podemos encontrar diversos ejmplos de ficheros de configuración.
- **`doxygen`** ficheros de entrada para la documentación generada con Doxygen.

`include` ficheros de cabecera (`.h`).

`library` en este directorio se encuentran los ficheros `.c` de la biblioteca, que contienen el código fuente de los distintos módulos y funciones de la misma.

`programs` programas de ejemplo, que demuestran las distintas características de mbedTLS.

`scripts` scripts útiles para configurar mbedTLS y medir su rendimiento.

`tests` programas y scripts de test.

`visualc` proyecto de Microsoft Visual Studio, usado para compilar mbedTLS en entornos MS Windows.

`yotta` ficheros usados por Yotta para compilar mbedTLS como un módulo ARM mbed.

9.1.2. Opciones de configuración

Dado que muy pocos sistemas requieren de todos los algoritmos criptográficos disponibles, mbedTLS permite habilitarlos o deshabilitarlos en tiempo de compilación. De este modo, tanto el tamaño del ejecutable como la memoria requerida en tiempo de ejecución son minimizados, atendiendo únicamente a las necesidades de la aplicación. Para facilitar el proceso de configuración a los desarrolladores, la biblioteca incluye un fichero de configuración con todas las opciones disponibles. Este fichero de configuración es, en realidad, un archivo de cabecera en lenguaje C, que contiene una lista exhaustiva de definiciones con su correspondiente documentación. De esta forma, el desarrollador puede activar o desactivar cualquiera de las opciones. El fichero de configuración es referenciado por cada uno de los otros ficheros de cabecera, consiguiendo que la configuración se haga efectiva en tiempo de compilación [48].

Las opciones de configuración pueden ser divididas en cuatro grupos:

Soporte del sistema estas opciones deben seleccionarse de forma acorde a la plataforma sobre la que se ejecutará mbedTLS.

Soporte de características de seguridad estas opciones permiten elegir las características que son necesarias de cada módulo habilitado. De esta forma, pueden seleccionarse qué modos de cifrado estarán disponibles, qué curvas elípticas, qué algoritmos de intercambio de claves, etc. Se trata de opciones con una alta granularidad.

Módulos de mbedTLS estas opciones permiten habilitar o deshabilitar los módulos de mbedTLS de forma completa. Por ejemplo, podríamos deshabilitar completamente RSA o MD5 si no son necesarios

Opciones de configuración de los módulos estas opciones permiten ajustas opciones específicas para cada módulo, como el tamaño máximo de los números enteros, el tamaño de los búferes internos de SSL, etc.

Todas estas opciones pueden modificarse de forma manual o usando un script en Perl proporcionado con la biblioteca [49, 50].

9.1.3. Opciones de compilación

Con el fin ejecutar mbedTLS en la plataforma objetivo, tendremos que compilarlo con las opciones apropiadas, usando el conjunto de herramientas de compilación y enlazado necesarias. Tenemos cuatro formas de compilar la biblioteca [51].

Makefile a pesar de ser el método de compilación por defecto en sistemas Unix-like, no es recomendada, ya que dejó de ser mantenida por el equipo de desarrollo de mbedTLS.

CMake este método permite generar un fichero Makefile bien estructurado y de forma automática, detectando las dependencias y las opciones de compilación. Además, crea una estructura de directorios con la que se consigue que los ficheros objeto no se mezclen con el código fuente. Este es el método recomendado por el equipo de mbedTLS.

Yotta esta herramienta, creada en el marco del proyecto mbed de ARM, pretende ayudar a los desarrolladores que incluyen distintos módulos de mbed en sus proyectos. Está basada en CMake, pero cuenta con el concepto de módulos, de forma que resulta sencillo compilar distintos módulos que vayan a usarse en mbedOS.

MS Visual Studio el equipo de mbedTLS proporciona un fichero de proyecto de Microsoft Visual Studio con el objetivo de que los desarrolladores que usen esta IDE puedan compilar la biblioteca sin preocuparse demasiado por el proceo de compilación.

Aunque estos son los procedimientos de compilación provistos con la biblioteca, existen más opciones. Por ejemplo, podemos incluir el código fuente en un proyecto de DAVE™ y dejar que este lo compile de forma automática. Esto es posible gracias a que DAVE™ genera los ficheros objeto para cada fichero `.c`, buscando las cabeceras en los directorios que se le indique. De esta forma, DAVE™ comprobará si hemos realizado algún cambio en los ficheros de mbedTLS antes de cada compilación, por lo que no tendremos que compilar la biblioteca y el proyecto por separado. Esta opción es muy flexible, y será la que usaremos en este proyecto.

9.1.4. Portabilidad a otras plataformas

mbedTLS está programado en código C estándar, por lo que es fácilmente portable. Además, el código de esta biblioteca es muy modular, siendo los siguientes los únicos módulos que presentan dependencias de la plataforma [52].

Módulo de red (*Networking*) [requerido] las funciones de red están implementadas usando la API de sockets BSD. Gracias es esto, el módulo de red incluido con la biblioteca es compatible tanto con sistemas Unix como con MS Windows. A fin de que mbedTLS pueda establecer comunicaciones de red en un sistema empotrado, será necesario compilarlo junto a una biblioteca que proporcione una interfaz de sockets BSD.

Módulo de temporización (*Timing*) este módulo es requerido únicamente cuando nuestro sistema establezca comunicaciones a través del protocolo DTLS. De lo contrario, podemos deshabilitarlo de forma segura.

Fuentes de entropía por defecto (*Default entropy sources*) [requerido] el pool de entropía (*entropy pool*) se encarga de acumular y combinar la entropía de distintas fuentes de una forma segura. Estas fuentes pueden estar basadas en software o en hardware, y deben proporcionársele al módulo de generación de números aleatorios. Aunque las funciones de entropía pueden ser desactivadas sin afectar a ningún otro módulo, la generación de números aleatorios es esencial para muchas des las funciones criptográficas, por lo que tendremos que asegurarnos una buena fuente de entropía para hacer que nuestro sistema sea seguro.

Aceleración Hardware (*Hardware Acceleration*) muchos de los módulos que implementan primitivas criptográficas pueden ser sustituidos por implementaciones alternativas de las mismas. De esta forma, la es posible aprovechar implementaciones hardware que pudieran estar presentes en la plataforma.

Acceso al sistema de ficheros (*Filesystem access*) todas las funciones que acceden al sistema de ficheros están implementadas de forma que, en realidad, simplemente se lee un búfer de memoria. De este modo, solo es necesario portar las funciones que cargan/guardan el contenido de estos búferes del/al sistema de ficheros. Gracias a esta implementación, los desarrolladores pueden elegir si prefieren leer los datos del disco o incrustarlos «a fuego» en el código fuente (*hard-coded*).

Reloj en tiempo real (*Real-Time Clock*) algunos módulos acceden de forma opcional a la hora y fecha actuales, bien para medir intervalos de tiempo, bien para conocer la fecha y hora actuales. Este módulo solo es imprescindible a la hora de validar la vigencia o caducidad de los certificados X.509.

9.1.5. Programas de ejemplo

mbedTLS incluye numerosos programas de ejemplo, con los que pueden probarse las distintas características de la biblioteca. Por su relevancia en este proyecto, haremos referencia a dos de ellos. Estos programas han sido cuidadosamente estudiados, usando parte de su código en algunos pasos de la integración.

Certificate Authorities

`programs/x509/cert_app.c` demuestra el uso de certificados X.509 en mbedTLS. Como se demuestra en este programa, las autoridades certificadoras pueden cargarse a partir del sistema de ficheros haciendo uso de las funciones `mbedtls_x509_crt_parse_file()` y `mbedtls_x509_crt_parse_path()`. Estas funciones no son más que una envoltura (*wrapper*) de la función `mbedtls_x509_crt_parse()`, la cuál analiza el contenido de un búfer para obtener los distintos campos que conforman el certificado, guardándolos en una estructura definida en la biblioteca.

Connection sockets

`programs/ssl/ssl_client1.c` demuestra el uso de sockets a la hora de realizar una conexión TLS desde el lado del cliente. Los siguientes pasos deben seguirse para establecer la conexión.

1. Inicializar el contexto TCP mediante la función `mbedtls_net_connect()`. En este paso deben especificarse la dirección del servidor (IP o DNS), el puerto y el protocolo.
2. Configurar el contexto TLS con `mbedtls_ssl_config_defaults()`. En este paso deben definirse algunos parámetros de configuración de TLS, como la versión o las ciphersuites.
3. Iniciar la conexión segura con `mbedtls_ssl_set_bio()` y `mbedtls_ssl_handshake()`. Si todo va bien, después de este paso el handshake habrá concluido con éxito, por lo que podremos pasar a transmitir datos de forma segura.
4. Recibir y enviar datos de forma segura con `mbedtls_ssl_read()` y `mbedtls_ssl_write()`.
5. Para finalizar, debe cerrarse la conexión con `mbedtls_ssl_close_notify()`. No debemos olvidarnos de liberar la memoria dinámica que hayamos reservado en los pasos anteriores.

9.1.6. Comprobaciones de integridad

mbedTLS incluye programas de test, con los que puede verificarse el correcto funcionamiento de cada uno de los módulos de la biblioteca. Al ejecutarse, estos programas llaman a las funciones de los distintos módulos con parámetros conocidos y comprueban que el resultado obtenido sea el esperado, verificando de esta forma si los módulos trabajan de manera adecuada. Si se detecta algún error en los módulos examinados, el programa de test informará al usuario de la existencia del problema. Los tests que verifican la funcionalidad de un módulo concreto solo se ejecutarán si mbedTLS fue configurado para usar este módulo, es decir, si la biblioteca fue compilada con el módulo en cuestión.

Además de estos programas de test, mbedTLS incluye numerosos scripts que realizan pruebas automatizadas (*automated testing*). Estos scripts no se limitan a la comprobación de los distintos módulos de la biblioteca, sino que son capaces de verificar el proceso de compilación, realizar pruebas de interoperabilidad con OpenSSL y GnuTLS, etc. Estos scripts proporcionan un banco de pruebas muy completo, con más de 6000 tests [53, 54].

9.2. Integración de mbedTLS en el miniGW

Para llevar a cabo la integracion de mbedTLS en el miniGW, partiremos de un proyecto inicial en el que están presentes CMSIS-RTOS y lwIP. Este proyecto de partida no realiza

ninguna función en especial, pero tiene una pila IP funcional, gracias a la cuál pueden establecerse conexiones TCP con gran facilidad. Además, lwIP responde a los *pings* de ICMP por defecto, lo cuál es muy útil para comprobar la conectividad.

9.2.1. Importación de mbedTLS al proyecto

Para integrar cualquier biblioteca, lo primero que tendremos que hacer es importarla al proyecto. De esta forma, el IDE la compilará automáticamente con el resto del código, pudiéndose referenciar los objetos de la misma desde cualquier lugar. Existen dos formas principales de importar la biblioteca al proyecto. La primera de ellas consiste en copiar los ficheros de la biblioteca al directorio del proyecto. De esta forma, todo el código fuente, tanto el del programa usuario como el de la biblioteca, se encontrará bajo el mismo directorio, siendo accesible al proceso de compilación. Sin embargo, esta aproximación presenta algunos inconvenientes, sobre todo cuando se trata de usar una misma biblioteca que está sujeta a cambios en distintos proyectos. En este caso, DAVE™ nos ofrece una alternativa, que hereda de Eclipse IDE. La alternativa consiste en *enlazar* la biblioteca al proyecto, de forma que en el mismo se almacena únicamente una referencia a la biblioteca. De esta forma, cualquier cambio que realicemos a la biblioteca dentro del proyecto quedará guardado en la misma. Este flujo de trabajo es especialmente conveniente cuando la biblioteca se tiene bajo control de versiones, en un repositorio distinto al del proyecto y que es compartido por todo el equipo[1].

Una vez que el código fuente de la biblioteca está accesible y puede compilarse en conjunto con el resto del proyecto, será necesario ajustar algunos parámetros de la compilación. En concreto, será necesario añadir algunos símbolos y rutas, que en DAVE™, al igual que en Eclipse IDE, puede hacerse de forma gráfica accediendo a las propiedades de compilación del proyecto. En el caso que nos ocupa, definiremos un símbolo `DAVE`, sin ningún valor específico, para usarlo como indicador de la plataforma cuando modifiquemos el código fuente de la biblioteca al portarla. Esto en GCC se consigue con el flag '`-D`'. También tendremos que indicar al compilador que busque archivos de cabecera contenidos en `mbedtls/include` añadiendo esta ruta con el flag '`-I`'.

9.2.2. Adaptación de la biblioteca a la plataforma

Una vez importado el código de la biblioteca y establecidos los parámetros de compilación, pasaremos a realizar los cambios necesarios en la biblioteca para que esta pueda compilarse y ser funcional en la nueva plataforma. Puesto que mbedTLS selecciona la plataforma a través de directivas condicionales del preprocesador de C (es decir, a través de directivas `#if` `#else`), tendremos que modificar cada uno de los ficheros dependientes de la plataforma. Como explicamos en la sección 9.1.4, los módulos dependientes de la plataforma son el módulo de red y la fuente de entropía. A continuación describiremos los cambios realizados en cada uno de estos módulos.

[1] Git submodules es otra forma de resolver este problema. Sin embargo, el flujo de trabajo de eesy-innovation no contempla el uso de submódulos.

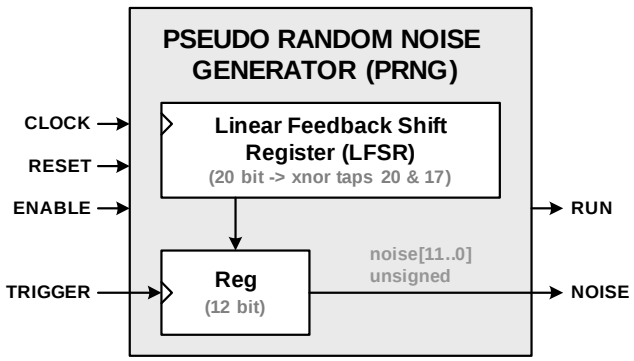


Figura 9.1: Diagrama de bloques del PRNG disponible en el XMC4500 [39].

Módulo de red

Con respecto a las capacidades de red, mbedTLS se basa en los sockets BSD estándar, que es la interfaz de red predeterminada en muchos sistemas tipo Unix, así como lo es en MS Windows. Las versiones recientes de lwIP proporcionan esta interfaz de sockets estándar, por lo podremos usar esta biblioteca como la pila de red del sistema.

Volviendo a mbedTLS, hemos modificado el fichero `net_sockets.c`, que es una capa de abstracción donde se interconecta mbedTLS con los sockets tipo BSD. Esta es la lista de cambios que hemos realizado en el fichero:

- Permitir compilar este archivo para un entorno distinto a Unix y MS Windows, incluyendo directivas `#if` al comienzo del fichero.
- Uso de lwIP como la pila de red, incluyendo los ficheros de cabecera al comienzo de `net_sockets.c`, bajo la condicional `#ifdef DAVE`, donde `DAVE` es el símbolo definido anteriormente.
- Deshabilitar IPv6 cuando esté definido el símbolo `DAVE`, ya que esta versión del protocolo no está soportado por lwIP en la fecha de realizar esta integración.
- Uso de la implementación de la función `fcntl()` provista por la biblioteca *newlib*, incluida en el kit de herramientas de compilación *arm-none-eabi* de DAVE™.

Fuentes de entropía

Una buena entropía es la base fundamental para una criptografía robusta. Esta entropía puede obtenerse de diversas maneras. Los módulos de hardware que proporcionan entropía, como los módulos de generación de números (pseudo-)aleatorios (*pseudo-random number generator*, PRNG) están presentes en muchos microcontroladores. El microcontrolador XMC4500 incluye uno de estos módulos PRNG como parte de los generadores de señal del módulo DAC. Utilizaremos este módulo como fuente principal de entropía en este proyecto.

El PRNG presente en el XMC4500, cuyo diagrama de bloque puede consultarse en la figura 9.1 genera números pseudoaleatorios de 12 bits. Para ello, utiliza un registro de

desplazamiento de realimentación lineal (*linear-feedback shift register*, LFSR) de 20 bits que funciona a la frecuencia de reloj del sistema. Al recibir una señal de *trigger*, el valor presente en el LFSR se copia al registro de salida, permaneciendo en este para su lectura por otro módulo. Cada vez que se reinicia el PRNG, el LFSR vuelve a su valor inicial y, por lo tanto, siempre se repite la misma secuencia de números pseudoaleatorios [39]. A pesar de esto, hay tres factores que deberían garantizar una buena entropía en la implementación.

1. Existen muchos factores que hacen que el sistema no sea determinista, por lo que el momento en que se leen los números aleatorios es cada vez distinto. Entre estos factores podemos contar con el propio RTOS, que no garantiza que las tareas se ejecuten siempre en el mismo orden, o el hecho de que los tiempos de transmisión en la red sean cada vez distintos.

2. El PRNG forma parte del módulo DAC del microcontrolador, por lo que será necesario transformar la señal analógica generada a un número digital, a través de un ADC. Esta lectura está sujeta a pequeñas variaciones, pues la señal analógica generada se verá afectada por distintos ruidos, siempre presentes en los circuitos electrónicos [55]. Además, usaremos una frecuencia relativamente alta en la generación de números aleatorios, de forma que el ADC no llegue a estabilizarse en la lectura y siempre cometa un error [56].

3. mbedTLS toma estos números aleatorios como una fuente de entropía, para luego combinarla con otras fuentes en el recolector de entropía. Esto se hace con la finalidad de asegurar una generación de números lo más aleatorios posibles [57].

El uso del PRNG en el proyecto se ha llevado a cabo mediante el uso de diversas DAVE™ APPs, con las que hemos configurado uno de los módulos DAC para la generación de ruido y uno de los ADC para leer la señal generada.

Para integrar esta fuente de entropía en mbedTLS hemos modificado el fichero `entropy_poll.c`, que contiene las funciones de entropía definidas por el usuario para cada plataforma. En este fichero, hemos añadido una sección específica para la plataforma `DAVE`. En esta sección, simplemente hemos completado la función `mbedtls_hardware_poll()` para que lea números aleatorios a partir del ADC. Esto último se consigue haciendo uso de las APIs correspondientes de la DAVE™ APP del ADC.

Módulo de temporización

Aunque este módulo no es un requisito para el funcionamiento de la biblioteca, será necesario a la hora de verificar la validez del certificado de los servidores de AWS. Para integrar el RTC con mbedTLS, hemos creados los fichero `platform_DAVE.c` y `platform_DAVE.h`, referenciándolos luego en `main.c`. Más concretamente, en la secuencia de inicialización llamamos a la función `mbedtls_platform_set_time()`, pasándole como parámetro la función `mbedtls_time_DAVE()` definida en `platform_DAVE.c`. Esta última función realiza una lectura del RTC del microcontrolador a través de la API definida en una DAVE™ APP.

La fecha y hora del RTC serán ajustadas gracias a un cliente NTP sencillo, que actualizará la hora periódicamente basándose en una lista de servidores. El código fuente de este cliente ha sido proporcionado por el equipo de desarrollo de eesy-innovation.

9.2.3. Configuración de mbedTLS

Una vez integrados los diferentes módulos con la plataforma, será necesario configurar mbedTLS para que pueda compilar. La configuración de mbedTLS se lleva a cabo modificando el fichero `config.h`. La biblioteca incluye varios de estos ficheros para ejemplificar algunas configuraciones comunes. Para llevar a cabo la configuración necesaria, hemos partido del fichero `config-suite-b.h`, que presenta una configuración mínima para la Suite B definida por la NSA. Esta configuración trata de minimizar el uso de RAM y emplea únicamente criptografía de curva elíptica (ECC) como algoritmos de clave pública, satisfaciendo los requisitos de AWS IoT en cuanto a la ciphersuite.

Destacamos la adición de los siguientes símbolos en nuestra configuración [48]:

`MBEDTLS_SELF_TEST` habilita las funciones de test, para poder verificar la integridad de la biblioteca.

`MBEDTLS_PLATFORM_C` habilita la capa de abstracción de plataforma que permite reasignar funciones como `calloc()`, `printf()`, etc. Es necesaria para reasignar las funciones de lectura del RTC.

`MBEDTLS_HAVE_TIME_DATE` habilita la verificación de la validez de los certificados digitales basada en la fecha y hora. Requiere la integración del RTC.

`MBEDTLS_PLATFORM_TIME_ALT` permite que mbedTLS soporte funciones alternativas en el módulo de temporización, que estarán definidas en la capa de abstracción de la plataforma.

`MBEDTLS_NO_PLATFORM_ENTROPY` evita que mbedTLS use las funciones de entropía definidas por defecto, y que requieren de sistemas tipo Unix o MS Windows.

`MBEDTLS_ENTROPY_HARDWARE_ALT` permite que mbedTLS use una implementación propia de un recolector de entropía hardware.

Además de estos nuevos símbolos, el fichero de configuración incluye opciones para reducir el uso de memoria de mbedTLS. El uso de algunas de estas opciones requiere conocer cuáles son los requisitos específicos de la comunicación con el servidor, ya que la reducción del uso de memoria consiste, en muchos casos, en limitar los *ciphersuites* disponibles y el tamaño de las operaciones. A continuación destacamos alguna de estas opciones [50]:

`MBEDTLS_AES_ROM_TABLES` almacena las tablas AES en la ROM, evitando consumo de RAM.

`MBEDTLS_ECP_MAX_BITS` limita el tamaño máximo reservado para las curvas elípticas.

`MBEDTLS_MPI_MAX_SIZE` limita el tamaño máximo de los enteros de precisión múltiple (*multiple-precision integers*, MPI). El uso de esta opción requiere conocer cuáles son los requisitos de la comunicación.

`MBEDTLS_SSL_CIPHERSUITES` permite definir las *ciphersuites* que serán requeridas. Solo estas estarán disponibles en tiempo de ejecución, evitando que mbedTLS incluya código de otras en su compilación y reduciendo por tanto el tamaño del programa.

Con esta configuración, hemos podido compilar el código de la biblioteca y llamar a sus funciones desde las distintas tareas del RTOS. En la siguiente sección hablamos de la verificación de la integración llevada a cabo.

9.3. Verificación de la integración

Llegados a este punto tenemos una integración de mbedTLS que compila en el sistema, y cuyas funciones pueden ser usadas desde las distintas tareas, siempre y cuando se incluyan los archivos de cabecera al inicio del código fuente de la tarea. A continuación comprobaremos que la biblioteca se comporta de la manera esperada, y que sus funciones producen una respuesta adecuada. Para ello, usaremos los programas de test de los que hablamos en la sección 9.1.6. Una vez que llevemos a cabo este proceso de auto-verificación, pasaremos a realizar un test funcional, que consistirá en descargar una web a través de HTTPS.

Tests unitarios: `selftest.c`

Hemos usado el fichero `selftest.c`, localizado en el directorio `programs/test`, como base para verificar el correcto funcionamiento de los módulos en la integración. Este programa cuenta el número de módulos comprobados en las variables `suites_tested`, cuando la comprobación se ha realizado con éxito, y `suites_failed`, en caso contrario.

Puesto que este fichero está diseñado para ejecutarse de forma independiente y no como parte de un programa más grande, hemos tenido que cambiar el nombre de la rutina `main()` por `self_test()`, ya que el primero está reservado como punto de entrada del programa principal. El firmware resultante se ejecuta en los siguientes pasos:

1. Inicialización del sistema, a través de la inicialización de las DAVE™ APPs mediante la función `DAVE_Init()`. Se inicializan tanto módulos hardware y componentes software, como las variables de CMSIS-RTOS.

2. Configuración de mbedTLS para usar el RTC, tal y como hemos explicado en la sección anterior.

3. Creación de una tarea de ejecución única, que llamará a la función `self_test()`.

4. Inicialización del núcleo del RTOS, mediante la función `osKernelStart()`.

Gracias a la interfaz de depuración hemos podido observar la ejecución del programa y la evolución de las variables `suites_tested` y `suites_failed`. En este proceso hemos podido comprobar que la totalidad de los tests han sido superados con éxito, por lo que todos los módulos funcionan correctamente en la integración llevada a cabo.

Test funcional: *cliente HTTPS sencillo*

Para comprobar que mbedTLS funciona en su conjunto y que es posible establecer comunicación con el exterior, hemos probado el programa de ejemplo `ssl_client1.c`. Este programa, que se encuentra en el directorio `programs/ssl`, establece conexión con un servidor HTTPS[2] y realiza una petición `HTTP GET`, imprimiendo la respuesta del servidor si todo ha ido bien.

Tal y como hemos hecho con el test anterior, hemos modificado este programa de ejemplo para que funcione como una tarea en RTOS. También hemos tenido que ajustar algunas líneas de `ssl_client1.c`. En concreto, hemos redefinido `SERVER_NAME` con el valor «https://www.wikipedia.org/».

Para completar este test, hemos tenido que conectar el miniGW a Internet. Esto lo hemos conseguido haciendo uso de un ordenador con GNU/Linux, que ha sido configurado para redireccionar el tráfico de red entre el puerto Ethernet y una conexión WiFi. De esta forma también hemos podido ver los paquetes de datos con el software de análisis de red Wireshark. Una vez hecho esto, hemos tenido que configurar lwIP y la interfaz Ethernet del computador convenientemente, a fin de que las interfaces de ambos dispositivos se encuentren en la misma subred IP y para que el computador se use como pasarela de red del miniGW.

Al ejecutar el programa y analizar algunas variables con el depurador, hemos podido ver la respuesta del servidor, que ha aceptado nuestra petición devolviendo un mensaje del tipo `200 OK`.

9.4. Resolución de problemas

En la sección anterior hemos presentado los resultados obtenidos al verificar la integración de mbedTLS. Como hemos comentado, tras ajustar los programas de ejemplo e integrarlos en el sistema hemos conseguido que la biblioteca funcione correctamente. Sin embargo, hemos tenido algunas dificultades en la integración que han ido más allá de las configuraciones expuestas anteriormente. En esta sección presentaremos estos problemas y la forma en que los hemos solucionado.

[2] HTTPS es una extensión del protocolo de aplicación HTTP, que consigue una comunicación segura gracias a una capa adicional de SSL o TLS. Por este motivo, a veces HTTPS es referido como *HTTP sobre SSL* o *HTTP sobre TLS*.

Configuración de CMSIS-RTOS

Como mencionamos en el capítulo 3, las funciones criptográficas suelen ser computacionalmente costosas, por lo que no solo requieren numerosos ciclos de reloj, sino que también hacen un uso intensivo de la memoria. Por este motivo, para hacer que mbedTLS funcione en nuestro sistema, hemos tenido que aumentar la memoria disponible para las tareas en RTOS hasta 10KB. Debe tenerse en cuenta que este valor se refiere exclusivamente a la memoria disponible para la pila, es decir, para las variables declaradas en las funciones de mbedTLS y de los programas de ejemplo. No tenemos una forma directa de saber cuánto heap está siendo usado por las funciones criptográficas, aunque podemos presumir que algo más.

Otra configuración que hemos tenido que realizar en este punto ha sido incrementar el número máximo de tareas definidas por el usuario que pueden ejecutarse en paralelo, para dar cabida a la tarea de test y a las requeridas por lwIP.

Actualización de lwIP

Al tratar de compilar la integración de mbedTLS en DAVE™, encontramos numerosos errores de compilación. Estos errores hacían referencia a funciones inexistentes y tipos de variable desconocidos, todos relacionados con la pila de protocolos IP. Tras analizar detenidamente el problema, nos dimos cuenta de que la versión de lwIP que se incluye con la DAVE™ APP estaba obsoleta, por lo que fue necesario llevar a cabo una actualización.

El proceso de actualización ha requerido analizar la APP en busca del código fuente de lwIP y de los ficheros de configuración, para sustituir el directorio donde se encuentra el código fuente de lwIP. Hemos usado la versión de lwIP 2.0.0, por ser la versión estable más reciente en el momento de realizar este trabajo. Finalmente, hemos tenido que aplicar una serie de cambios a la biblioteca y a la configuración proporcionada por la APP, en aras de la compatibilidad con el sistema.

La metodología de trabajo en este caso ha pasado por analizar los mensajes de error que arrojaba el compilador y realizar las correcciones pertinentes en el código. Puesto que esta actualización debe ser realizada y comprendida posteriormente por la empresa, hemos desarrollado un script que lleva a cabo la actualización de forma automática. Este script no hace más que copiar los ficheros de lwIP 2.0.0 al directorio correspondiente y aplicar un parche tanto a la biblioteca como a la APP. Este parche ha sido generado usando el comando `git diff`, para recoger las diferencias entre, por un lado, la versión del proyecto con lwIP recién copiado a la versión 2.0.0, y por otro, con lwIP funcional después de ajustar la APP. Este script tiene que ejecutarse cada vez que se modifica la configuración de las DAVE™ APPs, ya que al generar el código de las mismas se revierten los cambios realizados manualmente al código fuente de estas, y, en concreto, se revierten los cambios realizados en la APP de lwIP.

Entre los cambios recogidos en este parche, cabe destacar la inclusión de los nuevos archivos de cabecera y el renombramiento de algunos tipos y símbolos. Además, después de aplicar los cambios mencionados, hemos tenido que actualizar las propiedades de com-

pilación para incluir algunas rutas de archivos de cabecera, que se encuentran en directorios distintos ahora.

Configuración de mbedTLS

La configuración de mbedTLS expuesta anteriormente es la mínima requerida para tener la Suite B de la NSA funcionando en nuestro sistema. Con esta configuración, el proyecto compila y podemos verificar que los módulos de mbedTLS funcionan correctamente. Sin embargo, hemos tenido que realizar algunas modificaciones para poder realizar el test funcional y conectarnos con el servidor HTTPS. En concreto, hemos tenido que habilitar RSA como método para criptografía de clave pública e intercambio de claves. También hemos tenido que eliminar algunas de las opciones para reducir el uso de memoria, ya que estas limitaban las *ciphersuites* disponibles y el tamaño de las claves criptográficas. Finalmente, hemos tenido que incrementar el tamaño de los búferes de recepción a 2048 bytes, pues los paquetes de respuesta del servidor superaban el tamaño del búfer (`MBEDTLS_SSL_MAX_CONTENT_LEN  2048`).

Capítulo 10

Integración de AWS IoT Device SDK

Amazon Web Services proporciona un SDK oficial para conectarse con AWS-IoT desde cualquier dispositivo. Este kit de desarrollo, denominado *AWS IoT Device SDK para C embebido*, está escrito en el lenguaje de programación C y puede portarse a cualquier tipo de dispositivo. Además, su código fuente está disponible en GitHub (https://github.com/aws/aws-iot-device-sdk-embedded-C), de donde podemos descargarlo. Toda la información acerca de AWS IoT Device SDK está disponible en https://aws.amazon.com/iot/sdk/.

10.1. Detalles de la biblioteca

El Device SDK de AWS IoT para C embebido es una colección de ficheros C diseñada para conectar aplicaciones embebidas a la nube de AWS IoT de forma segura. El SDK incluye clientes de la capa de transporte, una implementación de TLS (mbedTLS) y ejemplos de uso. También da soporte a características específicas de AWS IoT, como una API para acceder al *Device Shadow service*. Se distribuye en la forma de código fuente, de modo que deberá compilarse junto con la aplicación deseada, permitiendo la existencia de otras bibliotecas y RTOS [58].

10.1.1. Estructura de ficheros

A continuación se presenta la estructura de ficheros del AWS IoT Device SDK [59].

`certs` este directorio es el que albergará los certificados digitales necesarios para la autenticación mutua entre el dispositivo y los servidores de AWS IoT. Aquí se almacenarán el certificado del cliente, su clave privada y la autoridad certificadora con la que se verificará la autenticidad del servidor.

`docs` documentación de la API del SDK.

`external_libs` código fuente de las bibliotecas de las que depende el Device SDK: mbedTLS y jsmn[1].

`include` este directorio contiene los archivos de cabecera que las aplicaciones deben incluir para hacer uso del SDK.

`src` este directorio contiene el código fuente del SDK, incluyendo la biblioteca MQTT, el código de la sombra del dispositivo y otras utilidades.

`platform` este directorio contiene los ficheros dependientes de la plataforma, tales como los temporizadores (*timers*), la implementación de TLS y la capa de concurrencia. Por defecto, el directorio incluye una implementación para GNU/Linux usando mbedTLS y pthread.

`samples` este directorio contiene algunos programas de ejemplo, así como sus Makefiles. Entre estos programas, cabe destacar dos ejemplos: el primero demuestra el uso de MQTT, a través de la suscripción y publicación en un *topic* de AWS IoT; el segundo muestra cómo interactuar con la sombra del dispositivo.

`tests` contiene tests para verificar la funcionalidad del SDK.

10.1.2. Portabilidad a otras plataformas

En esta sección, explicaremos los elementos del AWS IoT Device SDK que deben ser portados para hacer que este pueda ejecutarse en una nueva plataforma. Cabe destacar que las interfaces del SDK siguen un donde solo los prototipos están definidos por el Device SDK, mientras que la implementación corre a cargo del usuario del SDK, que deberá ajustarla a la plataforma en uso.

A continuación se presentan las funcionalidades necesarias para que el Device SDK se ejecute correctamente en una plataforma arbitraria [59].

Timer Functions [requerido]

El SDK necesita una implementación de temporizador para gestionar los tiempos de espera de las peticiones MQTT (como *connect* y *subscribe*, entre otros comandos), así como para el mantenimiento de la conexión (*MQTT keep-alive pings*). Los temporizadores deberán tener una resolución de milisegundos. Puesto que los temporizadores se consultan para comprobar si ha transcurrido un determinado intervalo de tiempo, es posible usar un contador de ejecución del tipo «milisegundos desde el inicio».

[1] Jsmn (pronunciado como 'jasmine') es una biblioteca para dar soporte a JSON en C. Gracias a su implementación minimalista, se puede integrar fácilmente proyectos con limitaciones de recursos, incluyendo sistemas embebidos.

Network Functions [requerido]

Para que la pila de clientes MQTT pueda comunicarse a través de la pila de protocolos TCP/IP utilizando una conexión TLS autenticada mutuamente, es necesario contar con una implementación de las funciones de red. La biblioteca TLS proporciona generalmente la API para el socket TCP subyacente.

Threading Functions

La capa de concurrencia proporciona la implementación de *mutexes* usados para operaciones *thread-safe*. Aunque el cliente MQTT utiliza una máquina de estados para controlar las operaciones un un entorno multi-hebra, requiere la implementación de los mutexes para garantizar la seguridad de las hebras (*thread safety*). Esto no es necesario en los casos en que la seguridad de las hebras no es importante, por lo que está deshabilitado por defecto.

10.2. Integración de AWS IoT Device SDK en el miniGW

Para llevar a cabo la integración de AWS IoT Device SDK, partiremos del proyecto con mbedTLS, sin ningún programa de ejemplo. Esto ha sido posible gracias al uso de diferentes ramas en git. En nuestro caso, hemos separado la rama principal de desarrollo (`dev`) de las ramas en las que hemos integrado una prueba del sistema (`test-`), de modo que ha sido muy fácil revertir los cambios y llevar a cabo distintas líneas de desarrollo y de corrección de errores.

10.2.1. Importación de AWS IoT Device SDK al proyecto

Al igual que en el caso de mbedTLS, tendremos que importar la biblioteca al proyecto. De nuevo, enlazaremos sus ficheros usando las propiedades del proyecto en DAVE™. En este caso, solo necesitamos los directorios `include`, `src` y `external_libs/jsmn`, por lo que prescindiremos del resto. Como ya hemos importado mbedTLS y hemos añadido las rutas de los archivos de cabecera, no será necesario introducir la biblioteca en el directorio `external_libs`, aunque podemos hacerlo para una mejor organización del código fuente.

Para terminar de importar AWS IoT Device SDK y permitir que el compilador encuentre los archivos de cabecera referenciados por la biblioteca, será necesario añadir algunas rutas en las opciones de compilación. En concreto, hemos añadido las siguientes entradas con el flag '`-I`', para que se encuentren los archivos de cabecera:

- `AWS-IoT-SDK/include`
- `AWS-IoT-SDK/external_libs/jsmn`

10.2.2. Definición de una nueva plataforma

A diferencia de mbedTLS, el Device SDK de AWS IoT se basa en el modelo de controladores, con el que se intenta facilitar la portabilidad a otras plataformas. Siguiendo este modelo, la biblioteca incluye un directorio `platform`, donde se colocan los ficheros dependientes de plataforma de una forma estructurada. De esta manera, es posible definir varias plataformas, usándose solo la que se incluya en las opciones de compilación del proyecto. Otra particularidad de este modelo es que las funciones que se encuentran en este directorio son, en realidad, envoltorios (*wrappers*), dentro de los cuáles se llama a las funciones que realizarán el trabajo real en cada caso. Esta estructura dota a la biblioteca de una gran versatilidad y hace muy sencilla su integración en nuevos sistemas. Añadiremos el directorio `platform/DAVE4_CMSIS_RTOS`, con la estructura encontrada en `platform/Linux`.

A continuación se listan los elementos que han debido portarse. Como hemos mencionado, será necesario añadir los subdirectorios de `port` a la ruta de búsqueda de archivos de cabecera.

Temporizadores

Estas funciones se encargan de los *timeouts*. Rellenaremos los *wrappers* encontrados en el fichero `timer.c` e incluiremos los archivos de cabecera necesarios en `timer_platform.h`, ambos ficheros encontrados en el directorio `platform/DAVE4_CMSIS_RTOS/common`. Las funciones definidas harán uso de SysTick, disponible a través de CMSIS-RTOS, para proporcionar las funcionalidades requeridas.

Concurrencia

Al igual que con los temporizadores, completaremos las funciones del fichero `threads.c` y añadiremos las cabecereas necesarias a `threads_platform.h`, ambos en el directorio `platform/DAVE4_CMSIS_RTOS/pthreads`. En esta ocasión, usaremos los mutexes proporcionados por CMSIS-RTOS para conseguir las funcionalidades necesarias.

Conectividad de red

El directorio `platform/DAVE4_CMSIS_RTOS/mbedtls` contiene las funciones que se encargan de inicializar los contextos TLS y usar los sockets de manera conveniente. Puesto que AWS IoT Device SDK usa mbedTLS por defecto, no ha habido que realizar grandes cambios aquí. Simplemente, hemos tenido que modificar algunas líneas de `network_mbedtls_wrapper.c`, en concreto, la función `iot_tls_init()`, para hacer que no cargue los certificados a partir de ficheros, sino que *parsee* certificados contenidos en variables directamente.

`aws_iot_config.h`

El Device SDK de AWS IoT requiere de un archivo de configuración nombrado `aws_iot_config.h` que debe ser accesible al compilador. Este fichero, que contiene tanto información sobre la conexión como configuraciones de los módulos, se incluye en algunas de las cabeceras de estos mismos módulos. Por ello, tendremos que importar este fichero en nuestro proyecto, copiándolo de cualquiera de los ejemplos que se proveen (por ejemplo, en `samples/linux/shadow_sample`).

10.2.3. Configuración de AWS IoT SDK

Llegados a este punto tenemos una integración de AWS IoT Device SDK que compila en el sistema, y cuyas funciones pueden ser usadas desde las distintas tareas, siempre y cuando se incluyan los archivos de cabecera al inicio de su código fuente. A continuación llevaremos a cabo algunas configuraciones que serán necesarias para establecer una comunicación con la nube de Amazon.

Certificados digitales y clave del dispositivo

Como explicamos en la sección 10.1.1, existe un directorio destinado a albergar los certificados y claves que el dispositivo necesita para llevar a cabo la autenticación mutua con el servidor. Normalmente, estos certificados se almacenarían como ficheros en formato `.pem` o `.der`, y serían abiertos por mbedTLS accediendo al sistema de ficheros. Sin embargo, nuestro sistema no cuenta con dicho sistema de ficheros, por lo que almacenaremos los certificados y la clave en unas constantes, que se definirán en un fichero `.c`. De esta forma, los certificados y claves permanecerán almacenados directamente en el firmware (*hardcoded*) por lo que será imposible acceder a ellos si no se dispone de una interfaz de depuración[2]. Por simplicidad, las constantes de este fichero no serán más que cadenas de caracteres ASCII con el contenido literal de los certificados y de la clave privada en formato PEM [28, 60].

Este fichero, que llamaremos `certificates.c`, será el responsable de almacenar el certificado del cliente (`client_cert`), su clave privada (`client_key`) y la autoridad certificadora con la que se verificará la autenticidad del servidor (`ca_root`). Estas constantes serán usadas posteriormente por el programa de ejemplo a la hora de establecer la conexión con el servidor. El contenido de estas constantes se obtendrá de la consola de AWS IoT, como se explicará en la siguiente sección.

`aws_iot_config.h`

Una vez integrada la biblioteca, tendremos que configurar algunos parámetros para conectarnos con los servidores de AWS. En concreto, necesitaremos los siguientes elementos para llevar a cabo la conexión. Estos elementos serán modificados en el fichero

[2] Aun con el hardware necesario para la depuración, existen formas de proteger el código contra lectura, de forma que la clave privada sea realmente inaccesible.

`aws_iot_config.h`, el cuál se importará al proyecto a partir del programa de ejemplo `subscribe_publish_sample`[3].

Como se vio en el capítulo 6, es necesario que el dispositivo tenga algunos parámetros de conexión de los servidores de AWS. Modificaremos las siguientes línas de `aws_iot_config.h` con los parámetros obtenidos de la consola de AWS IoT. .

`AWS_IOT_MQTT_HOST` URL del *personal endpoint* asociado a AWS IoT. Puede encontrarse en la pestaña *Settings* de la consola, y tiene el formato `xxxxxxx.region-key.amazonaws.com`.

`AWS_IOT_MQTT_PORT` puerto para la conexión MQTT. Dejaremos el valor por defecto (8883).

`AWS_IOT_MQTT_CLIENT_ID` ID del cliente MQTT, que deberá ser distinto para cada dispositivo. Este ID será una cadena alfanumérica definida en producción. Puesto que en nuestro test solo tendremos un dispositivo, podemos dejarlo en su valor por defecto.

`AWS_IOT_MY_THING_NAME` nombre del dispositivo virtual definido en la plataforma.

`AWS_IOT_ROOT_CA_FILENAME` variable o constante donde se almacena el certificado digital de la autoridad certificadora, con la que el dispositivo autenticará a los servidores de AWS IoT. Nótese que por los cambios realizados en la biblioteca, aquí no se indicará un nombre de archivo, sino el contenido del mismo. En este caso, será la constante `ca_root`, definida en la sección anterior.

`AWS_IOT_CERTIFICATE_FILENAME` variable o constante donde se almacena el certificado digital del dispositivo. Este certificado estará asociado al dispositivo virtual definido en la consola. Nótese que por los cambios realizados en la biblioteca, aquí no se indicará un nombre de archivo, sino el contenido del mismo. En este caso, será la constante `client_cert`, definida en la sección anterior.

`AWS_IOT_PRIVATE_KEY_FILENAME` variable o constante donde se almacena la clave privada del dispositivo. Nótese que por los cambios realizados en la biblioteca, aquí no se indicará un nombre de archivo, sino el contenido del mismo. En este caso, será la constante `client_key`, definida en la sección anterior.

El resto de parámetros definidos en el fichero se han dejado intactos, con sus valores por defecto. Estos parámetros controlan la configuración de búferes, uso de la sombra y otros parámetros de conexión. Además, incluiremos las constantes de los certificados digitales como `extern`, para que las mismas puedan ser usadas donde se requiera.

10.3. Verificación de la integración

A continuación comprobaremos que la integración de AWS IoT Device SDK se comporta de la manera esperada, realizando una conexión con AWS IoT y comprobando que

[3] En realidad, los ficheros de configuración de todos los programas de ejemplo son idénticos.

existe una transmisión de datos. Para ello, será necesario establecer un escenario de prueba, que definiremos a continuación. Solo llevaremos a cabo dos tests funcionales, ejecutando el código de los ejemplos `subscribe_publish` y `shadow` que se proporcionan en el directorio `sample`. Estos tests requerirán que configuremos de forma conjunta la plataforma y el dispositivo. En esta sección explicaremos las configuraciones requeridas por estos tests en particular, que van un poco más allá de las expuestas en el epígrafe anterior..

Disposición de los elementos de prueba

La configuración de prueba que llevaremos a cabo es la que podemos ver, de forma esquemática, en la figura 10.1. Como se muestra en la misma, conectaremos el miniGW a Internet a través de un computador GNU/Linux que hará las veces de puerta de enlace, gracias a que lo configuraremos para realizar *IP forwarding*. Esta conexión será monitorizada con Wireshark, donde veremos los paquetes transmitidos en ambas direcciones. Gracias a este software, podremos comprobar si se lleva a cabo la conexión, en qué puntos se producen errores en la misma, si hay paquetes erróneos, etc. En este mismo computador tendremos una terminal serial para visualizar la salida de las trazas de depuración definidas en el firmware. Esta conexión serial se usa para comprobar que se realizan todos los pasos correctamente (inicialización del RTOS, inicialización de los módulos, distintos pasos de la conexión, envío y recepción de mensajes, etc.), pero no está configurada para interactuar con el miniGW.

Por otro lado, tendremos un computador con la interfaz de depuración de DAVE™, con la que cargaremos el firmware cada vez que hagamos cambios. DAVE™ también nos permitirá usar las herramientas de depuración, como puntos de ruptura (*breakpoints*), con las que podremos analizar los problemas con más detalle. En este computador también tendremos abierta la consola de AWS IoT, desde la que veremos si la conexión se ha establecido y podremos analizar el intercambio de mensajes MQTT.

Configuración conjunta de la consola de AWS IoT y del Device SDK

Como hemos indicado anteriormente, para conectar el dispositivo a la Nube, tendremos que definir un dispositivo virtual en la plataforma. También tendremos que asignarle un certificado digital y una política de acceso.

En primer lugar, deberemos crear el dispositivo virtual en la plataforma. Tendremos que definir los distintos parámetros de nuestro dispositivo, como su nombre, que tendremos que definir de igual manera en el dispositivo físico (`AWS_IOT_MY_THING_NAME`). Este es, en realidad, el único campo requerido. Sin embargo, podemos definir algunos atributos, como la versión del *firmware* o los sensores que implementa el dispositivo. Recordemos que estos atributos sirven para describir el dispositivo, y no deben confundirse con el estado del mismo. De forma alternativa, nuestro dispositivo podría heredar los atributos de un tipo que hayamos definido previamente.

En la consola de AWS IoT, podemos acceder a la pestaña *Interact* del dispositivo , donde se mostrará la información que tendremos que copiar para configurar el fichero

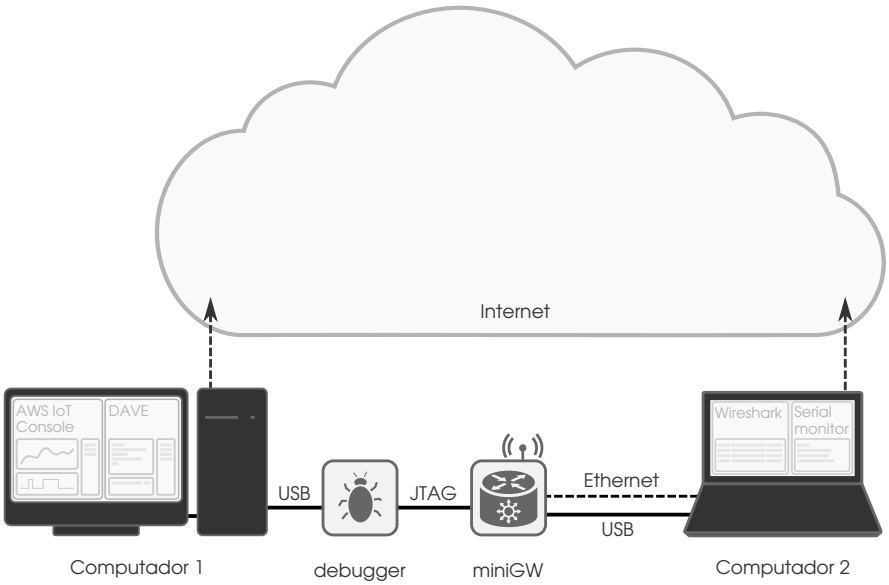


Figura 10.1: Configuración de prueba final.

`aws_iot_config.h`. En concreto, aquí se muestra la información sobre el *endpoint*, con la que deberá definirse el símbolo `AWS_IOT_MQTT_HOST`.

El siguiente paso será crear un certificado. Es imprescindible que al crear el certificado se descargue la clave privada del mismo, ya que al cerrar el diálogo de descarga la clave privada será eliminada de la plataforma. También tendremos que descargar el certificado raíz (*CA root*), disponible desde la misma pestaña.

A continuación, tendremos que asociar este certificado con el dispositivo y con una política de acceso. Como se trata de un test, definiremos una política de acceso bastante laxa, en la que el dispositivo podrá interactuar libremente con todos los elementos IoT de nuestra cuenta de AWS. Para ello, añadiremos a la política la acción `iot:*`. Llegados a este punto, habremos terminado de configurar la plataforma.

Verificación de la conexión

Al igual que hicimos con mbedTLS, usaremos los programas de ejemplo del Device SDK de AWS IoT para comprobar que el funcionamiento es el esperado. En concreto, usaremos los ejemplos `subscribe_publish_sample` y `shadow_sample`. También aquí tendremos que modificar el ejemplo para que su código se ejecute dentro de una tarea de RTOS. En este caso, los pasos seguidos por el firmware en su ejecución son los siguientes:

1. Inicialización del sistema, a través de la inicialización de las DAVE™ APPs mediante la función `DAVE_Init()`. Se inicializan tanto módulos hardware y componentes software, como las variables de CMSIS-RTOS.

2. Configuración de mbedTLS para usar el RTC.

3. Creación de una tarea de una única ejecución, que llamará a la función `shadow_sample()`.

4. Llamada a la instrucción de inicialización del núcleo de CMSIS-RTOS `osKernelStart()`.

La ejecución de este programa tiene como resultado el intercambio de mensajes MQTT entre el miniGW y la Nube. En primer lugar se lleva a cabo la ejecución de un handshake TLS, que hemos podido observar usando Wireshark. Tras el handshake, los datos de aplicación se envían encriptados, por lo que no podemos observar el contenido de los paquetes. En cambio, podemos comprobar que produce una comunicación con la Nube observando los mensajes a través de la consola de AWS IoT. Para ello nos dirigiremos a la pestaña *Activity*, donde podremos ver los mensajes MQTT que producirán las sucesivas actualizaciones de la sombra. Para observar el estado actual del dispositivo, podemos consultar su sombra en la pestaña *Shadow*.

Finalmente, tras este proceso de integración, el miniGW puede comunicarse con los servidores AWS IoT.

10.4. Resolución de problemas

Como no podía ser de otra manera, la integración del Device SDK ha supuesto resolver algunos conflictos. La mayoría de ellos tienen que ver con el uso de memoria, ya que trabajamos en un entorno con recursos muy reducidos. A continuación los exponemos con detalle.

Configuración de CMSIS-RTOS

De nuevo, hemos necesitado más memoria para CMIS-RTOS. Esto se debe a que en esta ocasión estamos verificando un certificado y estamos autenticando al dispositivo de cara al servidor, lo que supone la firma de un *challenge*. Estas operaciones requieren de mucha memoria, por lo que tendremos que incrementar el tamaño de la pila a 17KB. De nuevo, mencionamos que este valor no tiene nada que ver con el heap, solo la pila se está aumentando aquí.

Modificación del mapa de memoria

En mbedTLS, el uso de enteros de precisión múltiple (MPI), también conocidos como BigNum, hace uso de la memoria dinámica[4]. Sin embargo, el heap presente en el sistema con la configuración predefinida es demasiado pequeño para la verificación de certificado con claves de clave asimétrica tan largas. Para que las aplicaciones probadas se ejecuten

[4]También se puede configurar mbedTLS para usar pools de memoria reservados de forma estática. En el caso que nos ocupa, al no tratarse de un sistema crítico, hemos preferido usar el heap gestionado por *newlib* y el *linker script*.

correctamente, es necesario un heap de al menos 35KB. Esta heap será necesario para el proceso de validación del certificado y el proceso de firma en la autenticación mutua .

Por defecto, el heap definido en el proyecto usa la memoria restante en uno de los tres módulos RAM presentes en el XMC (ver figura 7.5). Para expandir la memoria usada por el heap, ha sido necesario modificar el comportamiento de `malloc()`. En concreto, se ha modificado la función `_sbrk()` del fichero `syscalls.c` , para permitirle reservar memoria en cualquiera de los tres bancos disponibles. Este fichero depende de las variables `Heap_BankX_Start` y `Heap_BankX_End`, que han tenido que definirse en el *linker script*. La definición de estas variables ha sido directa, a partir de la definición del primer bloque de heap.

Sockets BSD

Al tratar de compilar el proyecto, hemos encontrado algunos conflictos con las funciones `connect`, `read` y `write`, que estaban definidos en distintos lugares. Para solucionarlo, hemos modificado el fichero `aws_iot_mqtt_client.h` para asegurar que se usa la implementación proporcionada por lwIP.

Parte III

Conclusiones y trabajo futuro

Capítulo 11

Conclusiones y trabajo futuro

Tras la consecución de este trabajo hemos conseguido conectar un dispositivo que forma parte de un sistema *SmartHome* con la nube de Amazon (AWS IoT). Para ello, hemos tenido que integrar el protocolo TLS y el AWS IoT Device SKD en el sistema, que está basado en el microcontrolador XMC4500 (ARM Cortex-M4). La implementación de TLS utilizada ha sido mbedTLS de ARM. Los resultados obtenidos con este se exponen de forma más pormenorizada a continuación.

11.1. Conclusiones

Hemos podido comprobar que el sistema se comunica de forma segura con la Nube. Con los tests explicados en la sección 9.1.6, hemos visto que podemos llevar a cabo conexiones seguras. Estas conexiones se han visualizado usando Wireshark, pudiendo comprobar que el handshake se produce con éxito y que la conexión está realmente encriptada.

Tenemos un sistema donde puede configurarse la seguridad en tiempo de compilación, eligiendo las *ciphersuites* que sean necesarias en cada caso. Para la aplicación estudiada (*SmartHome gateway* comunicándose con AWS IoT) hemos usado las configuraciones que la plataforma recomienda. Sin embargo, la integración realizada puede modificarse fácilmente para conseguir la compatibilidad con otras plataformas.

En cuanto al uso de memoria, hemos comprobado que la aplicación necesita 17KB de stack, y más de 35KB de heap. Hemos podido comprobar que 1MB de RAM (memoria disponible en el XMC4500) es suficiente para llevar a cabo transacciones seguras. Aunque esto supone un incremento en el tiempo de desarrollo, las empresas del sector IoT no deberían descuidar la seguridad de los datos de los usuarios. Solo de esta forma podremos avanzar hacia un mundo realmente interconectado y proporcionar los nuevos servicios que llevamos años divisando.

11.2. Trabajo futuro

Los compañeros de la empresa eesy-innovation deberán integrar la solución propuesta en el sistema final. Además, se proponen las siguientes líneas de trabajo, con las que se podría mejorar el producto.

1. Caracterización de la generación de números aleatorios propuesta. Aunque tenemos indicios para pensar que la distribución de los números aleatorios generados será bastante homogénea, sería conveniente comprobarlo a través de la caracterización del módulo utilizado en varias ejecuciones de los tests.

2. Automatización de la definición de dispositivos virtuales en producción. Será necesario automatizar el proceso de definición de dispositivos virtuales en la plataforma AWS IoT, así como la configuración de los dispositivos físicos con los mismos parámetros.

3. Uso del miniGW como pasarela IoT. En este proyecto, el miniGW ha estado asociado a un único dispositivo virtual. Esto no se ajusta al escenario de producción, en el que los dispositivos virtuales están asociados a los nodos finales de la red, esto es, los sensores y actuadores.

4. Definición de políticas de acceso que se ajusten al sistema. En el caso que nos ocupa, hemos usado una política de acceso extremadamente laxa para la realización del test. Esta configuración nos aseguraba que los problemas vendrían de otro sitio. En cambio, será necesario definir una política de acceso adecuada al escenario de producción.

Bibliografía

[1] Daryl C Plummer, L Fiering, K Dulaney, M McGuire, C Da Rold, A Sarner, W Maurer, F Karamouzis, J Lopez, RA Handler, et al. Top 10 strategic predictions for 2015 and beyond: Digital business is driving 'big change'. Gartner. https://www.gartner.com/doc/2864817?refval=&pcp=mpe, 2014.

[2] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[3] Sabina Jeschke, Christian Brecher, Tobias Meisen, Denis Özdemir, and Tim Eschert. Industrial internet of things and cyber manufacturing systems. In *Industrial Internet of Things*, pages 3–19. Springer, 2017.

[4] Tamas Pflanzner and Attila Kertész. A survey of iot cloud providers. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on*, pages 730–735. IEEE, 2016.

[5] A. Sadeghi, C. Wachsmann, and M. Waidner. Security and privacy challenges in industrial internet of things. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[6] R. Roman, P. Najera, and J. Lopez. Securing the internet of things. *Computer*, 44(9):51–58, Sept 2011.

[7] Andrew S. Tanenbaum. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2001.

[8] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks (5th Edition)*. Pearson, 2010.

[9] Wikipedia contributors. Block cipher mode of operation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=853654843, 2018. Accedido el 10 de septiembre de 2018.

[10] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[11] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[12] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[13] Alfred J Menezes, Tatsuaki Okamoto, and Scott A Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *iEEE Transactions on information Theory*, 39(5):1639–1646, 1993.

[14] National Institute of Standards and Technology (US). Technology Administration. *Secure hash standard*, volume 180. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1993.

[15] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):461–491, 2004.

[16] Steve Heath. *Embedded Systems Design*. Elsevier Ltd., 2002.

[17] ARM Limited. *ARM Security Technology: Building a Secure System using TrustZone Technology*, 2009.

[18] Ed. P. Eronen and Ed. H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, RFC Editor, December 2005.

[19] U. Blumenthal and P. Goel. Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS). RFC 4285, RFC Editor, January 2007.

[20] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008.

[21] IBM Knowledge Cencter. An overview of the SSL or TLS handshake. https://www.ibm.com/support/knowledgecenter/SSFKSJ_7.5.0/com.ibm.mq.sec.doc/q009930_.htm, 2018. Accedido el 12 de septiembre de 2018.

[22] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, RFC Editor, 2006.

[23] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, RFC Editor, 2012.

[24] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. RFC 2459, RFC Editor, January 1999.

[25] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, RFC Editor, May 2008.

[26] Microsoft Docs. Digital Certificates. https://technet.microsoft.com/en-us/library/cc962029.aspx, 2012. Accedido el 11 de septiembre de 2018.

[27] Wikimedia Commons. File:chain of trust.svg — wikimedia commons, the free media repository, 2017. Accedido el 12 de septiembre de 2018.

[28] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, RFC Editor, October 2006.

[29] What Is AWS IoT? https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html. Accedido el 30 de agosto de 2018.

[30] Robert McCauley. Using Alexa Skills Kit and AWS IoT to Voice Control Connected Devices. https://developer.amazon.com/blogs/post/Tx3828JHC7O9GZ9/Using-Alexa-Skills-Kit-and-AWS-IoT-to-Voice-Control-Connected-Devices, May 2016. Accedido el 30 de agosto de 2018.

[31] International Organization for Standarization (ISO). Iso/iec 20922:2016 information technology – message queuing telemetry transport (mqtt) v3.1.1. https://www.iso.org/standard/69466.html, 2016.

[32] Shadow Document Syntax. https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-document-syntax.html. Accedido el 30 de agosto de 2018.

[33] Shadow MQTT Topics. https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-mqtt.html. Accedido el 30 de agosto de 2018.

[34] Device Shadow RESTful API. https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-rest-api.html. Accedido el 30 de agosto de 2018.

[35] AWS Free Tier. https://aws.amazon.com/free/. Accedido el 30 de agosto de 2018.

[36] Cloud Computing for Education - Amazon Web Services. https://aws.amazon.com/education/. Accedido el 30 de agosto de 2018.

[37] Infineon Technologies AG. *DAVE™ Integrated Development Environment*, March 2016.

[38] Infineon Technologies AG. *XMC4500 Family: Microcontroller Series for Industrial Applications*, December 2017. Rev. 1.5.

[39] Infineon Technologies AG. *XMC4500 Reference Manual*, July 2016. Rev. 1.6.

[40] Infineon Technologies AG. *TDA5340 SmartLEWIS™ TRX: High Sensitivity Multi-Channel Transceiver*, June 2012. Rev. 1.2.

[41] Wikipedia contributors. Comparison of tls implementations — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Comparison_of_TLS_implementations&oldid=851932757, 2018. Accedido el 10 de agosto de 2018.

[42] Security and Identity for AWS IoT. https://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html. Accedido el 3 de agosto de 2018.

[43] Compilation and Installation — OpenSSL. https://wiki.openssl.org/index.php/Compilation_and_Installation#Configuration. Accedido el 10 de agosto de 2018.

[44] L. Law and J. Solinas. Suite B Cryptographic Suites for IPsec. RFC 6379, RFC Editor, October 2011.

[45] mbed TLS Knowledge Base. https://tls.mbed.org/kb. Accedido el 13 de agosto de 2018.

[46] mbed TLS API Documentation. https://tls.mbed.org/api/. Accedido el 13 de agosto de 2018.

[47] mbed TLS Core Features. https://tls.mbed.org/core-features. Accedido el 12 de mayo de 2018.

[48] config.h File Reference — mbed TLS API documentation. https://tls.mbed.org/api/config_8h.html. Accedido el 12 de mayo de 2018.

[49] Manuel Pégourié-Gonnard. How do I configure mbed TLS. https://tls.mbed.org/kb/compiling-and-building/how-do-i-configure-mbedtls, July 2015. Accedido el 12 de mayo de 2018.

[50] Paul Bakker. Reduce mbed TLS memory and storage footprint. https://tls.mbed.org/kb/how-to/reduce-mbedtls-memory-and-storage-footprint, February 2016. Accedido el 12 de mayo de 2018.

[51] Paul Bakker. How do I build and compile mbed TLS. https://tls.mbed.org/kb/compiling-and-building/how-do-i-build-compile-mbedtls, September 2015. Accedido el 13 de mayo de 2018.

[52] Manuel Pégourié-Gonnard. Porting mbed TLS to a new environment or OS. https://tls.mbed.org/kb/how-to/how-do-i-port-mbed-tls-to-a-new-environment-OS, April 2017. Accedido el 13 de mayo de 2018.

[53] Simon Butcher. Mbed TLS tests guidelines — mbedTLS Knowledge Base. https://tls.mbed.org/kb/development/test_suites, dec 2017. Accedido el 17 de agosto de 2018.

[54] Paul Bakker. mbed TLS automated testing and Quality Assurance — mbedTLS Knowledge Base. https://tls.mbed.org/kb/generic/what-tests-and-checks-are-run-for-mbedtls, feb 2016. Accedido el 17 de agosto de 2018.

[55] Horace G. Hodges David A.; Jackson. *Analysis and Design of Digital Integrated Circuits*. McGraw-Hill Education (ISE Editions), 1988.

[56] Adel S. Sedra and Kenneth C. Smith. *Circuitos microelectrónicos 4a edición*. Oxford University Press, 1999.

[57] Paul Bakker. How to add entropy sources to the entropy pool — mbed TLS Knowledge Base. https://tls.mbed.org/kb/how-to/add-entropy-sources-to-entropy-pool, jul 2015. Accedido el 16 de agosto de 2018.

[58] AWS IoT Device SDK for Embedded C. https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html#iot-c-sdk. Accedido el 13 de agosto de 2018.

[59] AWS IoT Device SDK for Embedded C — Porting Guide. https://github.com/aws/aws-iot-device-sdk-embedded-C/blob/master/PortingGuide.md. Accedido el 13 de agosto de 2018.

[60] S. Josefsson and S. Leonard. Textual Encodings of PKIX, PKCS, and CMS Structures. RFC 7468, RFC Editor, April 2015.