Implementación de una librería Java para comunicación con tarjetas Mifare DESFire

Francisco Rodríguez Algarra

Projecte Final de Carrera
Enginyeria Superior en Telecomunicacions

Director:
Dr. Francisco José Rico Novella
Departament d'Enginyeria Telemàtica
Universitat Politècnica de Catalunya

Resumen

Las tecnologías de comunicación de corto alcance están recibiendo una atención cada vez mayor gracias, principalmente, a las ventajas que ofrece a nivel de facilidad y comodidad de uso. El número de dispositivos móviles que ofrecen conectividad NFC y de tarjetas inteligentes con comunicación sin contacto de los que disponemos a día de hoy es una muestra clara de que existen grandes oportunidades para el desarrollo de aplicaciones que aprovechen esta situación para ofrecer servicios útiles e imaginativos a los usuarios finales. Las herramientas de las que disponen actualmente los desarrolladores, sin embargo, no facilitan esta tarea. Por este motivo, el proyecto que se presenta en esta memoria se ha elaborado con el propósito principal de ofrecer una librería de programación Java que permita el desarrollo de aplicaciones con comunicación con tarjetas Mifare DESFire a un nivel de abstracción elevado y de la forma más simple posible.

Para lograr este objetivo hemos realizado un análisis de las principales características de las tecnologías de corto alcance en general, y de las tarjetas Mifare DESFire en particular. Este modelo de tarjetas sin contacto es uno de los más utilizados en los últimos años, ya que ofrece funcionalidades de seguridad mediante algoritmos criptográficos estándar y un amplio abanico de comandos nativos a través de su interfaz de programación. La implementación a un nivel elevado de abstracción de estos comandos supone el núcleo principal de la librería desarrollada. Para lograr que el desarrollo de aplicaciones resulte lo más intuitivo posible se incluyen, además, múltiples clases destinadas tanto a aislar el código implementado por los desarrolladores del interfaz de comunicación de bajo nivel empleado como a abstraer los parámetros de envío y recepción, facilitando así su manejo e interpretación.

Las posibilidades que ofrece el uso de la librería quedan reflejadas en la aplicación de demostración que se ha desarrollado. En concreto, se presenta un ejemplo en forma de sistema de compra y gestión de entradas a eventos con pago mediante monedero electrónico. De este modo se muestra la flexibilidad y seguridad que permiten las tarjetas Mifare DESFire, y cómo pueden aprovecharse mediante el uso de la librería.

Agradecimientos

Este proyecto no habría sido posible sin la paciencia inquebrantable de mi supervisor, el Dr. Francisco José Rico Novella. Desde el momento en el que iniciamos este proyecto, hace ya demasiados años, hasta el día de hoy he sentido siempre su aliento, sus ánimos, su buen humor, en especial cuando he sentido flaquear mis fuerzas. Ante cualquier contratiempo, ante cualquier desvío en el camino, él me ha rescatado una y otra vez sin perder la sonrisa. Sin él no habría sido capaz de llegar hasta aquí. Nunca podré agradecérselo cómo merece.

El apoyo y el amor incondicionales que he recibido de mis padres no tienen igual. Soy muy consciente de todo lo que les he hecho sufrir, del miedo que han tenido a que no llegara este día. Pero nunca han desfallecido. Siempre han estado a mi lado y me han dado mucho más de lo que merezco. Ellos son el alma de este proyecto.

No quisiera olvidarme de todos aquellos, familiares y amigos, que siempre han confiado en mí. Sus consejos, sus ánimos, sus tirones de orejas cuando eran necesarios, han sido fundamentales. Gran parte de este proyecto es suyo.

Índice general

Resumen I

Agradecimientos III

1. Introducción 2
- 1.1. Objetivos del Proyecto . . . 3
- 1.2. Contenido de la Memoria . . . 3

2. Tecnologías de Comunicación de Corto Alcance 4
- 2.1. RFID . . . 4
  - 2.1.1. Descripción Tecnológica . . . 5
  - 2.1.2. Estandarización . . . 9
  - 2.1.3. Aplicaciones . . . 9
- 2.2. Smart Cards y Contactless Smart Cards . . . 11
  - 2.2.1. Descripción Tecnológica . . . 12
  - 2.2.2. Estandarización . . . 13
  - 2.2.3. Aplicaciones . . . 15
- 2.3. NFC . . . 16
  - 2.3.1. Descripción Tecnológica . . . 17
  - 2.3.2. Estandarización . . . 18
  - 2.3.3. Aplicaciones . . . 19
- 2.4. Conclusión . . . 20

3. Mifare DESFire y Mifare DESFire EV1 21
- 3.1. Características Técnicas y Funcionales . . . 22
  - 3.1.1. Interfaz de RF y Transferencia de Datos . . . 22
  - 3.1.2. Organización de la Memoria . . . 23
- 3.2. Comandos de Operación . . . 24
  - 3.2.1. Comandos de Seguridad . . . 25
  - 3.2.2. Comandos a Nivel de Tarjeta . . . 25

3.2.3. Comandos a Nivel de Aplicación . . . 26
3.2.4. Comandos de Manipulación de Datos . . . 26
3.3. Mecanismos de Seguridad . . . 27
3.3.1. Identificador Único (UID) . . . 27
3.3.2. Autentificación y Clave de Sesión . . . 28
3.3.3. Derechos de Acceso y Modos de Comunicación . . . 29
3.3.4. Manejo de Claves . . . 35
3.4. Conclusión . . . 39

4. Infraestructura empleada **40**
4.1. Sistema Operativo . . . 41
4.2. Middleware . . . 42
4.2.1. Especificaciones del PC/SC Workgroup . . . 42
4.2.2. `pcsc-lite` . . . 46
4.2.3. Interfaz Java . . . 47
4.3. Hardware . . . 50
4.4. Conclusión . . . 54

5. Implementación de la Librería **55**
5.1. Interfaz de nivel intermedio `ComManager` . . . 55
5.2. Comunicación con tarjetas Mifare DESFire . . . 59
5.3. Funcionalidades auxiliares . . . 64
5.3.1. Utilidades de manejo de cadenas de bytes . . . 64
5.3.2. Utilidades de seguridad . . . 65
5.4. Conclusión . . . 66

6. Desarrollo de aplicaciones **67**
6.1. Anatomía básica de una aplicación . . . 67
6.2. Implementando un servicio “real” . . . 68
6.2.1. La aplicación de demostración en funcionamiento . . . 69
6.2.2. Detalles de implementación . . . 81
6.3. Conclusión . . . 88

7. Conclusiones y líneas futuras **90**

A. Configuración de la Infraestructura **94**

B. Configuración de las Bases de Datos **96**
B.1. Base de Datos Servicios Bancarios . . . 96
B.2. Base de Datos Servicios de Venta de Entradas . . . 98

C. Documentación de la clase `DFCard` 104
C.1. Package dflibrary.library 104
C.1.1. Class DFCard 104

Referencias 157

Índice de tablas

4.1. Software empleado en la elaboración del proyecto 40

Índice de figuras

2.1. Alimentación de un transpondedor RFID a través de acoplamiento inductivo de la energía del campo electromanético generado por el lector 6
2.2. Arquitectura habitual de una Contactless Smart Card 13
2.3. Esquema de la construcción física de una Contactless Smart Card 14

3.1. Esquema simplificado del funcionamiento del interfaz de RF en tarjetas MF3ICDXX 23
3.2. Estructura del Identificador Único de tarjeta (UID) 27
3.3. Esquema simplificado del proceso de Autentificación Mutua en 3 Pasos empleado en las tarjetas Mifare DESFire y Mifare DESFire EV1 29
3.4. Interpretación de la codificación hexadecimal de los derechos de acceso 31
3.5. Ejemplo de asignación de derechos de acceso en una aplicación con tres claves y tres ficheros 32
3.6. Esquema de enlazamiento CBC de bloques de cifrado en el lector en operaciones de envío y recepción de datos con tarjetas Mifare DESFire 35
3.7. Construcción estándar códigos CMAC a partir de un mensaje M de longitud n en dos casos posibles de "padding" 36
3.8. Diagrama de bloques simplificado del proceso recomendado para la diversificación de claves usando CMAC 38

4.1. Arquitectura propuesta en las especificaciones del PC/SC Workgroup, coon la parte de la documentación correspondiente a cada componente 45
4.2. Ejemplo de reconocimiento de dispositivo lector con `pcsc_scan` 51
4.3. Dispositivos lectores empleados durante la elaboración del proyecto 52

6.1. Selección de lector 70
6.2. Transiciones entre *apps* en la aplicación de demostración 72
6.3. Ventanas de selección de aplicaciones y subaplicaciones 73
6.4. Operaciones disponibles en la aplicación de monedero electrónico 74
6.5. Mensajes de finalización de operación 76
6.6. Operaciones disponibles en la aplicación de gestión de entradas 78

6.7. *App* de restauración de valores de fábrica 80
6.8. Transiciones entre estados en una *app* genérica 83

Índice de Listados de Código

5.1. Ejemplo de uso de una implementación del interfaz `ComManager` 58
5.2. Ejemplo de uso de una instancia de la clase `DFCard` para el envío de un comando DESFire 59
5.3. Respuesta obtenida al ejecutar el comando `df.GetVersion()` en una tarjeta Mifare DESFire EV1 61
5.4. Respuesta obtenida al ejecutar el comando `df.getVersion()` en una tarjeta Mifare DESFire EV1 61
5.5. Recuperación de la clase de objeto original del atributo `field` de una respuesta a través de una operación de "casting" 62
5.6. Recuperación de la clase de objeto original del atributo `field` de una respuesta a través de su método específico 62
6.1. Extracto del código de la clase `WTModel` correspondiente al método `purchase` . . . 85
6.2. Comparación de métodos con comunicación local (`writeTicketRecord`) y remota (`payment`) con una tarjeta Mifare DESfire, incluidos en la clase `WTModel` 87
B.1. Estructura de la Base de Datos de Servicios Bancarios 96
B.2. Ejemplo de población de la Base de Datos de Servicios Bancarios 97
B.3. Estructura de la Base de Datos de Servicios de Venta de Entradas 98
B.4. Ejemplo de población de las tablas de eventos y usuarios de la Base de Datos de Servicios de Venta de Entradas 99
B.5. Ejemplo de población de la tabla de tickets de la Base de Datos de Servicios Bancarios 99

1 Introducción

Vivimos en una era de inmediatez. Sometidos a la exigencia de aprovechar el tiempo al máximo, abrazamos encantados todo aquello que permita ganarle un segundo al reloj. Las tecnologías de información y comunicaciones, que han desdibujado los límites entre vida profesional y particular, nos están volviendo cada vez más impacientes. Los computadores nos proporcionaron análisis complejos en un parpadeo; los teléfonos móviles nos permitieron comunicarnos a cualquier hora y cualquier lugar; Internet nos ofreció respuestas inmediatas a prácticamente cualquier pregunta que pudiéramos imaginar; y, recientemente, los teléfonos inteligentes nos dan todo esto y mucho más desde la palma de nuestra mano.

Es en este sentido en el que las tecnologías de comunicación sin contacto están empezando a ganar relevancia. La promesa de acelerar los procesos de pago probablemente sea la razón principal que ha llevado a que la mayoría de nosotros contemos actualmente con tarjetas de crédito/débito con facultades *contactless*[1]. El Gobierno de España también ha decidido apostar por estas tecnologías para incrementar la comodidad en la realización de trámites introduciendo capacidades *contactless* en el denominado DNI electrónico 3.0[2]. Estos hechos nos llevan a la convicción de que la aceptación por parte de los usuarios de estas tecnologías es cada vez mayor y seguirá aumentando en los próximos años.

El número y diversidad de las aplicaciones que aprovechan las ventajas de las tarjetas sin contacto es, sin embargo, muy limitado. Una de las pocas compañías que han apostado por ofrecer servicios más allá de las tarjetas institucionales es Stratos[3], y estamos convencidos de que la tecnología tiene potencial suficiente como para que más desarrolladores, investigadores y emprendedores den el paso y sigan su ejemplo. Pero para lograrlo creemos que es imprescindible disponer de un entorno que facilite la implementación de todo aquel servicio que un desarrollador pueda imaginar, permitiéndole centrarse únicamente en la lógica de la aplicación sin preocuparse de los entresijos de la tecnología subyacente. Con esta intención hemos elaborado el presente proyecto.

[1] https://portal.lacaixa.es/tarjetas/contactless/ventajas_es.html
[2] http://www.policia.es/prensa/20150112_1.html
[3] https://stratoscard.com

1.1. Objetivos del Proyecto

El objetivo principal del proyecto que se reporta en este documento es la implementación de una librería escrita en Java que permita el desarrollo de aplicaciones de alto nivel mediante la comunicación con tarjetas Mifare DESFire y Mifare DESFire EV1. La librería se ha diseñado para ser lo más sencilla, funcional y escalable posible de forma que pueda ser empleada en el futuro para la construcción de aplicaciones que saquen provecho del potencial de las mencionadas tarjetas.

En este sentido, de forma auxiliar a la implementación de la librería, el objetivo secundario del proyecto es la creación de una aplicación concreta que, además de servir como ejemplo de uso de la librería, permita comprobar las características de seguridad de las tarjetas Mifare DESFire y Mifare DESFire EV1, así como mostrar su soporte multiaplicación. Finalmente, dicha aplicación incluye funcionalidades de acceso remoto a las tarjetas, ya que a día de hoy es un ámbito en el que consideramos que no se está explotando suficiente todo el potencial que ofrecen estas tecnologías.

1.2. Contenido de la Memoria

La presente memoria se organiza principalmente en dos grandes secciones. La primera de ellas, que comprende los Capítulos 2 y 3, está especialmente dedicada a la descripción de los antecedentes tecnológicos necesarios para la elaboración del proyecto. En ellos se explican brevemente las tecnologías de comunicación por proximidad más destacadas, y se entra en mayor detalle al abordar las características de las tarjetas Mifare DESFire y Miare DESFire EV1, que se emplearán más adelante. Por otro lado, la segunda parte de la memoria tiene como objetivo describir el trabajo efectivamente realizado durante la elaboración de este proyecto, empezando por los principales rasgos de la infraestructura que ha sido empleada, haciendo especial hincapié en el software y el hardware necesarios para la comunicación a medio y bajo nivel con las tarjetas en el Capítulo 4. El Capítulo 5 explica las características más relevantes de la librería implementada, mientras que el Capítulo 6 describe el desarrollo de la aplicación demostrativa que ha sido creada, así como las funcionalidades que ofrece. Finalmente, se comentan las conclusiones que pueden ser extraídas de la elaboración del proyecto y se sugieren algunas líneas futuras de trabajo en el Capítulo 7.

2 | Tecnologías de Comunicación de Corto Alcance

Entre las tecnologías de comunicación de corto alcance, o también denominadas de proximidad, la que se encuentra con un mayor auge actualmente es la tecnología Near Field Communication, más conocida por sus siglas NFC. Surge de la evolución de los dispositivos de identificación por radiofrecuencia (RFID), cuya explicación abordaremos primero para establecer el contexto en el que NFC fue desarrollado. Tras la descripción de los sistemas RFID, realizaremos una breve explicación de las denominadas tarjetas inteligentes ("Smart Cards"), cuyos fundamentos han sido combinados con los de las comunicaciones de corto alcance para el desarrollo de las "Contactless Smart Cards", que constituyen el verdadero núcleo del presente proyecto. Posteriormente, se hará un rápido recorrido alrededor de las principales características de NFC ya que, aunque es una tecnología que engloba mucho más allá del alcance del proyecto, es imprescindible conocerla para comprobar las verdaderas posibilidades del trabajo que se ha desarrollado.

2.1. RFID

El acrónimo RFID hace referencia a las siglas de la voz inglesa "Radio-Frequency IDentification". Se trata de un método de identificación y autentificación de objetos y personas sin necesidad de disponer de ningún contacto físico ni óptico, usando circuitos integrados denominados etiquetas, tags o transpondedores RFID, que incluyen una antena y que son interrogados por lectores a través de un canal de radiofrecuencia. Esta tecnología es, sin ninguna duda, una de las más fructíferas de los últimos años debido a la gran variedad de campos en los que puede aplicarse y a las enormes mejoras en cuanto a tamaño y coste que se han producido en sus elementos. Sin embargo, los pilares fundamentales sobre los que se sustenta esta tecnología no son excesivamente novedosos ya que parten de ideas surgidas en la primera mitad del siglo XX.

La opinión general acerca del origen de la tecnología de identificación por radiofrecuencia es que tiene sus raíces en el uso del radar durante la II Guerra Mundial para la detección de aviones a kilómetros de distancia. Sin embargo, dicho sistema no permitía su identificación, por lo que no podía determinarse si se trataba de aviones amigos o enemigos. En este sentido, la aviación

alemana descubrió que un balanceo del avión al aproximarse a su propia base alteraba la señal de radio que se devolvía al radar, permitiendo así distinguir sus propios aparatos de los enemigos[1]. Podríamos considerar este como el primer sistema pasivo de identificación por radiofrecuencia de la historia. En el Reino Unido, por su parte, se desarrolló un sistema de identificación de "amigo o enemigo" (IFF, en sus siglas anglosajonas) más sofisticado a base de situar un transmisor en cada uno de sus aviones, que emitía una señal que identificaba la aeronave como amiga cuando recibía señales de las estaciones de radar de tierra. Este sistema es la base de la tecnología RFID actual.

Los sistemas de radar y de comunicación por radiofrecuencia avanzaron en las décadas de los 50 y los 60 para tratar de identificar objetos remotamente. Muchas compañías pronto empezaron a utilizar en sus tiendas sistemas antirrobo que se basaban en el uso de ondas de radio para determinar si un objeto había sido pagado o no. Las etiquetas de vigilancia electrónica de artículos, que hoy en día siguen siendo utilizadas, almacenan solamente un bit de información que toma el valor 1 cuando el producto no ha sido vendido, activando los lectores situados en las puertas si alguien trata de salir de la tienda con él, y que es modificado a 0 cuando se paga en caja, pudiendo ser sacado de la tienda sin que los lectores hagan saltar la alarma.

Durante los años 70 y 80 se desarrollaron importantes avances en la tecnología RFID, incluyendo etiquetas activas con memoria reescribible y la técnica del "*backscatter*" para el uso de etiquetas pasivas a distancias más grandes[2], así como nuevas aplicaciones como sistemas de control de acceso a edificios y de paso preferente en peajes. En los últimos años se han desarrollado mejoras en los procesos de fabricación que han permitido una mayor miniaturización y reducción de costes que ha tenido un gran impacto tanto en el precio final como en la capacidad y rango de emisión y recepción. Esto ha contribuido a extender su uso en ámbitos tanto domésticos como, incluso, de seguridad nacional.

2.1.1. Descripción Tecnológica

El modo de funcionamiento de los sistemas RFID es bastante simple y se basa en lo que se ha denominado acoplamiento inductivo, que se encuentra esquematizado en la Fig. 2.1. La comunicación que se produce es asimétrica en el sentido que una parte, llamada lector o interrogador, toma el rol de transmisor mientras que la otra, la etiqueta, se limita a responder. La antena del lector crea un campo magnético y, cuando la etiqueta se encuentra en el espacio de actuación de dicho campo, reacciona de forma inmediata, enviando al lector la información que tiene almacenada. La señal de respuesta es captada por el lector, que descodifica los datos y los transfiere, en formato digital, a una aplicación específicamente creada para tal propósito. En gran medida esta arquitectura es la causa del éxito de los sistemas RFID, ya que permite que un

[1] http://www.rfidjournal.com/articles/view?1338

[2] Si la etiqueta RFID modula la señal recibida y la reenvía es posible alcanzar distancias de unos cuantos metros únicamente con la energía que se ha recibido del lector [11]

pequeño número de lectores puedan ser usados con una elevada cantidad de etiquetas de escasa complejidad, lo que logra una alta escalabilidad manteniendo el coste total muy bajo.

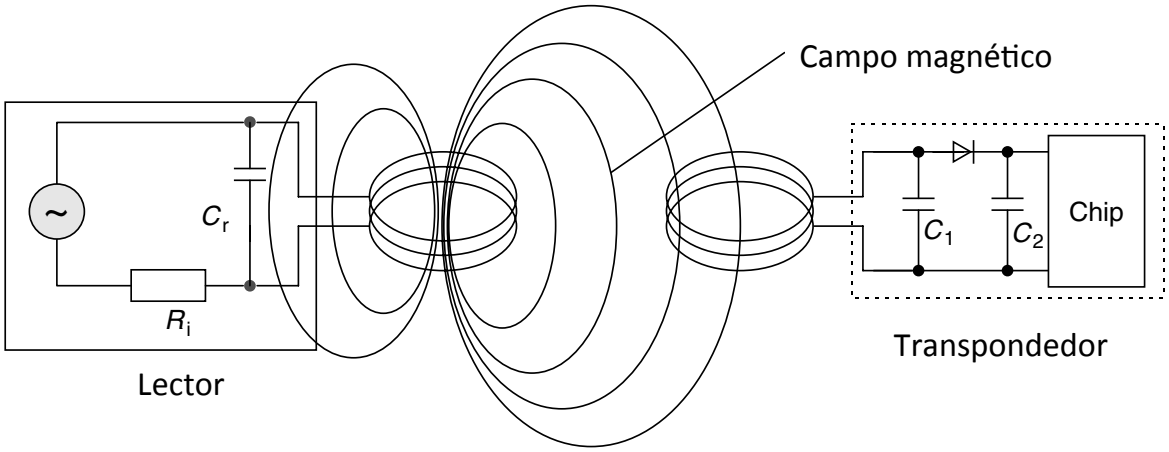


Figura 2.1: Alimentación de un transpondedor (etiqueta) RFID a través de acoplamiento inductivo de la energía del campo electromanético generado por el lector (Figura adaptada de Finkenzeller [11])

Por tanto, los elementos básicos de un sistema RFID son los siguentes:

- Etiqueta RFID
- Lector RFID
- Opcionalmente, un subsistema de procesamiento de datos

Etiquetas RFID

Las etiquetas RFID son el componente principal de cualquier sistema de radiofrecuencia y son las encargadas de emitir una señal de respuesta a otra previamente enviada por un lector. También reciben el nombre de transpondedor ("*transponder*"), término que se construye a través de la unión de las palabras inglesas *Transmitter* y *Responder*, y, en muchas ocasiones se les denomina por el vocablo anglosajón "*tag*". Se trata de elementos que pueden ser incorporados a un producto, un animal o, incluso, a una persona, para lograr su identificación mediante el uso de ondas de radio.

Los elementos esenciales de las que están compuestas las etiquetas RFID son los siguientes:

Antena Encargada de recibir las señales enviadas por el lector, responder a dichas señales y, en algunos tipos de etiquetas, suministrar la energía necesaria para su funcionamiento.

Circuito integrado Almacena un número de identificación (UID) y contiene la lógica de operación de la etiqueta para una correcta comunicación con el lector. Este chip se compone de: radio receptor, radio modulador, control lógico y sistema de energía. Algunas etiquetas RFID, sin embargo, permiten la identificación sin la presencia de este chip integrado. Estas "*Chipless RFID tags*" tienen, por tanto, un coste de fabricación todavía menor.

Memoria Se compone de una parte no volátil denominada ROM, que contiene instrucciones básicas para su funcionamiento, y una memoria RAM para almacenar datos durante la comunicación con el lector.

Otros componentes electrónicos Tratan la señal de la antena y se encargan de procesar los datos recibidos.

Protección El sistema eléctrico de la etiqueta es protegido mediante una envoltura papel o polímero para evitar daños

Las etiquetas RFID pueden ser clasificadaes en tres categorías diferentes dependiendo del origen de la energía que utilizan para responder una petición del lector:

Pasivas Las etiquetas RFID pasivas se caracterizan por no incorporar una fuente de alimentación propia, lo que les obliga a extraer la potencia de una fuente externa, en este caso generando una corriente eléctrica a partir de la señal de radiofrecuencia emitida por el lector cuando se encuentra escaneando dicha etiqueta. Cuando el receptor se dispone a realizar la lectura del tag envía energía que activa a la etiqueta para que esta pueda transmitir una respuesta, por lo que se conoce como señal de excitación. Este tipo de etiquetas poseen unos recursos muy limitados, por lo que pueden almacenar solamente unos pocos KBytes de información y normalmente son de solo lectura. Su campo de cobertura es también muy reducido, con distancias de lectura que varían entre unos 10 milímetros hasta cerca de 6 metros, dependiendo de sus características concretas en cuanto a tamaño de la antena, de la potencia de la señal recibida y la frecuencia de operación. No obstante, su bajo coste hace que sean los tags más utilizados.

Activas Las etiquetas RFID activas llevan incorporada su propia fuente de alimentación en forma de bateria u otra conexión eléctrica para activar los circuitos internos y enviar la señal a través de la antena. Esto provoca que dispongan de rangos de cobertura mayores y memorias más grandes que las etiquetas pasivas, así como capacidad para poder almacenar información adicional enviada por el interrogador. Muchas etiquetas activas tienen rangos de cobertura que van entre los diez y los cien metros, y pueden ser tanto de solo lectura como de lectura-escritura. A cambio tienen un coste por chip y un tamaño mayores que las etiquetas pasivas, lo que impide que puedan ser utilizadas en productos de gran consumo. Acostumbran a reservarse para otros usos, como la identificación de personas o de productos de gran valor.

Semi-pasivas Las etiquetas RFID semipasivas, a veces denominadas también semiactivas, son muy similares a las pasivas, salvo que además incorporan una pequeña batería que se utiliza para activar la circuitería interna del chip. Sin embargo, utilizan la energía recogida de las ondas de radio del lector para generar la señal de respuesta, de forma análoga a lo que sucede en las etiquetas pasivas, aunque dicha respuesta se produce de forma más rápida en comparación con estas.

Lectores RFID

Un lector RFID, también denominado en ocasiones interrogador, es un dispositivo encargado de la comunicación con una etiqueta RFID, emitiendo señales de radio a una frecuencia predeterminada con el objetivo de obtener de ella una respuesta. Cuando dicha respuesta es obtenida, convierte la señal de radiofrecuencia en un código numérico que puede ser transmitido a otros dispositivos o sistemas.

Los lectores RFID disponen de los siguientes tres componentes principales:

- Una o más antenas, que pueden ser integradas o externas
- El interfaz de radio, que es el responsable de la modulación, demodulación, transmisión y recepción de la comunicación. Debido a la alta sensibilidad que se requiere en muchas ocasiones, es habitual que los lectores RFID dispongan de caminos diferentes para la transmisión y la recepción.
- El sistema de control, que consiste en un microcontrolador, una o más interfaces de red y, en determinados casos, módulos adicionales específicos para determinadas tareas, como, por ejemplo, co-procesadores criptográficos. El papel de este sistema de control es dirigir la comunicación con la etiqueta, así como interactuar con las aplicaciones de la capa de cliente.

Dependiendo el tipo de bobina de la que disponen en su antena, podemos distinguir dos tipologías de lectores. En los más simples, una misma bobina sirve para transmitir la energía necesaria para alimentar a los tags pasivos y para enviar los datos. Su complejidad de construcción y su coste son menores, pero poseen un alcance limitado. Por otro lado, los sistemas interrogadores de dos bobinas se caracterizan por emplear una de ellas para transmitir energía y la otra diferente para la información. A causa de sus mejores prestaciones, este tipo de lectores son, obviamente, bastante más caros.

Siguiendo otro criterio, podemos clasificar a los lectores en función de su portabilidad, distinguiendo los lectores móviles de los fijos:

Lectores móviles Se trata de dispositivos que pueden ser transportados de un lugar a otro gracias a su pequeño tamaño y a la posibilidad de intercambiar información con otros dispositivos a través de una conexión inalámbrica.

Lectores fijos También denominados "de infraestructura", se instalan de manera permanente dentro del área de acción de los distintos tags a leer o escribir. Estos permiten detectar las señales de los transmisores RFID dentro de un radio de alcance mayor del que permiten los lectores móviles. Estos lectores se utilizan en sistemas de detección y seguimiento de personas y animales en tiempo real.

Cuando un lector recibe una petición por parte de una aplicación cliente, se inicia un proceso que consta habitualmente de tres etapas. En la primera, el lector difunde un mensaje a todas las etiquetas que se encuentran dentro de su rango y recibe sus respuestas construyendo lo que se denomina un "inventario" de identificadores. Entre ellos, selecciona uno para establecer una comunicación punto a punto y, finalmente, intercambia información con la etiqueta seleccionada. Debido a la gran cantidad de tags que potencialmente pueden encontrarse dentro del radio de alcance del lector, normalmente resulta conveniente implementar medidas anticolisión que eviten la confusión entre los datos enviados en la comunicación punto a punto con aquellas respuestas tardías que puedan ser recibidas con simultaneidad.

2.1.2. Estandarización

Principalmente existen dos organismos que publican regulaciones para la tecnología RFID y que, de algun modo, compiten entre ellos. Por una parte, EPCglobal publica las normas relativas a las etiquetas EPC[3], mientras que ISO ha desarrollado una serie de estándares relacionados con diversas aplicaciones de RFID. La familia de normas más estrechamente relacionada con esta tecnología es la serie 18000, que trata acerca de la gestión de objetos a través de RFID. Asimismo, la norma 10374 describe un sistema de identificación automática de contenedores a través de transpondedores (activos) de microondas. Las normas ISO 11784, 11785 y 14223 tratan de la identificación de animales a través de sistemas RFID. Finalmente, las normas ISO 10536, 14443 y 15693 relativas a las tarjetas sin contacto son también altamente relevantes.

Uno de los puntos en los que existe una mayor desregulación y donde es preciso un mayor trabajo de estandarización es en las frecuencias de trabajo de los sistemas. Muchos paises tienen bandas reservadas que sólo pueden ser utilizadas bajo licencia, sobre todo en el rango de UHF, lo que impide una generalización de su uso para esta tecnología. Sin embargo, la banda de alta frecuencia situada entorno a los 13.56 MHz es la más ampliamente utilizada y a la que tienden a converger las distintas normas.

2.1.3. Aplicaciones

En los últimos años el uso de sistemas RFID se ha extendido enormemente, abarcando un número incontable de posibles aplicaciones. Comentarlas aquí todas resultaría una tarea inabarcable, por lo que a continuación tan sólo describiremos algunas de las más relevantes.

[3] http://www.gs1.org/sites/default/files/docs/tds/TDS_1_8_Standard_20140203.pdf

SCM

La gestión de la cadena de suministro, más conocida por sus siglas anglosajonas SCM ("*Supply Chain Management*"), ha sido uno de los motores que han conducido a la tecnología RFID al nivel de popularidad del que goza actualmente. Los sistemas de SCM se encargan de gestionar el movimiento de bienes entre organizaciones a través de su cadena de suministro, desde la obtención de las materias primas hasta ser puestas a disposición del cliente final como productos comerciales terminados. Los requisitos a satisfacer en la cadena de cada bien son diferentes, pero todas comparten la necesidad de mantener un proceso simple, uniforme, veloz y seguro. Para ello es fundamental disponer de un sistema que permita un seguimiento inequívoco de los productos que sea común en sus diferentes etapas.

A inicios del presente siglo, las etiquetas RFID habían alcanzado un nivel de coste-efectividad como métodos de identificación automática de productos que los hacía adecuados para su uso en pruebas piloto para SCM. Los beneficios que dicha práctica suponia fueron evidentes de inmediato, por lo que grandes cadenas de distribución al por menor, como Wal-Mart o Tesco, iniciaron la implementación de sistemas RFID para el seguimiento de sus cadenas de suministro. Generalmente, dichos sistemas se centran en el control de contenedores y palets mediante la presencia de lectores RFID en las puertas de los almacenes, que registran las entradas y salidas de los productos y se comunican con los sistemas centrales de la compañía.

La red académica "Auto-ID Labs"[4], anteriormente denominada "Auto-ID Center", es el máximo exponente en la normalización de RFID para la logística empresarial. Fue creado con el soporte de una gran cantidad de fabricantes, distribuidores y minoristas, y ha tenido un papel muy importante en la diseminación de los beneficios que esta tecnología supone para la SCM. En particular, el centro introdujo el denominado "Electronic Product Code" (EPC), que proporciona un esquema de numeración no ambigua para la identificación de productos, contenedores, servicios, compañías, localizaciones y activos en todo el mundo. Asimismo, "Auto-ID" ha desarrollado una serie de estándares relacionados que definen como dichos códigos deben ser almacenados en etiquetas RFID y transmitidos de forma inalámbrica. Actualmente, la regulación de EPC se encuentra a cargo de EPCglobal, filial de la asociación industrial sin ánimo de lucro GS1, que desde hace varias décadas también se encarga de la normalización de los códigos de barras. Entre sus objetivos principales se halla la creación de una red mundial de seguimiento de productos que se acerque a una ideal "Internet de las cosas".

E-Passports

A mediados del año 2004, la Organización Internacional de Aviación Civil (ICAO, en sus siglas inglesas) aprobó la especificación de un documento de viaje legible por máquinas ("Machine Readable Travel Document", o MRTD). Dicho documento emplea tecnología estándar RFID para

[4] http://autoidlabs.org/

el almacenaje de información personal y biométrica en pasaportes, visados y tarjetas de viaje. Debido a la capacidad de almacenaje actual de las etiquetas, en la práctica los datos biométricos se limitan a una fotografía del poseedor del documento, pero el estándar también prevé normas para el registro de información sobre el iris y la huella dactilar. A día de hoy, más de un centenar de paises de todo el mundo han implementado, o están en proceso de implementar, este sistema. La ICAO aboga por el uso en este tipo de documentos de etiquetas RFID compatibles con el estándar ISO 14443 que, como veremos más adelante, regula las tarjetas de identificación sin contacto.

Ticketing

Una de las primeras y más exitosas aplicaciones de la tecnología RFID fue su implementación como billetes de acceso a redes metropolitanas de transporte público. Actualmente pueden encontrarse estos sistemas en un gran número de ciudades en todo el mundo, siendo la Oyster Card londinense uno de los ejemplos más reconocidos. Otros casos relevantes son los de la tarjeta Octopus en Hong-Kong o la Suica en Tokyo, que han logrado trascender los límites de su uso en el transporte de masas para convertirse en sistemas de pago utilizables en un gran número de tiendas. Sin embargo, las funcionalidades que requieren estos sistemas van mucho más allá de lo que la tecnología RFID puede ofrecer por sí sola, por lo que usualmente se rigen por las especificaciones de las "Contactless Smartcards" que comentaremos en secciones posteriores.

Otras Aplicaciones

Como ya ha sido comentado anteriormente, el número de aplicaciones en las que se han implementado sistemas basados en RFID es muy elevado. Aparte de los ya explicados, cabe destacar su uso en sistemas de medición en carreras, especialmente en competiciones con un gran número de participantes, así como el desarrollo de implantes de identificación. El uso de estos implantes es habitual en el caso de los animales, permitiendo su control y seguimiento de alimentación y salud. Sin embargo, su empleo en humanos es muy controvertido y susceptible de problemas de seguridad, aunque se han llevado a cabo algunas pruebas en entornos reales, como, por ejemplo, en clubes nocturnos de Barcelona[5]. A pesar de todo, dichos sistemas fueron implementados hace ya más de 10 años y no han logrado difundirse a otros entornos, por lo que su consolidación definitiva es más que discutible.

2.2. Smart Cards y Contactless Smart Cards

Las tarjetas inteligentes o, como se les conoce habitualmente, las Smart Cards, son unas fichas del tamaño de tarjetas de crédito que contienen un microprocesador que computa y almacena

[5] http://edition.cnn.com/2004/WORLD/europe/06/09/spain.club/

información para dar soporte a una o varias aplicaciones, y que pueden funcionar tanto en local como de forma remota.

Sus orígenes se remontan al año 1968, durante el cual los alemanes Gröttrup y Dethloff inventaron la tarjeta con chip automatizado, para la cual no se les concedió la patente solicitada hasta casi quince años más tarde. En 1974, el inventor francés Roland Moreno patentó el concepto de tarjeta de memoria, mientras que tres años después Michael Ugon inventó la primera tarjeta inteligente con microprocesador. El primer uso masivo de las tarjetas fue en 1983 cuando se lanzaron las tarjetas de prepago para teléfonos públicos en Francia denominadas Télécarte.

Estas primeras tarjetas requerían el uso de interfaces por contacto para poder comunicarse con sus lectores. Sin embargo, es de sobras conocido que los contactos son una de las fuentes más frecuentes de fallos en los sistemas electromecánicos. Estos contactos pueden contaminarse y/o desgastarse, provocando errores de funcionamiento, así como recibir descargas electrostáticas por parte del lector que pueden incluso llegar a destruir la circuitería interna. Estos problemas pueden evitarse mediante las tarjetas sin contacto (en inglés, Contactless Smart Cards), que usan ondas de radio para comunicarse con el lector en lugar de un contacto físico. Estas fueron introducidas a partir de 1995 y su uso se ha ido generalizando en los últimos años.

2.2.1. Descripción Tecnológica

Una Smart Card, que también puede encontrarse en la literatura como tarjeta con chip o como tarjeta con circuito integrado (ICC o "Integrated Circuit Card"), es, en términos generales, un sistema electrónico de almacenaje de datos, con la posibilidad de incluir capacidad de computación adicional ("*microprocessor card*") que, por conveniencia, ha sido incrustado en una tarjeta de plástico de las dimensiones de una tarjeta de crédito convencional. Para poder operar deben ser capaces de recibir datos de entrada, que son procesados generando una respuesta de salida. Estas tarjetas son fabricadas en plástico, generalmente PVC y en ocasiones incluyen hologramas para evitar su falsificación. En los últimos años han sido elaboradas incluyendo incluso pantallas y teclados.

En relación con sus capacidades computacionales, podemos clasificar las ICCs en dos grandes categorías. Las tarjetas de memoria contienen únicamente componentes de almacenaje no volátiles, así como en algunas ocasiones determinada lógica de seguridad específica. Por otro lado, las tarjetas con microprocesador incluyen, como su nombre indica, un microprocesador, así como memoria volátil, lo que les permite desarrollar computaciones complejas y, por tanto, contener aplicaciones de alto nivel.

En cuanto a la forma de interacción con el exterior, las Smart Cards convencionales deben colocarse en un lector, que genera una conexión galvánica con las superficies de contacto de la tarjeta a través de unos muelles de contacto. A través de esas superficies se le suministra a la tarjeta la energía y el pulso de reloj necesarios para su funcionamiento. Sin embargo, existen también Smart Cards sin contacto, como las de proximidad y las de vecindad, que son capaces

de comunicarse con el lector a través de un campo de radiofrecuencia usando el principio de acoplamiento inductivo que ha sido descrito para la tecnología RFID. Por tanto, a grandes rasgos podríamos entender las tarjetas sin contacto como etiquetas RFID a las que se les ha dotado de la funcionalidad que ofrecen las Smart Cards, como la posibilidad de desarrolar cálculos complejos, ampliando así sus potenciales aplicaciones. Un diagrama de bloques simplificado de la estructura interna de este tipo de tarjetas puede encontrarse en la Fig. 2.2, mientras que la Fig. 2.3 muestra como se construye habitualmente una Smart Card con antena RFID incorporada.

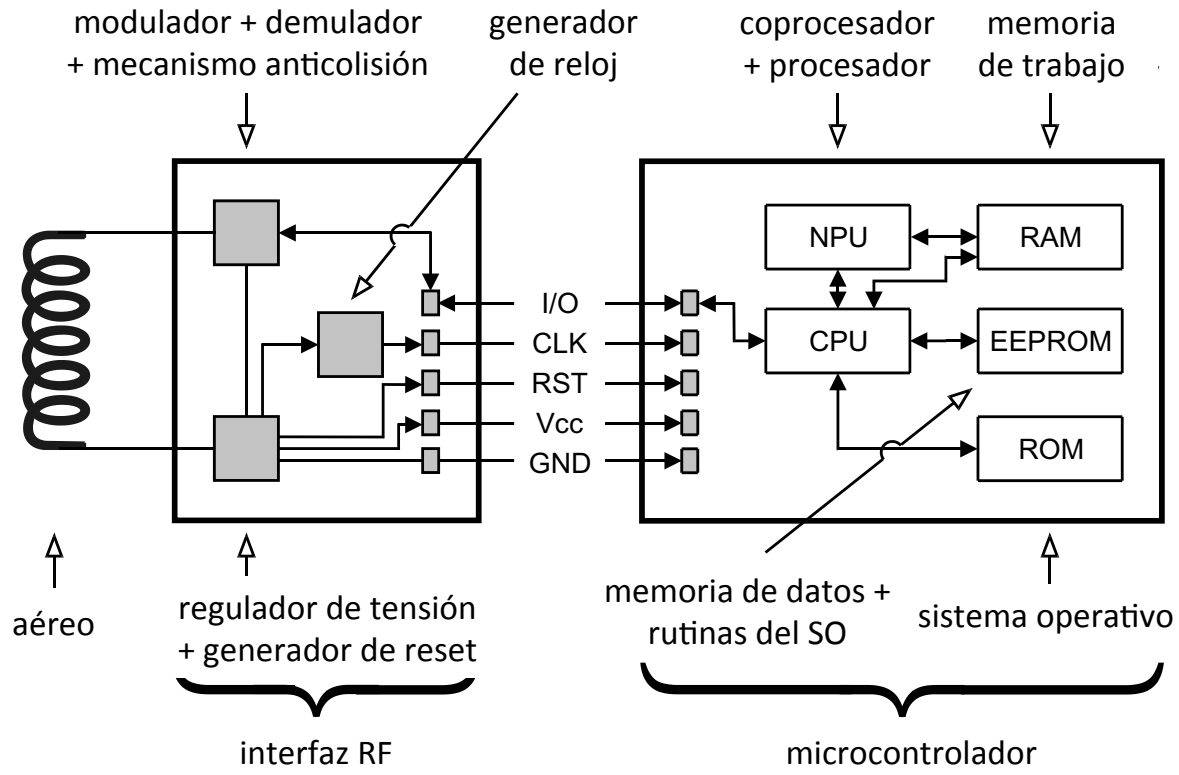


Figura 2.2: Arquitectura habitual de una Contactless Smart Card (Figura adaptada de Rankl y Effing [32])

2.2.2. Estandarización

Las Smart Cards y sus tecnologías de comunicación relacionadas se encuentran estandarizadas por ISO/IEC, principalmente en la norma ISO 7816, que deriva de la más genérica normativa relativa a las tarjetas ID-1 que se encuentra recogida en la publicación ISO 7810. En relación con aplicaciones concretas, el estándar adoptado por la gran mayoría de emisores de tarjetas de pago con chip integrado se denomina EMV (acrónimo de "Europay, Mastercard, Visa"), mientras que el estándar al que se acogen las Smart Cards usadas para las comunicaciones móviles es conocido como SIM/UICC.

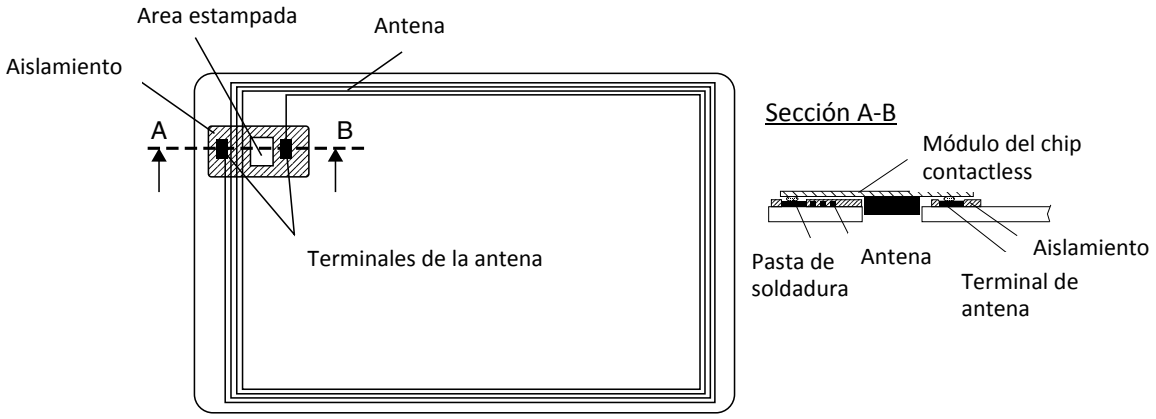


Figura 2.3: Esquema de la construcción física de una Contactless Smart Card (Figura adaptada de Finkenzeller [11])

Con respecto a las Contactless Smart Cards, actualmente existen tres estándares diferentes publicados por ISO, que se corresponden a grandes rasgos con tres rangos distintos de operacion entre el lector y la tarjeta.

- Las denominadas tarjetas de acoplamiento cercano ("*Close coupling*"), cuya distancia de operación es inferior al centímetro, se encuentran regladas en la norma ISO 10536. Este estándar fue desarrollado en su mayoría entre 1992 y 1995, aunque ha sido raramente puesto en práctica ya que ofrece escasas ventajas respecto a las tarjetas con contacto tradicionales con un coste muy superior. El principal inconveniente que presentan es que deben ser insertadas en el lector del mismo modo que las Smart Card convencionales, por lo que para el usuario no ofrecen un incremento de confortabilidad suficiente como para justificar su mayor coste.

- El estándar ISO 14443 trata acerca de las tarjetas de acoplamiento próximo o, simplemente, tarjetas de proximidad ("*Proximity cards*"). Estas engloban a todas aquellas que estén preparadas para trabajar a distancias de unos 10 centímetros del lector. La gran mayoría de tarjetas sin contacto que se encuentran en el mercado se incluyen en esta categoría, como, por ejemplo, las destinadas a billetería de transporte público.

- Finalmente, la norma ISO 15693 regula las características de lo que denomina tarjetas de vecindad ("*Vicinity cards*"), cuya distancia de operación alcanza el metro. Este tipo de tarjetas son las más adecuadas para sistemas de control de acceso y, habitualmente, cuentan con sistemas de procesamiento mucho más simples que las de proximidad.

De estos tres estándares, el que más nos interesa es el correspondiente a las tarjetas de proximidad (ISO 14443) ya que es el que adoptan, entre otras, las tarjetas Mifare DESFire que

serán utilizadas en el desarrollo del presente proyecto. Esta norma está compuesta por cuatro partes, que regulan los siguientes aspectos:

1. Características físicas
2. Interfaz de potencia y señal por radiofrecuencia
3. Inicialización y anticolisión
4. Protocolos de transmisión

La nomenclatura empleada en esos cuatro documentos es la misma que utilizaremos en este informe. En concreto se hace referencia a las tarjetas sin contacto como PICCs (acrónimo de "Proximity Integrated Circuit Card") y a los lectores como PCDs ("Proximity Coupling Device").

Es importante mencionar que en el estándar se definen dos interfaces de comunicación diferentes, a los que se denomina Tipo A y Tipo B. Este hecho no fue causado por razones técnicas sino a que en el momento de la preparación del estándar ya existían varios diseños de distintos fabricantes, por lo que fue necesario llegar a una solución de compromiso. Esto implica que si se pretende lograr una completa interoperabilidad con todas las tarjetas que se acojan a la norma ISO 14443, los terminales deben ser capaces de soportar ambos métodos ya que, en general, las tarjetas tan sólo serán compatibles con uno de ellos. Por tanto, mientras un terminal se encuentra esperando la detección de una PICC, debe alternar periódicamente entre los dos métodos de comunicación. En el momento en que el PCD haya reconocido una tarjeta en su campo, mantendrá el método adecuado hasta que la PICC haya sido desactivada por el terminal o haya abandonado el rango de trabajo del terminal. De los dos tipos mencionados el que más nos interesa es el Tipo A ya que las tarjetas de la familia Mifare, como las DESFire, se engloban dentro de esa categoría.

2.2.3. Aplicaciones

Las tarjetas inteligentes se han convertido en elemento importante para el desarrollo de aplicaciones electrónicas y de entorno móvil. Su capacidad para almacenar datos de forma segura, como claves privadas del usuario en infraestructuras de clave pública o información biométrica de su propietario, las hacen especialmente idóneas para un gran número de tareas con elevados requisitos de seguridad. La tarjeta, o una aplicación concreta que se encuentre en la tarjeta, pueden ser activadas a través de métodos de verificación, como códigos PIN o muestras biométricas, lo que evita problemas de seguridad asociados con la transmisión de credenciales de autentificación a través de redes de computadores. Entre la gran variedad de aplicaciones en las que son usadas las Smart Cards cabe destacar los servicios financieros (hoy en día es raro encontrar una tarjeta de crédito o débito que no incluya un chip) y las comunicaciones móviles (en forma de tarjetas SIM).

En cuanto a las tarjetas sin contacto, aparte de sus ventajas técnicas ya mencionadas, sus particularidades ofrecen una amplia gama de nuevas aplicaciones atractivas tanto para los emisores de tarjetas como para sus poseedores. Por ejemplo, dichas tarjetas no tienen por qué ser insertadas directamente en un lector, llegando incluso a distancias de un metro o más. Esto es una gran ventaja para los sitemas de control de acceso en los que una puerta o torno giratorio debe ser abierto, ya que la autorización de acceso de un individuo puede ser comprobada sin la necesidad de que su tarjeta sea extraída de su cartera o bolsillo para ser insertada en el lector. El transporte público es uno de los ámbitos en los que este tipo de tarjetas ha tenido un mayor éxito, ya que sus características resultan especialmente ventajosas cuando debe identificarse a un gran número de personas en el menor tiempo posible. En los últimos años, además, hemos sido testigos de su incorporación a las tarjetas de crédito y débito, con especial hincapié en los pequeños pagos que pueden realizarse de una forma más ágil.

2.3. NFC

Utilizando la misma idea del acoplamiento inductivo que ha popularizado la tecnología RFID explicada anteriormente, los sistemas NFC (acrónimo de "Near Field Communications" o Comunicaciones de Campo Cercano) se idearon como implementación del denominado "paradigma del toque", por el cual se obtienen servicios a través del acercamiento deliberado de objetos. Se podría decir que NFC es una combinación de tecnologías de identificación e interconexión sin contacto que requiere el acercamiento de dos dispositivos compatibles, esencialmente haciendo que se toquen. El usuario primero interacciona con un objeto inteligente (como una etiqueta NFC, un lector NFC o un dispositivo móvil con capacidad NFC) usando su propio dispositivo. Tras el toque, el dispositivo del usuario puede hacer uso de los datos recibidos o iniciar un servicio específico, como abrir una determinada página web, por ejemplo.

El desarrollo de la tecnología NFC se inició a finales del año 2002 en una acción conjunta de Philips y Sony, con el fin de conseguir un protocolo compatible con las tecnologías sin contacto propietarias existentes en el mercado, FeliCa (Sony) y Mifare (Philips). En esencia, se trata de un protocolo half-duplex de corto alcance, que proporciona comunicaciones sencillas y seguras entre diversos dispositivos. Como en el caso de RFID, se basa en el acoplamiento inductivo entre los dispositivos transmisor y receptor, y la comunicación se establece entre ambos a distancias de pocos centímetros, siendo la frecuencia de operación de 13.56 MHz.

La principal diferencia que le distingue de otras tecnologías inalámbricas es que su alcance es tan corto que se necesita que los dos dispositivos a interaccionar prácticamente se toquen durante un instante. Pese a que esta característica pueda parecer una limitación, es en realidad la clave. Al contrario de lo que ocurre con los servicios por RFID o Bluetooth, basados en el descubrimiento de la presencia del dispositivo en la proximidad, estirar el brazo para acercar el dispositivo NFC hacia el detector es en sí una afirmación clara de nuestra voluntad por

autentificarnos, por pagar la compra o simplemente transferir un contacto.

2.3.1. Descripción Tecnológica

En una comunicación NFC siempre encontramos dos partes diferenciadas a las que se les denomina dispositivo iniciador ("initiator"), que, como indica su nombre, es el que comienza y guía el proceso de intercambio de datos entre las partes, y objetivo ("target"), que se limita a responder a las peticiones del primero. La comunicación que se produce entre ellos puede realizarse de dos modos distintos, que el protocolo NFC denomina modo pasivo y modo activo:

Pasivo En este modo, únicamente el dispositivo iniciador de la comunicación genera el campo electomagnético, mientras que el dispositivo objetivo utiliza la energía transferida por dicho campo para transmitir la respuesta. Este modo resulta de utilidad tanto con etiquetas que no disponen de fuente de alimentación propia como en dispositivos alimentados por batería en aquellos momentos en que deben priorizar la conservación de energía. El protocolo permite que tales dispositivos, como sería el caso de los teléfonos móviles, puedan operar en modos de ahorro de energía, evitando tener que alimentar la antena de radiofrecuencia si recibe la energía necesaria de la antena del lector iniciador de la comunicación.

Activo Cuando se emplea el modo activo de comunicación, ambos dispositivos generan su propio campo electomagnético, que utilizarán para transmitir sus datos. Esto implica que ambas partes requieren de alimentación interna. Este modo supone la mayor diferencia con la tecnología RFID ya que permite que un dispositivo actúe simultáneamente como lector y como tag, lo que incrementa las posibles aplicaciones a desarrollar. En RFID, por contra, los roles de etiqueta y lector son fijos, lo que en cierto sentido limita las funcionalidades disponibles a cambio de un coste mucho menor.

A grandes rasgos, en las comunicaciones NFC pueden distinguirse entre dispositivos activos NFC (usualmente teléfonos inteligentes con funcionalidad NFC), etiquetas NFC y lectores NFC. Dichos elementos son utilizados en alguno de los tres modos de operación que se hallan estandarizados: lector/escritor, peer-to-peer y emulación de tarjetas. Cada modo de operación utiliza un interfaz de comunicación diferente en la capa de RF, así como diferentes requisitos técnicos, operacionales y de diseño.

Modo de operación Lector/Escritor En el modo de operación lector/escritor, un dispositivo NFC activo inicia una comunicación wireless, y puede tanto leer como modificar datos almacenados en etiquetas NFC (o tags RFID simples). En este modo de operación, un dispositivo NFC tiene la capacidad de leer los diferentes tipos de etiqueta establecidos por el NFC Forum, lo que permite al usuario del dispositivo recuperar los datos almacenados en la etiqueta y realizar la acción apropiada con posterioridad.

Modo de operación Peer-to-peer Al emplear modo de comunicación peer-to-peer, dos dispositivos activos NFC establecen una conexión bidireccional para intercambiar información. Este modo permite un modelo de petición-respuesta en el que se puede intercambiar cualquier tipo de datos, como, por ejemplo, tarjetas de visita, fotografías digitales y muchos más.

Para permitir este tipo de operaciones, el protocolo NFCIP-1 definido por el NFC Forum proporciona capacidad SAR (Segmentación y Reensamblaje) de nivel 1, así como control del flujo de datos de acuerdo con el principio "Go and Wait" habitual en los protocolos half-duplex. Además, el protocolo NFCIP-1 permite el manejo de errores mediante el uso de tramas de aceptación (ACK) y rechazo (NACK), proporciona un fujo de datos ordenado y desarrolla la comunicación en la capa de enlace de forma fiable y libre de errores.

Un nuevo protocolo de la capa de enlace llamado LLCP ha sido estandarizado por el NFC Forum para soportar comunicaciones peer-to-peer entre dos dispositivos NFC. Este protocolo es esencial para cualquier aplicación NFC que desarrolle una comunicación bidireccional, proporcionando unos fundamentos sólidos de comunicación punto a punto y mejorando las funcionalidades básicas proporcionadas por el protocolo NFCIP-1.

Modo de operación de Emulación de tarjetas Finalmente, en el modo de operación de Emulación de tarjetas, los dispositivos NFC usan protocolos analógicos y digitales similares a los definidos en los estándares para tarjetas inteligentes, como ISO/IEC 14443 Tipo A, Tipo B y FeliCa. En el momento en el que el usuario toca un lector NFC con su dispositivo NFC, dicho dispositivo se comporta como si se tratara de una tarjeta inteligente. El lector interactua con las aplicaciones que se encuentran en el SE (Secure Element) del dispositivo para realizar una comunicación eficiente y segura. La existencia de este modo de operación garantiza que aquellas aplicaciones diseñadas para ser llevadas a cabo con tarjetas sin contacto que cumplan con los estándares anteriormente mencionados podrán también ser empleadas en dispositivos inteligentes NFC.

2.3.2. Estandarización

La tencología NFC se beneficia de diversos elementos como smart cards, dispositivos inteligentes, lectores de tarjetas y sistemas de pago. Varios cuerpos de estandaridación han definido como la tecnología NFC debe ser integrada en teléfonos móviles y otros dispositivos relacionados, así como las arquitecturas y estándares para elementos de seguridad. El punto de vista común para todos dichos cuerpos de estandarización es aumentar la facilidad de acceso, interoperabilidad y seguridad para la tecnología NFC.

La asociación más importante que se centra en desarrollar y mejorar el uso de interacciones inalámbricas de rango corto a través de la tecnología NFC es el NFC Forum[6], que ya ha sido

[6] http://nfc-forum.org/

mencionado anteriormente. Se trata de una asociación industrial sin ánimo de lucro que fue establecida con el objetivo de lanzar la tecnología NFC en sus inicios, y más tarde en extender su uso por todo el mundo. Su misión es promocionar el uso de la tecnología NFC desarrollando especificaciones, asegurando la interoperabilidad entre dispositivos y servicios, y educando al mercado acerca de la tecnología NFC. Hasta el momento, el NFC Forum ha proporcionado diversas especificaciones para los distintos componentes de la tecnología NFC como LLCP (Logical Link Control Protocol), tipos de etiquetas NFC, NFC RTDs (Record Type Definitions) y otros. Asimismo, este organismo ha estandarizado un formato de intercambio de datos entre las partes de una comunicación llamado NFC Data Exchange Format, o NDEF. Este estándar define el formato que deben seguir los datos intercambiados tanto entre un dispositivo activo y una etiqueta pasiva como entre dos dispositivos activos.

Finalmente, cabe mencionar que los estándares anteriormente mencionados en los apartados correspondientes a RFID y Contactless Smart Cards también deben ser tenidos en cuenta a la hora de desarrollar elementos de un sistema NFC, ya que dichas normas resultan también de aplicación.

2.3.3. Aplicaciones

Como el acoplamiento inductivo funciona solamente en distancias cortas, el rango de operación de la tecnología NFC es tan solo de un máximo de 20 cm, muy inferior al de Bluetooth, que alcanza los 10 m, o el de Wi-Fi, que puede llegar hasta los 100 m. Este hecho, juntamente con la limitada velocidad de transferencia, del orden de los centenares de kbps frente a los varios Mbps de Bluetooth y Wi-Fi, hacen que NFC no sea la solución óptima para la transmisión de grandes volúmenes de información. Sin embargo, las aplicaciones potenciales que pueden ser desarrolladas en este entorno son innumerables. De ellas, podríamos destacar los siguientes ejemplos:

- El intercambio de datos como tarjetas de visita o sincronización de eventos de agenda entre dos dispositivos activos NFC, como podrían ser dos teléfonos móviles o un teléfono con un ordenador portatil, usando el modo peer-to-peer

- Uso de NFC para la configuración, inicialización y emparejamiento de dispositivos Bluetooth, Wi-Fi o Wireless USB, protocolos que posteriormente podrían ser utilizados para la transferencia de contenidos de datos mucho más voluminosos de lo que permite NFC

- Sistemas de pago sin contacto, compatibles tanto con dispositivos móviles como con tarjetas inteligentes

- Posters inteligentes con funcionalidad NFC, que respondan al acercamiento de dispositivos

2.4. Conclusión

Las funcionalidades que aportan las tecnologías de comunicación sin contacto tienen unas ventajas indudables que las hacen cada vez más atractivas. El número de dispositivos que cuentan con conectividad NFC es cada día mayor, aunque actualmente es el formato de las tarjetas *contactless* el que parece estar recibiendo un mayor impulso institucional y el que poco a poco se está asentando entre los usuarios. De entre este tipo de tarjetas, las de la familia Mifare, desarrolladas actualmente por NXP, han sido siempre unas de las más comunes, siendo empleadas por multitud de entidades a lo largo y ancho del globo. Dentro de los distintos modelos incluidos en la familia Mifare destacan por prestaciones las denominadas DESFire, cuya principal característica es el uso de criptografía de clave simétrica estándar para garantizar la seguridad tanto de la información almacenada como de las comunicaciones con el exterior. En el próximo capítulo explicaremos con detalle las principales características de este tipo de tarjetas, ya que se trata del modelo concreto al que se dirige la librería software desarrollada en el presente proyecto.

3 | Mifare DESFire y Mifare DESFire EV1

En este capítulo describiremos las principales características de los modelos de Contactless Smart Card para los que la librería desarrollada en el presente proyecto va dirigida. Dado que dicha librería fue diseñada inicialmente para la comunicación con tarjetas Mifare DESFire y, posteriormente, fue adaptada para que resultara igualmente funcional con la versión evolucionada denominada Mifare DESFire EV1, consideramos que es necesario tratar ambas en este capítulo, a pesar de que actualmente únicamente la versión más reciente se encuentra comercialmente disponible.

La tarjeta Mifare DESFire fue lanzada en el año 2002 por Philips Semiconductors como una evolución de su familia de tarjetas inteligentes sin contacto Mifare, que dominaba el mercado desde mediados de los años 90. Más concretamente, se trataba de una versión especial de las tarjetas de tipo Mifare `T = CL` llamadas SmartMX, que disponían de un sistema operativo de tarjeta (COS) que les permitía realizar operaciones complejas de modo similar a las Smart Card con contacto. Como su nombre indica, la principal particularidad de las DESFire es que incorporaban un motor criptográfico para la autentificación y el cifrado de las comunicaciones basado en DES/Triple-DES. Su uso fue muy extendido, especialmente en sistemas de pago de transporte en ciudades de todo el mundo.

Desde un punto de vista teórico, el cifrado Triple-DES empleado en dichas tarjetas es seguro, ya que no se conocen análisis criptoanalíticos eficientes, más allá de la fuerza bruta. Sin embargo, en estudios realizados por Oswald y Paar [25] se utilizaron ataques de canal lateral ("side-channel attacks"), es decir, ataques que explotan las características físicas de la implementación concreta en el hardware, para obtener las claves almacenadas en la tarjeta a base de realizar medidas no invasivas de las emisiones electromagnéticas del dispositivo. De este modo fueron capaces de leer, manipular y duplicar los contenidos de tarjetas Mifare DESFire.

NXP Semiconductors[1], empresa escindida en 2006 de Philips que se quedó con su negocio de semiconductores, incluida la línea de tarjetas sin contacto Mifare[2], decidió a mediados de

[1] http://www.nxp.com/
[2] http://www.mifare.net/

2010 finalizar su producción de tarjetas Mifare DESFire para sustituirlas progresivamente por su evolución, denominada Mifare DESFire EV1, que había sido lanzada dos años antes con mejoras tanto de seguridad como de funcionalidad. Como veremos más adelante, este modelo se ofrece con diferentes capacidades de almacenamiento, y amplía las posibilidades criptográficas al permitir el uso de 3KTDES (Triple-DES de Triple Clave) y AES, alcanzando la certificación de Common Criteria EAL 4+ (tanto en hardware como en software).

Antes de proceder a detallar las características de cada uno de los modelos, es imprescindible hacer notar que, para poder acceder a la documentación completa que NXP ofrece acerca de este tipo de tarjetas, ha sido necesario firmar un "Non-Disclosure Agreement" (NDA) con la compañía, por el cual no puede revelarse cierta información contenida en documentos que han sido clasificados como confidenciales. Esto implica que, aunque muchas de las especificaciones de carácter confidencial han resultado imprescindibles para la correcta implementación de la librería, el presente informe únicamente puede mencionar aquellas características de las tecnologías empleadas que pueden ser encontradas en las versiones públicas de la documentación, así como las obtenidas de forma experimental por terceras partes.

3.1. Características Técnicas y Funcionales

A grandes rasgos, las características técnicas de las tarjetas Mifare DESFire (MF3ICD40) y las de los diferentes modelos de Mifare DESFire EV1 (MF3ICDX1, con X pudiendo ser 2, 4, u 8, dependiendo de la capacidad de almacenamiento), son muy similares, diferenciándose únicamente en ciertas particularidades. Por ello, se ha decidido explicarlos conjuntamente, indicando explícitamente aquellas especificaciones que hayan sido modificadas en la evolución. En adelante, para referirnos tanto a las tarjetas Mifare DESFire como a las Mifare DESFire EV1 utilizaremos la denominación común MF3ICDXX derivada de las referencias empleadas por el fabricante.

Dada la elevada importancia de las características de seguridad, se tratarán de forma detallada en la Sección 3.3, por lo que no se incluyen aquí.

3.1.1. Interfaz de RF y Transferencia de Datos

El interfaz de radiofrecuencia de las tarjetas MF3ICDXX está diseñado para trabajar con lectores compatibles con la norma ISO 14443 Tipo A, ofreciendo transmisión de datos sin contacto y alimentación obtenida del campo de radiofrecuencia generado por dicho lector, por lo que no necesitan batería propia. Su distancia de operación alcanza los 100 mm, dependiendo de la geometría de la antena del lector, y su frecuencia de operación es la estándar de 13.56 MHz.

Cuando la tarjeta es situada en las proximidades de la antena de un PCD (Proximity Coupling Device), el interfaz de comunicación RF permite transmitir datos a 106 kbit/s, 212 kbit/s o 424 kbit/s, así como a 848 kbit/s en el caso de tratarse de una tarjeta EV1. Los datos son transmitidos con sistemas de preservación de la integridad, como paridad de bits, codificación y contaje de

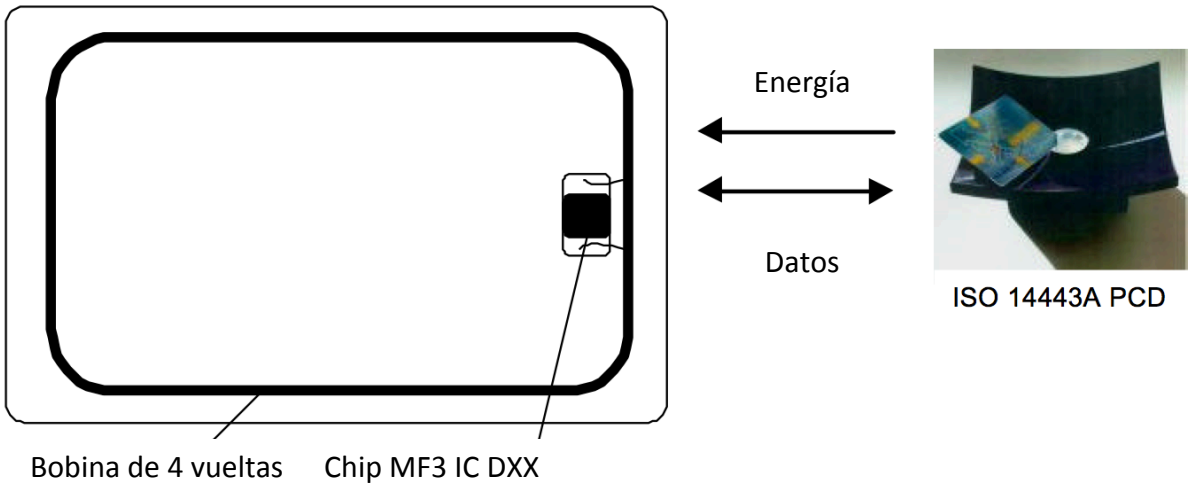


Figura 3.1: Esquema simplificado del funcionamiento del interfaz de RF en tarjetas MF3ICDXX (Figura adaptada de [27])

bits, y CRC de 16 bits (y de 32 en el caso de EV1), además de la posibilidad de usar códigos MAC de 4 bytes, y CMAC de 8 bytes en las tarjetas Mifare DESfire EV1.

Las tarjetas MF3ICDXX también incluyen un mecanismo de anticolisión inteligente, que permite que puedan realizarse transacciones incluso con más de una tarjeta presente en el campo al mismo tiempo. El algoritmo selecciona cada tarjeta individualmente y asegura que la ejecución de una transacción con una tarjeta seleccionada se realice correctamente sin corrupción de datos por la presencia de otras tarjetas operando simultáneamente.

3.1.2. Organización de la Memoria

La memoria no volátil de las tarjetas MF3ICDXX está organizada en un sistema de almacenaje flexible de 4 kbytes, en caso de las tarjetas Mifare DESFire originales, y de 2, 4, u 8 kbytes, según la opción comercial escogida, en caso de las tarjetas Mifare DESFire EV1. Las especificaciones oficiales aseguran una retención de los datos en la memoria no volátil de 10 años, y una capacidad de escritura de hasta 100.000 ciclos, incrementando esta cifra en el caso de las EV1 hasta los 500.000 ciclos.

Una de las características principales de las tarjetas MF3ICDXX es que se trata de tarjetas multiaplicación, es decir, que permiten la creación y utilización de espacios de memoria independientes, cada uno con una serie de ficheros y claves propios. En concreto, pueden manejarse hasta 28 aplicaciones diferentes en una tarjeta. Cada una de esas aplicaciones es identificada con un número único de 3 bytes denominado AID (Application IDentifier), y puede contener hasta 16 ficheros (32 en el caso de EV1) y 14 claves distintas.

Los ficheros pueden ser de los siguientes tipos:

Standard Data Fichero para el almacenaje de datos de usuario sin formato

Backup Data Fichero para el almacenaje de datos de usuario sin formato que soporte un mecanismo de respaldo integrado

Value Fichero para el almacenaje y manipulación de un entero con signo de 32 bits

Linear Record Fichero para el almacenaje de múltiples datos de similar estructura, con escritura lineal (cuando el fichero está ocupado completamente con registros de datos no es posible continuar escribiendo hasta que no sea limpiado)

Cyclic Record Fichero para el almacenaje de múltiples datos de similar estructura, con escritura cíclica (se sobreescribe el más antiguo cuando el fichero está ocupado completamente)

De ellos, únicamente el primero no dispone de un sistema de respaldo automático para asegurar la integridad de los datos a nivel de aplicación. Es posible mezclar en una misma aplicación los diferentes tipos de fichero, pero en las tarjetas Mifare DESFire originales, los ficheros con respaldo deben necesariamente almacenarse en los 8 primeros espacios de cada aplicación (identificadores de fichero 0 a 7), ya que los demás no disponen del mecanismo de respaldo.

La ejecución de comandos que tienen un impacto directo en la estructura de ficheros, como la creación o eliminación de aplicaciones, por ejemplo, activan un sistema automático de "rollback" para proteger dicha estructura de una posible corrupción. En caso de que sea necesario su uso, no se requiere la intervención del usuario para su funcionamiento, y se llevará a cabo antes de proceder con nuevas instrucciones recibidas.

3.2. Comandos de Operación

Aunque las tarjetas MF3ICDXX disponen de unidades centrales de proceso (CPU), no soportan la ejecución de código definido por el cliente. En su lugar deben emplearse una serie de comandos predefinidos para la construcción de aplicaciones específicas. Aparte de aquellos definidos en las normas ISO/IEC 14443-3 y 14443-4, y los inter-industria incluidos en la norma ISO/IEC 7816-4, los comandos más útiles para los desarrolladores son aquellos que se conocen como "comandos nativos DESFire". Dichos comandos se listan a continuación agrupados en las cuatro categorías que se distinguen habitualmente: de seguridad, a nivel de tarjeta, a nivel de aplicación y de manipulación de datos.

Es importante mencionar que, aparte de modificar el funcionamiento de muchos de los comandos, en la Mifare DESFire EV1 algunos nuevos fueron introducidos especialmente en dicha versión de la tarjeta, por lo que no se encuentran disponibles para su uso con las Mifare DESFire originales. Estos comandos serán indicados en los apartados posteriores con "*(EV1)*" tras su nombre.

3.2.1. Comandos de Seguridad

Los mecanismos que son utilizados en las tarjetas MF3ICDXX para garantizar la seguridad en las comunicaciones serán explicados con mayor detalle en su propia sección. En cualquier caso, listamos aquí los comandos que han sido definidos para las operaciones de seguridad con las tarjetas.

Authenticate Permite que la tarjeta y el dispositivo lector se muestren mutuamente que conocen un mismo secreto o, más concretamente, una clave

Change Key Settings Cambia la configuración de claves tanto a nivel de tarjeta como de aplicación

Set Configuration *(EV1)* Configura la tarjeta y la pre-personaliza con una clave, define si en la inicialización de la comunicación se devuelve la UID de la tarjeta o un identificador aleatorio, y permite definir la cadena ATS ("Answer to Select")

Change Key Permite cambiar cualquier clave almacenada en la tarjeta

Get Key Version Obtiene la versión de clave actual de cualquier clave almacenada en la tarjeta

3.2.2. Comandos a Nivel de Tarjeta

Create Application Crea nuevas aplicaciones en la tarjeta

Delete Application Desactiva permanentemente aplicaciones en la tarjeta

Get Application IDs Devuelve los identificadores de aplicación de todas las aplicaciones activas de la tarjeta

Free Memory *(EV1)* Devuelve la cantidad de memoria disponible en la tarjeta

Get DF Names *(EV1)* Devuelve los nombres DF[3] que hayan sido especificados

Get Key Settings Devuelve información acerca de la configuración de claves

Select Application Selecciona una aplicación específica en la tarjeta para su ulterior uso

Format PICC Libera la memoria de usuario de la tarjeta

Get Version Devuelve la información de fabricación de la tarjeta

Get Card UID *(EV1)* Devuelve el identificador único de tarjeta (UID)

[3]El concepto de `DF-Name`, o nombre DF, se encuentra definido en la documentación confidencial de NXP

3.2.3. Comandos a Nivel de Aplicación

Get File IDs Devuelve los identificadores de todos los ficheros activos dentro de la aplicación seleccionada en ese momento

Get File Settings Obtiene información acerca de las propiedades de un fichero específico

Change File Settings Modifica los parámetros de acceso a un fichero preexistente

Create Standard Data File Crea un fichero de tipo Standard Data

Create Backup Data File Crea un fichero de tipo Backup Data

Create Value File Crea un fichero de tipo Value

Create Linear Record File Crea un fichero de tipo Linear Record. Cuando el fichero está ocupado completamente con registros de datos no es posible continuar escribiendo hasta que no sea limpiado

Create Cyclic Record File Crea un fichero de tipo Cyclic Record. Cuando el fichero está ocupado completamente con registros de datos la tarjeta automáticamente sobreescribe el registro más antiguo con la nueva información

Delete File Desactiva permanentemente un fichero de la aplicación que se encuentre seleccionada

3.2.4. Comandos de Manipulación de Datos

Read Data Lee datos de ficheros de tipo Standard Data o Backup Data

Write Data Escribe datos en ficheros de tipo Standard Data o Backup Data

Get Value Lee el valor actualmente almacenado en un fichero de tipo Value

Credit Incrementa el valor almacenado en un fichero de tipo Value

Debit Decrementa el valor almacenado en un fichero de tipo Value

Limited Credit Permite un incremento limitado de un valor almacenado en un fichero de tipo Value sin disponer del acceso completo a las operaciones Credit

Read Records Lee un conjunto de registros completos de un fichero de tipo Cyclic o Linear Record

Write Record Escribe datos en un registro de un fichero de tipo Cyclic o Linear Record

Clear Record File Resetea un fichero de tipo Cyclic o Linear Record a su estado vacío

Commit Transaction Valida todos los accesos previos de escritura a ficheros de tipo Backup Data, Value o Record dentro de una aplicación

Abort Transaction Anula todos los accesos previos de escritura a ficheros de tipo Backup Data, Value o Record dentro de una aplicación

3.3. Mecanismos de Seguridad

Uno de los aspectos más relevantes de las tarjetas Mifare DESFire y Mifare DESFire EV1 es el conjunto de medidas de seguridad que incluyen para poder ofrecer transacciones de confianza a sus usuarios. En esta sección veremos de qué mecanismos se dispone para garantizar la autenticidad de las partes, así como para asegurar la confidencialidad de las transmisiones. Finalmente, también veremos cuales son los recursos más habituales para manejar las claves a emplear, reduciendo el riesgo de que resulten comprometidas.

3.3.1. Identificador Único (UID)

Uno de los mecanismos más básicos de seguridad de los que disponen las tarjetas MF3ICDXX es la presencia de un identificador único de 7 bytes (UID) programado en una zona bloqueada de la memoria no volátil, únicamente accesible por parte del fabricante. Debido a requisitos de seguridad, esos bytes son protegidos de escritura tras haber sido programados por el fabricante durante su producción.

Como puede verse en la Fig. 3.2, tal y como se indica en la norma ISO/IEC 14443-3, el primer byte del UID contiene el identificador del fabricante. En el caso de las tarjetas MF3ICDXX, este corresponde al `0x04` que inicialmente usaba Philips y actualmente representa a NXP.

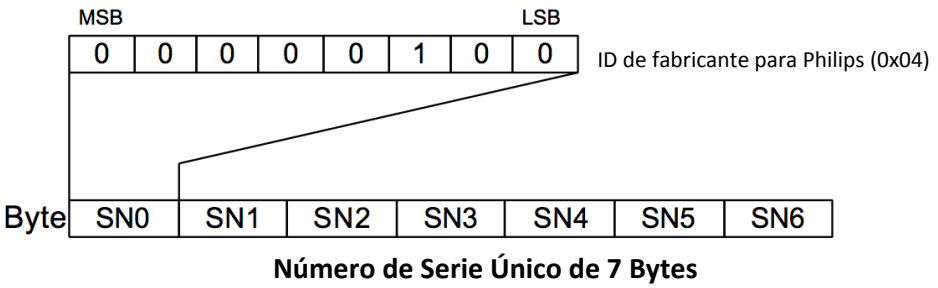


Figura 3.2: Estructura del Identificador Único de tarjeta (UID) (Figura adaptada de [27])

Las tarjetas Mifare DESFire EV1 también permiten el uso de un identificador aleatorio (Random ID), que se genera cada vez que se inicia una transacción en caso de que así haya sido indicado mediante el comando `Set Configuration`. Esto consigue aumentar la privacidad del poseedor de la tarjeta (por ejemplo, impidiendo que pueda determinarse el número de veces que

un mismo individuo accede a un servicio) pero, a cambio, no permite el uso de herramientas como la diversificación de claves basada en el UID.

3.3.2. Autentificación y Clave de Sesión

Gran parte de las acciones que pueden realizarse con las tarjetas MF3ICDXX requieren que una autentificación válida sea completada con anterioridad. El propósito de la autentificación con clave es no sólo garantizar la autorización para realizar determinadas acciones, sino también permitir que la tarjeta y el lector establezcan una clave de sesión común para realizar transmisiones encriptadas. Dicha clave de sesión impide que el contenido de los paquetes intercambiados sean comprendidos por atacantes que puedan capturarlos. Los métodos de encriptación que pueden ser escogidos son DES, Triple DES (con dos claves `k1-k2-k1`, o, también, con tres distintas en EV1) y AES, estando esta última opción disponible únicamente en la versión EV1.

Es importante mencionar que en determinadas ocasiones, como veremos más adelante, es posible que se requiera una autentificación con una clave concreta y, sin embargo, la posterior transferencia de datos se realice en texto llano, es decir, sin cifrado con la clave de sesión generada. Las autentificaciones son, además, perecederas, ya que un lector únicamente puede estar autentificado con una clave en cada momento y cada nuevo uso del comando de autentificación anula la anterior, generando una nueva clave de sesión, aunque dicha autentificación se haya producido con la misma clave consecutivamente.

El proceso de autentificación se denomina habitualmente "Autentificación Mutua en 3 Pasos" ("3-Pass Mutual Authentication") y puede resumirse en los siguientes puntos, que se encuentran esquematizados en la Fig. 3.3:

1. El proceso se inicia cuando el lector solicita la ejecución de un comando `Authenticate`, especificando el número de clave (`K`) con la que desea ser autentificado. En caso de que el número de clave sea `0x00`, la clave maestra (de la tarjeta o de la aplicación, dependiendo de qué aplicación se encuentre seleccionada) será la usada para la autentificación.

2. Si dicha clave se encuentra almacenada en la tarjeta, entonces ésta responde a la petición mediante la encriptación de un número aleatorio de 8 bytes (`RndB`) con la clave `K`. En caso contrario, se devuelve un código de error.

3. El lector ejecuta una operación de descifrado sobre el mensaje recibido usando la clave `K` para recuperar el número aleatorio `RndB`. Este número es rotado hacia la izquierda 8 bits, valor al que denominamos `RndB'`. Asimismo, el lector genera otro número aleatorio de 8 bytes (`RndA`), que es concatenado con `RndB'`. Ambos son entonces *desencriptados* usando la clave `K`.

4. La tarjeta realiza un proceso de encriptación sobre el mensaje recibido. De este modo es capaz de verificar el número `RndB'` obtenido y compararlo rotando internamente el original

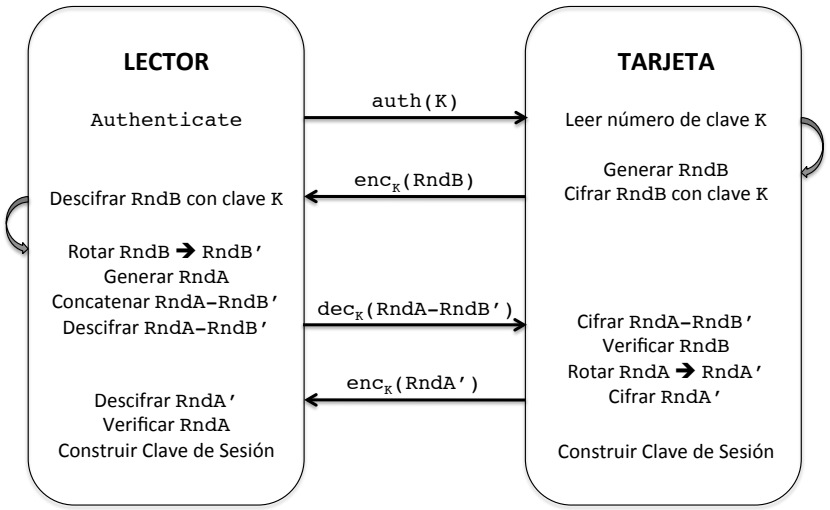


Figura 3.3: Esquema simplificado del proceso de Autentificación Mutua en 3 Pasos empleado en las tarjetas Mifare DESFire y Mifare DESFire EV1

hacia la iquierda en 8 bits. Si la verificación tiene éxito, se realiza la misma rotación sobre el número aleatorio `RndA` (`RndA'`) y se envía *encriptado* con la clave `K` de vuelta al lector. Además, la tarjeta activa el estado de autentificación correspondiente a dicha clave, ya que puede confiar en que el lector conoce el mismo secreto que ella.

5. Finalmente, el lector desencripta el mensaje de respuesta de la tarjeta y compara el valor `RndA'` con el `RndA` original rotado. Si ambos coinciden, el lector podrá confiar en la tarjeta y continuará con el proceso deseado.

Como hemos mencionado antes, en el caso de que la autentificación tenga éxito, se genera una clave de sesión que se obtiene de los números aleatorios que han sido intercambiados. En concreto, $\textit{Session Key} = \texttt{RndA}_{1stHalf} + \texttt{RndB}_{1stHalf} + \texttt{RndA}_{2ndHalf} + \texttt{RndB}_{2ndHalf}$.

3.3.3. Derechos de Acceso y Modos de Comunicación

En el apartado anterior hemos mencionado el proceso que se sigue para establecer una autentificación mutua entre lector y tarjeta, y, a su vez, negociar una clave de sesión. Sin embargo, este paso no siempre es imprescindible, ya que determinados comandos pueden llevarse a cabo sin autentificación previa (`Get Key Version`, por ejemplo) y, en ciertos casos, la comunicación posterior puede realizarse sin requerir la clave de sesión. Como ya ha sido comentado anteriormente, estas dos situaciones son independientes entre sí, ya que puede darse el caso que la autentificación con una determinada clave sea obligatoria para obtener el permiso necesario para realizar una operación concreta, pero que la ejecución de dicha operación se realice en texto llano sin cifrar. El primer caso viene determinado tanto por la definición del comando a ejecutar

como, si aplica, a los *derechos de acceso* que hayan sido definidos para el fichero sobre el que pretende realizar la operación. En el segundo caso serán los *modos de comunicación* que se hayan establecido los que determinarán cómo debe construirse el mensaje a enviar o recibir.

Derechos de Acceso

Los ficheros creados en las aplicaciones de una tarjeta MF3ICDXX cuentan con cuatro derechos de acceso diferentes que deben ser configurados: Lectura ("Read" o R), Escritura ("Write" o W), Lectura-Escritura ("Read-Write" o R&W) y Cambio de Configuración ("Change Configuration"). Los tres primeros derechos permiten realizar operaciones de lectura y/o escritura en todos los tipos de fichero excepto en los Value, en los que dan permiso para ejecutar los siguientes comandos:

R: `Get Value`, `Debit`

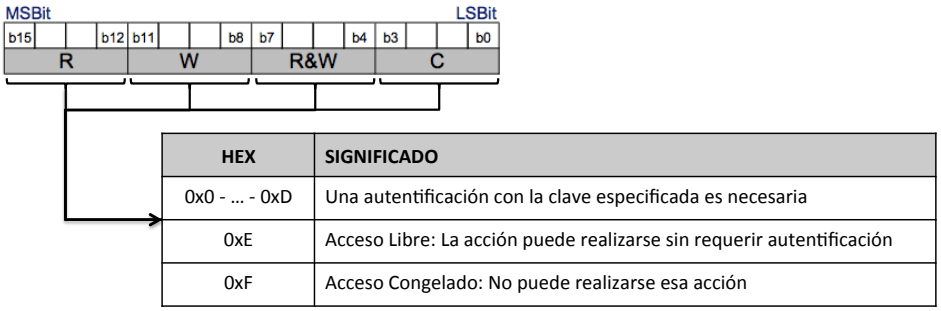
W: `Get Value`, `Debit`, `Limited Credit`

R&W: `Get Value`, `Debit`, `Limited Credit`, `Credit`

Como puede observarse, en los ficheros de tipo Value los comandos cuya ejecución se permite con cada derecho de acceso no se corresponden directamente con su denominación. Por ejemplo, la operación de lectura en dichos ficheros (`Get Value`) puede realizarse una vez obtenido cualquiera de los tres derechos, mientras que una de las posibles operaciones de escritura (`Credit`) tan sólo podrá ejecutarse con el permiso de lectura-escritura y no con el de únicamente escritura.

El derecho de cambio de configuración funciona de forma idéntica para todos los tipos de ficheros, y permite modificar cómo se consiguen adquirir cada uno de los posibles derechos para ese fichero concreto. Estos modos de adquisición vienen codificados en 2 bytes cada uno, que se definen en el momento de creación de un fichero y pueden ser modificados mediante el comando `Change File Settings`. Más concretamente, los "Access Rights" se codifican del modo que se encuentra representado en la Fig. 3.4, donde se indica con qué clave debe realizarse una autentificación satisfactoria para adquirir cada uno de los diferentes derechos.

Presentar un ejemplo concreto puede que resulte de utilidad para entender mejor cómo se utilizan los derechos de acceso en una comunicación con una tarjeta DESFire. Supongamos que deseamos modificar el contenido de un fichero que se halla incluido en una determinada aplicación de una tarjeta MF3ICDXX, y que ya hemos realizado todos los pasos previos para tener dicha aplicación seleccionada. Supongamos, además, que el fichero de interés es de tipo Standard Data y, por tanto, los derechos de acceso de lectura, escritura y lectura-escritura garantizan aquello que su nombre indica. Generalmente, si solicitamos la ejecución de un comando `Write Data` sin una autentificación previa lo más probable es que obtengamos un mensaje de error ("Permission Denied") como respuesta. Para tratar de solucionarlo, debemos usar el comando `Get File Settings`, que nos devuelve, entre otros datos, los derechos de acceso que hayan sido



HEX	SIGNIFICADO
0x0 - ... - 0xD	Una autentificación con la clave especificada es necesaria
0xE	Acceso Libre: La acción puede realizarse sin requerir autentificación
0xF	Acceso Congelado: No puede realizarse esa acción

Figura 3.4: Interpretación de la codificación hexadecimal de los derechos de acceso (Figura adaptada de [30])

definidos para ese fichero. Imaginemos que los "Access Rights" que recuperemos son `0xE10F`, lo que indicaría que:

- La lectura de los contenidos del fichero se puede realizar libremente (`0xE`), es decir, sin necesidad de una autentificación previa
- Para poder modificar el contenido del fichero es necesaria una autentificación satisfactoria con la clave 1 (`0x1`) o bien con la clave maestra de la aplicación (`0x0`). Esta última clave también concede permisos de lectura ya que se encuentra asociada al derecho de lectura-escritura
- La configuración del fichero, incluidos sus derechos de acceso, no pueden volver a ser modificados, ya que han sido congelados expresamente (`0xF`)

Por tanto, para poder realizar la modificación deseada en el fichero debemos proceder con anterioridad a solicitar una autentificación con una de las dos claves que conceden permiso de escritura, que en este caso se corresponden con las claves 0 y 1. Como ya se ha comentado antes, la clave 0 nos concedería permisos tanto de lectura como de escritura. Sin embargo, en el ejemplo que nos ocupa esto resultaría redundante ya que las operaciones de lectura pueden realizarse libremente, por lo que ambas claves estarían concediendo esencialmente los mismos derechos. No obstante, en caso de que la aplicación contenga más ficheros, esas mismas claves pueden ser utilizadas para otorgar otros derechos, como sucede en el ejemplo mostrado en la Fig. 3.5.

En el ejemplo de la Fig. 3.5 podemos ver como las diferentes claves de una aplicación pueden empleadas para garantizar diferentes derechos en los ficheros de dicha aplicación. En concreto,

- La clave maestra (`0x0`) concede derechos de lectura-escritura en los ficheros `0x01` y `0x04`, así como permite modificar la configuración del fichero `0x02`
- La clave 1 (`0x1`) concede derecho de escritura en el fichero `0x01` y de lectura en el fichero `0x02`

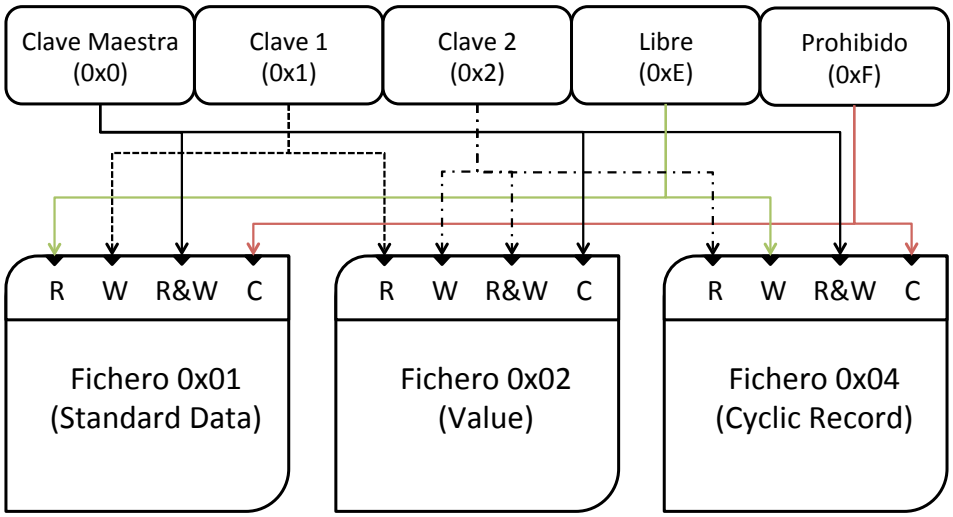


Figura 3.5: Ejemplo de asignación de derechos de acceso en una aplicación con tres claves y tres ficheros

- La clave 2 (`0x2`) concede derechos de escritura y de lectura-escritura en el fichero `0x02`, así como de lectura en el fichero `0x04`

- Las operaciones de lectura en el fichero `0x01` y de escritura en el fichero `0x04` son libres, es decir, pueden realizarse sin necesidad de una autentificación previa con ninguna clave

- Los ficheros `0x01` y `0x04` tienen su configuración congelada, lo que implica que no puede ser modificada bajo ningún concepto

Este ejemplo ilustra como una misma clave puede ser utilizada para conceder diversos derechos en varios ficheros, o incluso en un mismo fichero. De este modo, las diferentes claves pueden ser distribuidas entre diferentes tipos de usuarios, como podrían ser el emisor, el propietario y el comerciante en un sitema de tarjeta monedero, y, de ese modo, asegurar que cada parte únicamente pueda realizar aquellas acciones para las cuales tiene permiso.

En este punto sabemos que para poder modificar el contenido del fichero `0x01` de nuestro ejemplo debemos realizar una autentificación con una de las dos claves que nos otorgan permiso de escritura (`0x0` o `0x1`). Sin embargo, en caso de que dicha autentificación se realice con éxito, todavía no sabemos cómo debemos construir el mensaje que se enviará a la tarjeta con los datos que deseamos almacenar en el fichero. La manera en que se transmitirán los bytes del mensaje viene determinada por el "modo de comunicación" que haya sido establecido, aspecto que trataremos en el próximo apartado.

Modos de Comunicación

Las diversas posibilidades en las que pueden realizarse las comunicaciones con una tarjeta es uno de los puntos que más han evolucionado las Mifare DESFire EV1 con respecto a las DESFire originales. No obstante, para permitir que las versiones más recientes puedan ser utilizadas con aquellas aplicaciones diseñadas para los modelos antiguos, las tarjetas EV1 cuentan con un modo de "retrocompatibilidad" ("*backwards compatibility mode*") que es activado automaticamente cuando es necesario. Por tanto, las explicaciones que a continuación se detallarán acerca de los diferentes modos de comunicación que sean válidos para las Mifare DESFire originales también lo serán para las EV1 trabajando en "retrocompatibilidad". No sucede lo mismo en sentido contrario, lo que implica que aquellas aplicaciones que sean específicamente diseñadas para utilizar los modos de comunicación de las tarjetas EV1 no podrán trabajar con las DESFire originales.

Los modos de comunicación hacen referencia al nivel de seguridad empleado en la transmisión de los bytes de información entre PCD y PICC, o viceversa. Más concretamente, definen el grado de confidencialidad y autenticidad que se desea garantizar en las transacciones. A este respecto, las tarjetas DESFire soportan los siguientes modos de comunicación:

- **Comunicación en texto llano:** Los bytes a transmitir se envían sin ninguna transformación para garantizar su confidencialidad o su autenticidad. Esta modo de comunicación únicamente se encuentra disponible en las tarjetas Mifare DESFire originales y en las EV1 funcionando en modo de retrocompatibilidad.
- **Comunicación en texto llano con código de autentificación de mensaje (MAC):** De la misma forma que en el caso anterior, los bytes a transmitir se envían sin encriptar. Sin embargo, en este modo de comunicación a dichos datos se les concatena un código MAC de 4 bytes en el caso de las tarjetas Mifare DESFire originales o de las EV1 en modo de retrocompatibilidad, o un CMAC de 8 bytes en el caso de las EV1.
- **Comunicación con cifrado completo y código CRC:** Si este modo de comunicación se halla activo, se deberán realizar dos pasos. El primero consiste en el cálculo de un código de redundancia cíclica (CRC) de 16 bits en DESFire originales o EV1 en modo de retrocompatibilidad, o de 32 bits en las EV1, que se concatena con los datos a transmitir. Posteriormente, los bytes concatenados serán cifrados usando la clave de sesión que haya sido generada con anterioridad.

Dado que el cálculo de las secuencias MAC (y CMAC) requiere llevar a cabo operaciones de cifrado, creemos conveniente explicar primero en detalle las particularidades del último de los tres modos de comunicación disponibles.

Comunicación Cifrada Como ya se ha mencionado anteriormente, las tarjetas Mifare DESFire originales pueden realizar operaciones criptográficas tanto con cifrado DES como Triple-DES

de dos claves. La decisión de emplear un tipo de cifrado u otro viene determinada por cómo se encuentre almacenada en la tarjeta la clave con la que se ha realizado la última autentificación satisfactoria, o aquella con la que se está realizando en ese momento una autentificación. En caso de que la primera y la segunda mitad de la clave almacenada sean idénticas, el cifrado se realizará usando DES simple. En caso contrario deberá emplearse Triple-DES de dos claves. En las tarjetas EV1 se añaden dos posibilidades más: Triple-DES de tres claves y AES. En este caso deberemos recurrir al comando `Get Key Settings` para determinar qué tipo de cifrado debe emplearse en las comunicaciones con la aplicación seleccionada.

Una de las mayores particularidades de las operaciones de cifrado realizadas en las comunicaciones con tarjetas Mifare DESFire es que, para mejorar la velocidad de procesamiento de dichas operaciones, las tarjetas *siempre* realizarán operaciones de *encriptación*, tanto para enviar como para recibir datos procedentes del lector. Esto implica que en el lado del lector *siempre* se deberán realizar operaciones de *desencriptación*, tanto al enviar como al recibir datos procedentes de la tarjeta. Este comportamiento difiere de los estándares recomendados por ISO, pero se ha decidido así para minimizar el tiempo de recarga de claves. La Fig. 3.6 muestra los procesos de cifrado y descifrado de datos que debe realizar el código del lector, en el que se refleja el modo de enlazar diversos bloques de cifrado en caso de que el tamaño del mensaje supere el del método criptográfico empleado. En concreto, dichos bloques deben enlazarse mediante el modo CBC ("Cipher-Block Chaining").

No debemos olvidar que en el caso del envío de datos, con anterioridad a su cifrado es preciso computar un CRC de 16 o 32 bytes que será concatenado a los datos originales. Asimismo, si la longitud del mensaje más los 16 o 32 bytes del CRC no es múltiplo del tamaño de bloque del algoritmo criptográfico empleado, será necesario realizar un "padding" (relleno) hasta que se alcance dicha longitud.

Comunicación con MAC Dado que, como ya hemos mencionado, las tarjetas Mifare DESFire siempre realizan un proceso de *encriptación* de los datos, la generación y verificación de códigos MAC se realizará mediante la encriptación de los datos en llano, en este caso sin la inclusión de bytes de CRC. Si la comunicación se ha establecido con una tarjeta Mifare DESFire original, o bien con una EV1 funcionando en modo de retrocompatibilidad, el código MAC que se anexará a los datos en plano se corresponderá con los 4 bytes de mayor peso del criptograma generado.

El proceso incrementa en complejidad en caso de tratarse de una Mifare DESFire EV1. El código de autentificación de mensaje generado debe corresponderse en este caso a un CMAC de 8 bytes de acuerdo con la recomendación publicada por el NIST [10]. De este modo, el proceso a seguir para computar el CMAC requiere la generación de dos subclaves (`K1` y `K2`) a partir del cifrado del mensaje "todo 0's" con la clave original `K`. El uso de una u otra subclave dependerá de la longitud del mensaje del cual se desea computar su MAC, tal y como se indica en la Fig. 3.7.

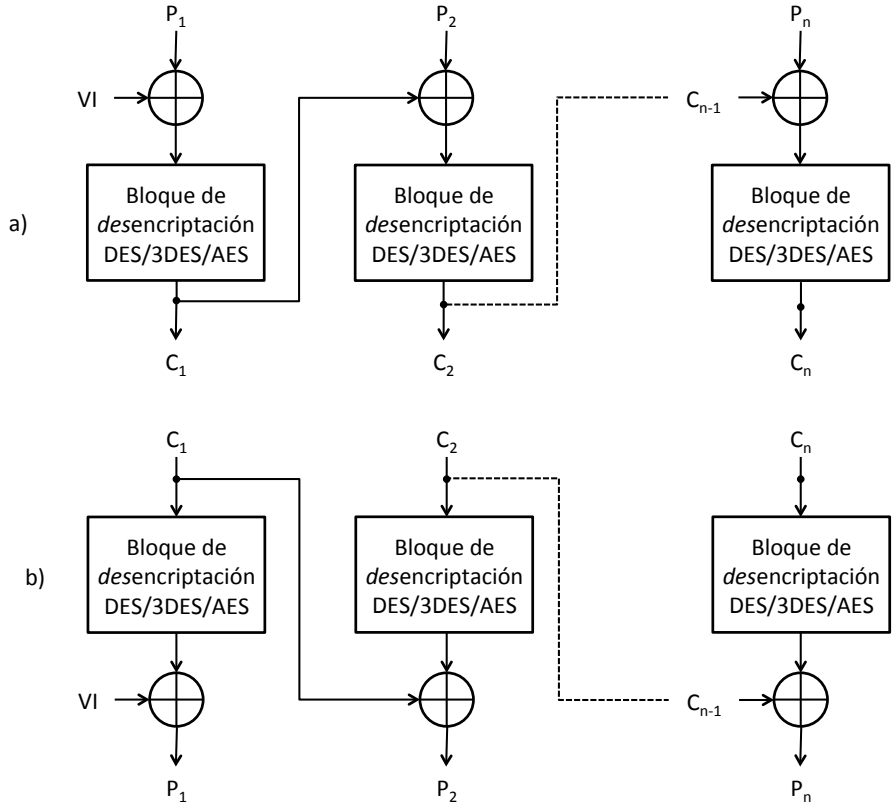


Figura 3.6: Esquema de enlazamiento CBC de bloques de cifrado en el lector en operaciones de envío (a) y recepción (b) de datos con tarjetas Mifare DESFire. Los bloques en texto llano están representados por Pi, mientras que los bloques de texto cifrado se indican con Ci. VI son las siglas de "vector inicial", y la operación de suma es una adición en módulo 2 (XOR)

Si el lector desea conocer en profundidad los detalles del cálculo del CMAC le remitimos a la recomendación citada anteriormente.

Tanto en un caso como en el otro, la inclusión de códigos MAC nos permite incrementar el grado de confianza en que el mensaje que ha sido recibido ha sido generado por una parte que conoce un secreto compartido y que, por tanto, no ha sido modificado mediante un ataque de "man-in-the-middle".

3.3.4. Manejo de Claves

Las medidas anteriormente descritas resultan de poca utilidad para garantizar la seguridad de las transacciones si el secreto de las claves empleadas es comprometido de algún modo. Como

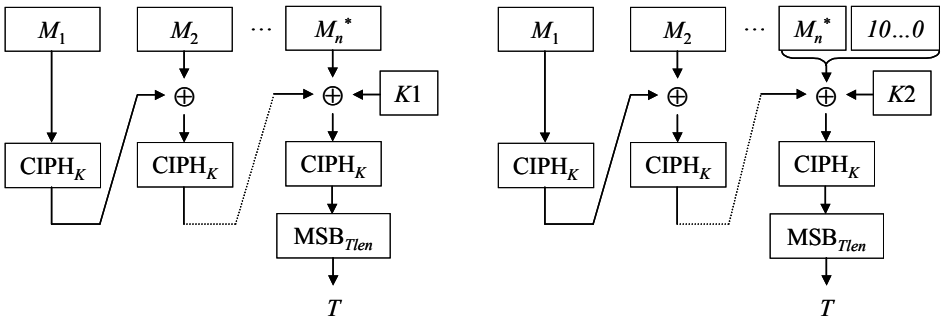


Figura 3.7: Construcción estándar códigos CMAC a partir de un mensaje M de longitud n en dos casos posibles de "padding"

ya hemos visto en la introducción de este capítulo, es posible realizar ataques que permiten obtener las claves almacenadas en tarjetas Mifare DESFire originales y, aunque todavía no ha sido reportado ningún ataque exitoso a las EV1, es una posibilidad que no puede ser descartada. Además, no debemos olvidar que en muchas ocasiones el punto débil de un sistema puede ser la máquina en la que se encuentren almacenadas las claves en el lado del lector. Aunque no es posible impedir que dichos contratiempos puedan llegar a suceder, el versionado y la diversificación de claves son herramientas que pueden ayudar a reducir el impacto que un compromiso de esa clase pueda suponer para toda una infraestructura basada en tarjetas Mifare DESFire.

Versionado de Claves

Tradicionalmente, las claves DES simple, y en consecuencia también las TDES, contenían 8 bits de paridad que se encontraban en el bit menos significativo de cada uno de los 8 bytes de los que estaban formadas. En el caso de las tarjetas Mifare DESFire, dichos bits de paridad no son necesarios para realizar una operación de confianza, por lo que pueden ser usados para guardar un número de versión que resulta de utilidad como índice en una tabla de almacenaje de claves. Al contar con 1 byte completo para tal propósito es posible distinguir entre 256 claves diferentes (0-255), siendo 0 el índice reservado para la clave por defecto.

Es importante hacer notar que incluir un bit de paridad en cada byte reduce significativamente la longitud efectiva de la clave, ya que aporta información adicional que permite facilitar ataques de fuerza bruta. Por tanto, para maximizar la seguridad del sistema es fundamental que dichos bits se sobreescriban tras la generación de las claves, teniendo entonces el uso del versionado de clave todavía más razón de ser. Esto implica que, cuando se deseen modificar las claves DES/TDES almacenadas en tarjetas MF3ICDXX, las nuevas claves deberán ser enmascaradas con una tira de bits de su misma longitud que deberá contener codificado en los bits de menor peso de cada byte el número de versión que le corresponde.

En el caso de las tarjetas Mifare DESFire EV1, debemos tener en cuenta que el uso de

cifrado AES también es posible y, en este tipo de cifrado no hay bits de paridad que puedan ser utilizados para almacenar la versión de clave. Por esa razón el comando `Change Key` permite en las tarjetas EV1 que se indique como parámetro independiente la versión a emplear si la clave es AES.

Para recuperar el número de versión de una clave almacenada en una tarjeta MF3ICDXX debemos utilizar el comando `Get Key Version`. Este paso resulta fundamental para realizar una autentificación satisfactoria con dicha clave, ya que nos permite determinar qué entrada de la tabla de claves local debe ser empleada para realizar las operaciones criptográficas. Además, en caso de que un cierto número de claves se hayan visto comprometidas, puede resultar de utilidad para comprobar si las claves almacenadas en una PICC concreta requieren ser actualizadas para preservar su seguridad.

Diversificación de Claves

En muchas ocasiones resulta interesante que cada PICC cuente con su propia tabla de claves. Sin embargo, mantener un registro de todas las claves posibles para todas las tarjetas que cuentan con autorización para ejecutar una determinada aplicación sería extremadamente costoso. Para evitar este inconveniente, los diversos métodos de diversificación de claves derivan cada una de las claves para cada tarjeta a partir de una clave base utilizando información única para cada una de ellas. De ese modo, cada tarjeta recibe un valor diferente para cada clave, por lo que si alguna de ellas es comprometida por alguna causa, la vulnerabilidad queda limitada a dicha clave en esa tarjeta exclusivamente, evitando su extensión a toda la infraestructura.

Habitualmente se utilizan dos métodos diferentes para obtener claves diversificadas para tarjetas MF3ICDXX. El primero de ellos, basado exclusivamente en el UID de la tarjeta, es muy sencillo y únicamente se recomienda en caso de emplear claves DES simple. Para el resto de algoritmos criptográficos resulta mucho más conveniente el uso del método basado en CMAC.

Método UID El método más sencillo para particularizar las claves a emplear según la tarjeta con la que se produzca la comunicación es el que en su momento fue recomendado por Philips para las tarjetas Mifare DESFire originales. Este método, sin embargo, no se encuentra actualmente recomendado y únicamente debería usarse en el improbable caso de que se utilicen claves DES simple. El proceso a seguir es el siguiente:

1. Se concatena el byte denominado “Cascade Tag” (`0x88`) con el UID obtenido de la tarjeta. La secuencia total tendrá una longitud de 8 bytes.

2. La secuencia obtenida anteriormente se usará como vector inicial en la encriptación DES de la clave que se desea diversificar con ella misma.

3. La concatenación de las salidas de los bloques DES constituirán la nueva clave diversificada.

4. Finalmente, no debe olvidarse sobreescribir los bits de paridad de la nueva clave con su versión, tal y como se ha explicado en el apartado anterior.

De este modo, las claves habrán sido modificadas de tal modo que al ser almacenadas en cada tarjeta serán completamente diferentes entre ellas y no se pondrá en compromiso a la infraestructura completa en caso de que una tarjeta particular hubiera sido atacada con éxito.

Método CMAC Con la introducción de las tarjetas Mifare DESFire EV1 y su compatibilidad con criptografía Triple-DES de tres claves y AES, NXP pasó a recomendar el empleo del CMAC que ha sido explicado anteriormente como método para diversificar las claves. Sin embargo, en este caso el CMAC no se usaría con fines criptográficos, sino como función de un solo sentido. El proceso más básico se haya esquematizado en el diagrama de bloques de la Fig. 3.8, aunque en realidad para cada tipo de clave la estructura es algo diferente. En concreto, según el tamaño de bloque del algoritmo criptográfico y la longitud de la clave a diversificar, es posible que se requieran varias ramas de ese mismo proceso actuando sobre subclaves de la original. En ese caso, la clave diversificada se generará por la concatenación de las salidas obtenidas para cada una de las subclaves. Los detalles de los algoritmos concretos a emplear para cada tipo de clave pueden ser encontrados en [18]. El proceso básico sigue los siguientes pasos:

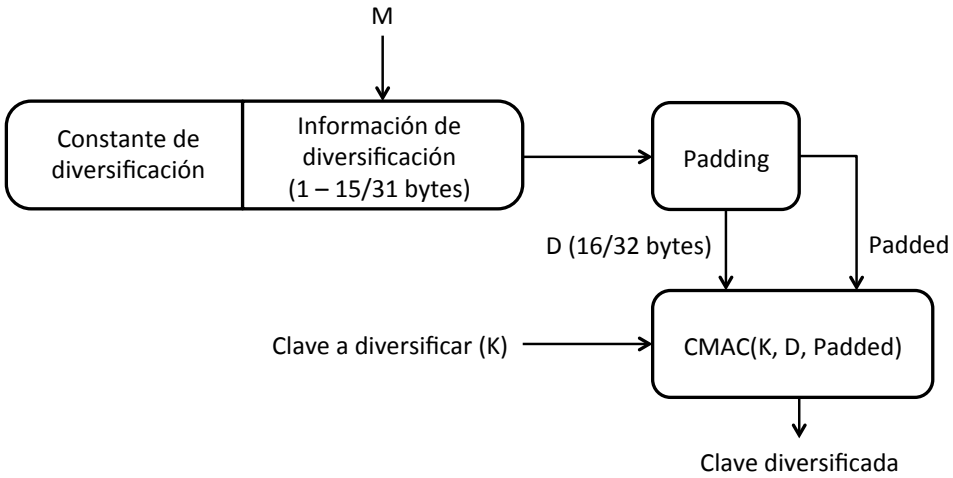


Figura 3.8: Diagrama de bloques simplificado del proceso recomendado para la diversificación de claves usando CMAC

1. Se construye una información de diversificación (`M`) que incluya, al menos, elementos que describan unívocamente a la tarjeta (o grupo de tarjetas) para la que se desea realizar la diversificación de clave. Para ello, lo más habitual es utilizar el UID de la tarjeta, al que se le puede concatenar el número de clave y en número de aplicación, por ejemplo. Esto provoca que una misma clave sea almacenada de formas diferentes dentro de varias

aplicaciones en una misma tarjeta, o incluso en una misma aplicación de diversas formas, aumentando la dificultad de los ataques manteniendo el espacio de almacenaje de claves en local reducido.

2. A `M` se le añade un prefijo de un byte denominado "constante de diversificación" que depende del algoritmo criptográfico al que corresponda la clave.

3. La cadena resultante es rellenada hasta alcanzar un tamaño de 16 bytes en el caso de claves Triple-DES, o de 32 bytes en el caso de claves AES, construyendo el vector `D`. En caso de que haya sido necesario dicho relleno, el flag `Padded` toma el valor `true`, mientras que será `false` si ya tenía el tamaño adecuado.

4. La clave a diversificar `K`, o una de sus subclaves, es usada como entrada, juntamente con `D` y `Padded`, a una función CMAC que usa como bloque criptográfico el algoritmo correspondiente a la clave concreta.

5. La salida de la función CMAC, o la concatenación de las diversas funciones CMAC si ha sido necesario el empleo de subclaves, será la clave diversificada a almacenar en la tarjeta.

Un punto importante que debe tenerse en cuenta es que las tarjetas Mifare DESFire EV1, como ya hemos mencionado anteriormente, permiten modificar su configuración para devolver un identificador aleatorio en lugar del UID al ejecutar su comando `Get UID`. Esto implica que no siempre será posible la obtención de información que permita la diversificación de las claves de forma sistemática en este tipo de tarjetas.

3.4. Conclusión

Las tarjetas Mifare DESFire y Mifare DESFire EV1 ofrecen un amplio abanico de posibles operaciones, así como un sistema de archivos flexible y un conjunto de medidas de seguridad que ofrece un elevado grado de confianza. Sin embargo, todas estas características tienen escasa relevancia sin una plataforma que permita aprovecharlas para crear aplicaciones que pongan al servicio del usuario todo su potencial. Los próximos capítulos estarán dedicados, por tanto, a describir el desarrollo que se ha llevado a cabo para conseguir dicho fin.

4 | Infraestructura empleada

Para lograr nuestro objetivo de implementar una librería que nos permita crear aplicaciones que aprovechen las funcionalidades de las tarjetas Mifare DESFire y Mifare DESFire EV1 resulta imprescindible disponer de una infraestructura adecuada. En este capítulo describiremos a grandes rasgos los elementos esenciales que constituyen la infraestructura que ha sido empleada durante el desarrollo del presente proyecto, tanto a nivel de software como de hardware. Comenzaremos indicando sobre qué sistema operativo ha sido construida nuestra librería, ya que dicha elección puede condicionar su utilidad futura. Con posterioridad se comentarán las librerías de nivel intermedio desarrolladas por terceros que han sido aprovechadas en el presente proyecto, haciendo especial hincapié en las especificaciones elaboradas por el PC/SC Workgroup y su implementación en código libre `pcsc-lite`. Finalmente, mencionaremos los distintos dispositivos lectores que han sido testados con mayor o menor éxito durante el proceso de elaboración de la librería.

En la Tabla 4.1 se resumen las versiones del software que ha sido empleado durante la elaboración de la versión más reciente de la librería en el momento de escribir la presente memoria. A efectos de una posterior utilización de la librería para el desarrollo de aplicaciones por parte de terceros, consideramos que las versiones indicadas ahí ofrecen las garantías suficientes para su correcto funcionamiento, sin perjuicio de que versiones posteriores puedan resultar igualmente válidas.

Tabla 4.1: Software empleado en la elaboración del proyecto

Funcionalidad	**Recurso**	**Versión**
Sistema Operativo	Ubuntu Linux	`14.04`
Plataforma Java	Open JDK	`7`
Implementación PC/SC	`pcsc-lite`	`1.8.13`
Controladores USB	`ccid`	`1.4.18`
Interfaz Java PC/SC (librería externa)	`jpcsc`	`0.8.0`
Interfaz Java PC/SC (librería estándar)	`javax.smartcardio`	-

Adicionalmente, también se incluyen en el Apéndice A las instrucciones de instalación que

consideramos que pueden ser útiles para aquellos que deseen utilizar la librería desarrollada en este proyecto.

4.1. Sistema Operativo

Idealmente, la librería a desarrollar debería resultar transversal con respecto al sistema operativo sobre el que se empleara. De esta forma se podría ofrecer a los usuarios un entorno de desarrollo de aplicaciones que les permitiera implementar sus servicios de forma agnóstica al sistema operativo en el que se fuera a ejecutar, pudiendo llegar así a un público lo más amplio posible. Sin embargo, la realidad es que el alcance del presente proyecto nos fuerza a limitar su ámbito de aplicación a un entorno determinado, por lo que su correcta elección es fundamental.

Los sistemas Windows son, sin ninguna duda, los más extendidos entre el gran público[1], por lo que su uso podría estar justificado en este sentido. Además, como veremos más adelante en la Sección 4.2, las últimas versiones de los sistemas operativos de Microsoft incluyen de forma nativa soporte para la comunicación con tarjetas inteligentes. Estas características podrían impulsar a considerar Windows como el candidato ideal para nuestros propósitos. Sin embargo, desde un principio se consideró esencial que la librería pudiera ser construida respetando los principios del software libre, para que la comunidad pudiera aprovecharla libremente para el desarrollo de aplicaciones. Este factor resulta absolutamente incompatible con la dependencia de la librería de un sistema operativo propietario como Windows, por lo que su candidatura fue rápidamente descartada a pesar de los beneficios anteriormente mencionados. Este problema podría extenderse también a los sistemas basados en el Mac OS X de Apple, añadiendo, asimismo, la dependencia de un único fabricante.

La alternativa que más se ajusta a nuestras pretensiones es el uso de una de las diversas distribuciones de Linux. El número de variantes que pueden encontrarse es enorme[2], pero de entre ellas cabe destacar Ubuntu[3] como una de los más populares en entornos de escritorio[4]. Esta combinación de libertad en el código y relativa popularidad resultó clave para la elección de Ubuntu como plataforma ideal para el desarrollo del proyecto. Basado en Debian y con un entorno de escritorio orientado a usuarios noveles e intermedios, Ubuntu puede utilizarse en una variedad de dispositivos. Un rasgo a destacar de esta distribución es que se lanzan nuevas versiones de forma periódica, incluyendo nuevas características y funcionalidades. Debemos tener en cuenta que, aunque esto es habitualmente beneficioso, en ocasiones resulta problemático realizar la actualización debido a posibles incompatibilidades entre el software/hardware que el usuario venía utilizando y los cambios introducidos en la nueva versión. Esto nos sucedió a nosotros al actualizar a la versión `12.04`, provocando que debiéramos realizar trabajo no previsto

[1] http://en.wikipedia.org/wiki/Usage_share_of_operating_systems
[2] http://en.wikipedia.org/wiki/List_of_Linux_distributions
[3] http://www.ubuntu.com/
[4] http://www.zdnet.com/article/the-five-most-popular-end-user-linux-distributions/

para hacer funcionar de nuevo nuestro código. Sin embargo, actualizaciones posteriores no han supuesto ningún problema, por lo que ha podido comprobarse su correcto funcionamiento en la última versión lanzada con soporte a largo plazo (`14.04`).

4.2. Middleware

La principal meta del presente proyecto es proporcionar una librería software que permita el desarrollo de aplicaciones de usuario que hagan uso de tarjetas sin contacto de la familia Mifare DESFire. Sin embargo, dicha meta no es factible si no se dispone de un método fiable para establecer diálogos entre el sistema operativo y los periféricos que actuarán como dispositivos lectores/escritores de las tarjetas. El desarrollo del middleware, o capas de nivel intermedio, que permiten realizar esos diálogos queda fuera del alcance del presente proyecto. Afortunadamente se encuentran disponibles librerías de terceros que podemos aprovechar para tal fin, y sobre las que se construirá la nuestra.

4.2.1. Especificaciones del PC/SC Workgroup

La librería de código abierto `pcsc-lite`[5] desarrollada en el contexto del proyecto M.U.S.C.L.E.[6] se concibió como una adaptación de las especificaciones publicadas por el PC/SC Workgroup[7]. Este grupo de trabajo se creó con la finalidad de desarrollar unas recomendaciones que faciliten la interoperabilidad necesaria para el uso integrado de tarjetas inteligentes en ordenadores personales. De esa forma, se definió una API de programación estándar que ha sido incorporada en los sistemas operativos de Microsoft en la librería `winscard.dll`. Este formato, sin embargo, no es válido en otros sistemas operativos, por lo que mediante la implementación de la librería `pcsc-lite` se pretendía emular el comportamiento de WinSCard en sistemas basados en Unix, como Linux y Mac OS X. La inclinación hacia los sistemas operativos Windows se debe a la participación de Microsoft que, junto con otras grandes compañias como Hewlett-Packard, IGM, Intel o Sun Microsystems, se unió a empresas especializadas en el sector como Bull CP8 o Gemplus para formar el PC/SC Workgroup. Sin embargo, a día de hoy la composición del grupo de trabajo ha variado enormemente y muy pocos de los miembros fundadores se mantienen en él, de acuerdo con lo que indica su página web[8]. De entre los miembros actuales cabría destacar NXP, escisión de Philips encargada del desarrollo de las tarjetas sin contacto de la familia Mifare, y los fabricantes de dispositivos lectores ACS[9] y Gemalto[10] (antigua Gemplus).

[5] https://pcsclite.alioth.debian.org/

[6] Movement for the Use of Smart Cards in a Linux Environment. https://pcsclite.alioth.debian.org/musclecard.com/

[7] http://www.pcscworkgroup.com/

[8] http://www.pcscworkgroup.com/membership/members.php

[9] http://www.acs.com.hk/

[10] http://www.gemalto.com/

Las especificaciones PC/SC fueron publicadas por primera vez en 1997, siendo actualizadas a la versión 2.0 en 2004. En el momento de escribir esta memoria, la versión más reciente que se encuentra disponible es la 2.01.14, que data de junio de 2013. En su origen fueron creadas a partir de los estándares industriales preexistentes relativos a tarjetas inteligentes, principalmente ISO 7816 y EMV, que fueron complementados mediante la definición de interfaces de bajo nivel para dispositivos de lectura/escritura con tarjetas, así como interfaces de programación de aplicaciones (APIs) independientes del dispositivo empleado. También incluye herramientas para la gestión de recursos que permiten que múltiples aplicaciones compartan los dispositivos que se encuentren conectados al sistema.

El principal rasgo que hace esta especificación especialmente interesante para nuestros intereses es que dedica gran parte de su documentación a establecer qué características deben cumplir los dispositivos de lectura/escritura (IFDs o *InterFace Device*) de tarjetas inteligentes, tanto con contacto como sin contacto. De ese modo, cualquier librería construida a través de la API proporcionada será compatible con todo dispositivo que cumpla dicha especificación. Actualmente, la gran mayoría de dispositivos que se comercializan son compatibles con PC/SC aunque, como veremos más adelante en la Sección 4.3 al describir el hardware empleado, no siempre es así.

El PC/SC Workgroup ha publicado sus especificaciones divididas en diez partes que tratan sobre los distintos aspectos que deben ser regulados:

Part 1: Introduction and Architecture Overview
Proporciona una visión general de la arquitectura de los sistemas PC/SC y los componentes que la norma define.

Part 2: Interface Requirements for Compatible IC Cards and Readers
Detalla las características y los requisitos de interoperabilidad que deben cumplir tanto las tarjetas como los lectores compatibles con las especificaciones.

Part 3: Requirements for PC-Connected Interface Devices
Describe el interfaz y la funcionalidad requerida para los IFD compatibles. Asimismo, incluye un suplemento que proporciona información acerca de números RID (*Registered Application Provider Identifier*) para su uso en las tramas ATR (*Answer to Reset*).

Part 4: IFD Design Considerations and Reference Design Information
Discute una serie de consideraciones de diseño relativas a los IFD. En particular, proporciona una implemencación recomendada para aquellos dispositivos con teclados PS/2 integrados.

Part 5: ICC Resource Manager Definition
Describe los interfaces y la funcionalidad soportada por el Administrador de Recursos ICC (*ICC Resource Manager)*, un componente que se require a nivel de sistema.

Part 6: ICC Service Provider Interface Definition
Describe el modelo de proveedor de servicios ICC (*ICC Service Provider)*, identificando interfaces requeridas e indicando cómo puede ser extendido para satisfacer determinados requisitos específicos del dominio de la aplicación.

Part 7: Application Domain/Developer Design Considerations
Describe una serie de consideraciones de diseño para desarroladores de aplicaciones, proporcionando sugerencias de uso del resto de componentes.

Part 8: Recommendations for ICC Security and Privacy Devices
Describe la funcionalidad recomendada para ICCs que pretenden soportar los requisitos de criptografía de propósito general y almacenaje. Se encuentra orientado al soporte de los estándares de seguridad y privacidad tanto en Internet como en computadoras.

Part 9: IFDs with Extended Capabilities
Describe la gestión de IFDs que incluyen capacidades extendidas, tales como la entrada segura de PIN o funcionalidad de interfaz de usuario.

Part 10: IFDs with Secure Pin Entry Capabilities
Describe en mayor detalle la gestión de IFDs con capacidades de entrada segura de códigos PIN.

En dichas especificaciones se propone una arquitectura como la que se muestra en la Fig. 4.1, en la que se indica en qué parte de las anteriormente comentadas se encuentran las recomendaciones correspondientes a cada elemento que la compone. En este contexto, nuestro objetivo es proporcionar un interfaz de las capas superiores del esquema, adecuado para las tarjetas de la familia Mifare DESFire, dejando en manos de una implementación específica la gestión de los elementos de las capas de más bajo nivel. El punto clave de esta arquitectura es el denominado Administrador de Recursos del ICC, ya que:

- Es el responsable de la identificación y rastreo de recursos, incluyendo:
  - Rastrear los IFDs instalados y presentar información respecto a ellos a las demás aplicaciones
  - Rastrear los tipos de ICC conocidos, junto con los proveedores de servicios asociados e interfaces que soporta
  - Rastrear los eventos de inserción y retiro de ICCs para mantener información exacta sobre los ICCs que se encuentran en un determinado momento insertados en los IFDs
- Se encarga de gestionar los recursos disponibles en entornos multiaplicación

La relevancia de estas funciones es evidente, por lo que resulta fundamental disponer de una implementación fiable. Es importante mencionar que la coexistencia en un mismo sistema de

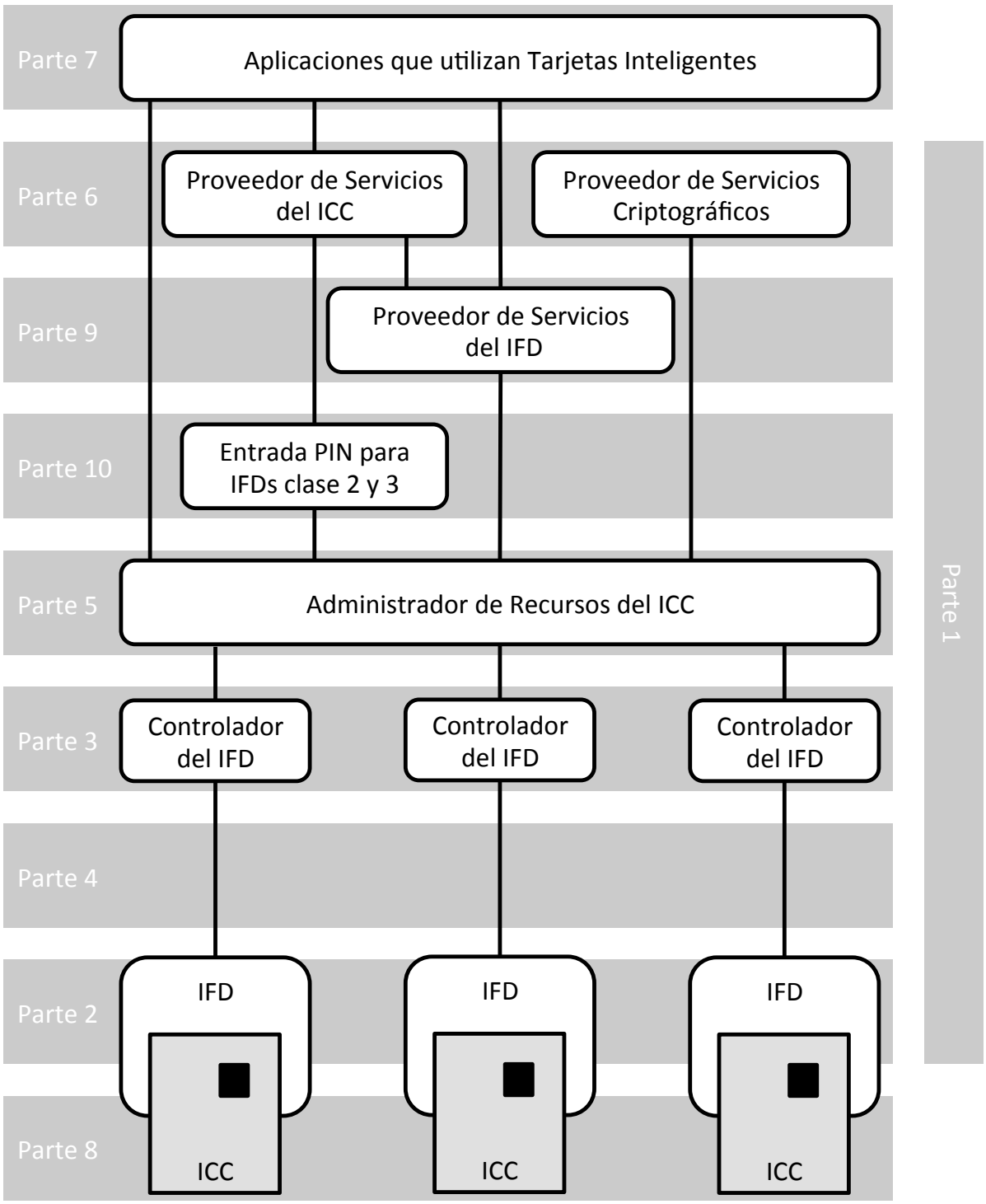


Figura 4.1: Arquitectura propuesta en las especificaciones del PC/SC Workgroup, con la parte de la documentación correspondiente a cada componente (Figura adaptada de la web del PC/SC Workgroup[a])

[a] http://www.pcscworkgroup.com/specifications/overview.php

distintos administradores de recursos puede causar conflictos que provoquen malfuncionamientos, como se menciona en las especificaciones y nosotros mismos hemos experimentado con distintas versiones de una misma librería. Esto hace más crítica, si cabe, la elección de una implementación determinada. Como ya se ha mencionado anteriormente, en los sistemas Windows se encuentra preinstalada la librería `winscard.dll`, que proporciona dichas funcionalidades y, por tanto, evita la necesidad de buscar distintas implementaciones. Sin embargo, en los sistemas basados en Unix es necesario acudir a implementaciones no oficiales, y de entre las disponibles destaca `pcsc-lite`, que se ha erigido como el estándar *de facto*, por lo que resulta la opción más adecuada para nuestros objetivos.

4.2.2. `pcsc-lite`

Desarrollada inicialmente por David Corcoran y lanzada en su primera versión en el año 1999, la librería de código abierto `pcsc-lite` es hoy en día la adaptación de las especificaciones PC/SC más habitualmente empleada en sistemas Unix. De hecho, Apple incluye desde 2002 en sus sistemas operativos una versión de `pcsc-lite` basada en el código inicialmente creado por Corcoran, adaptada para satisfacer mejor sus propias necesidades. Incluso Oracle tiene su propia versión para los sistemas Solaris. Actualmente, el desarrollador principal y mantenedor de la versión genérica es Ludovic Rousseau[11].

La librería `pcsc-lite` es una implementación de la API de alto nivel descrita en las especificaciones PC/SC para el acceso estandarizado a tarjetas inteligentes y sus lectores. Escrita en ANSI C, se desarrolló para que pudiera ser usada con la mayoría de compiladores existentes en Unix, y se evitó el uso de estructuras de datos grandes y complejas para que resultara lo más ligera posible. Su compliación genera un archivo `libpcsclite.so`, que debe ser enlazado en las aplicaciones o librerías de capa superior que quieran acceder a sus funcionalidades.

Las funciones definidas en la API son básicamente las mismas que se pueden encontrar en la librería `winscard.dll` en Windows, implementadas de forma que sus llamadas resulten compatibles con las de la API de Microsoft. De esta forma es posible la comunicación con los lectores de una manera homogénea. El lector interesado en conocer en mayor profundidad las funciones que ofrece `pcsc-lite` puede acudir a la documentación que se encuentra en la página web de la librería[12].

Aparte de las funciones incluidas en la interfaz de programación, la librería también proporciona un Administrador de Recursos como el definido en la parte 5 de las especificaciones PC/SC. Este Administrador de Recursos toma la forma de un *daemon* denominado `pcscd` que se ejecuta en segundo plano, encargándose del manejo de peticiones de cliente (conexiones, tansmisiones, etc.) y de la sincronización del acceso desde múltiples aplicaciones. La presencia del *daemon* en nuestro sistema, además, nos alerta de la existencia de una versión de la librería `pcsc-lite`

[11] http://ludovicrousseau.blogspot.fr/
[12] http://pcsclite.alioth.debian.org/api/index.html

ya instalada. En algunas ocasiones, como sucede en los sistemas Mac OS X, una versión de `pcsc-lite` puede encontrarse preinstalada. No es extraño que se produzca un conflicto entre el Administrador de Recursos y el interfaz si se detectan diversas versiones de la librería. Por ello, resulta conveniente comprobar si esto sucede en nuestro caso antes de proceder a instalar una nueva versión. En concreto, si ejecutando el comando "`which pcscd`" en la terminal obtenemos una ruta al ejecutable, es aconsejable asegurarnos de realizar una desinstalación completa de `pcsc-lite` antes de realizar la actualización.

El papel del Administrador de Recursos resulta fundamental para que puedan producirse las comunicaciones deseadas con tarjetas inteligentes y sus lectores, por lo que es recomendable comprobar que se encuentra en ejecución (por ejemplo, mediante "`pgrep pcscd`") antes de lanzar nuestras aplicaciones. Es importante comentar que en las versiones más recientes de `pcsc-lite`, `pcscd` no se ejecuta al inicio ya que no se le han concedido permisos de superusuario, por lo que es necesario hacerlo manualmente desde la terminal. Alternativamente, pueden seguirse las instrucciones recomendadas por Ludovic Rousseau en su blog[13] para que `pcscd` sea lanzado de forma automática en nuestro sistema.

Finalmente, también debemos mencionar que, aparte de la implementación de las especificaciones PC/SC, el mismo autor también mantiene unos controladores de código abierto basados en las especificaciones CCID[14] del USB Workgroup para dispositivos lectores de tarjetas inteligentes. Estas especificaciones pretenden unificar los controladores de los distintos fabricantes para que puedan ser manejados mediante un único software. Mientras que en Windows se encuentran soportados de forma nativa desde hace varias versiones, en los sistemas Unix se recomienda instalar la versión compatible con `pcsc-lite`[15]. Además, para asegurar que los controladores permiten la correcta comunicación con los dispositivos a través del *daemon* `pcscd` es conveniente realizar primero la instalación de `pcsc-lite` y posteriormente la de `ccid`, tal y como se indica en el Apéndice A.

4.2.3. Interfaz Java

Aunque el interfaz C que nos ofrece `pcsc-lite` sería suficiente para su uso en la creación de aplicaciones, ya que podríamos generar una librería de capa superior en el mismo lenguaje especialmente dedicada a la comunicación con tarjetas Mifare DESFire, a día de hoy resulta mucho más habitual el desarrolo de aplicaciones en lenguajes de una mayor abstracción. Uno de los más populares es, sin duda, Java, en el que a través del framework JNI[16] es posible la llamada a código implementado en C.

[13] http://ludovicrousseau.blogspot.com.es/2010/12/configuring-your-system-for-pcscd-auto.html

[14] Chip Card Interface Device. http://www.usb.org/developers/docs/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf

[15] https://pcsclite.alioth.debian.org/ccid.html

[16] Java Native Interface. http://docs.oracle.com/javase/8/docs/technotes/guides/jni/

4.2.3.1. jpcsc

Con el objetivo de proporcionar a los desarrolladores interfaces de programación en los lenguajes más habitualmente empleados para la implementación de aplicaciones, en el contexto del proyecto M.U.S.C.L.E. se crearon distintas capas de nivel superior que se comunicaban con la API escrita en ANSI C. Entre ellas se podía encontrar `jpcsc`, desarrollada por Marcus Oestreicher, que ofrecía abstracciones de las funcionalidades de `pcsc-lite` para Java, a través de una capa JNI.

La librería `jpcsc` ofrece una API Java para acceder a la mayoría de las funciones relacionadas con las tarjetas y el contexto en entornos PC/SC, soportando nativamente Windows y Linux. Proporciona, por ejemplo, funciones para el establecimiento del contexto, el listado de lectores conectados, la obertura de una comunicación o el envío de tramas en formato APDU a un lector o una tarjeta. Por contra, no ofrece herramientas para el acceso a nivel de aplicación ni para tipos particulares de tarjetas. Ese es, de hecho, el objetivo de nuestra propia librería, que se construye encima suyo.

El paquete en el que se distribuye la librería (`com.linuxnet.jpcsc`) se compone básicamente de dos clases, `Context` y `Card`, así como otras clases e interfaces auxiliares. Mediante las funcionalidades incluidas en `Context` es posible establecer una comunicación con el *daemon* `pcscd`, interrogar a lectores o configurar una conexión con una tarjeta. Esto deriva en una instancia de la clase `Card`, que se emplea para intercambiar tramas APDU con la tarjeta en cuestión. Los métodos de ambas clases se corresponden casi directamente con las funciones nativas de `pcsc-lite` a las que encapsulan. Adicionalmente, los errores que puedan aparecer son, en general, señalados mediante excepciones de tipo `PCSCException`. La clase `State` encapsula estructuras para el retorno del estado de las conexiones con lectores y tarjetas en métodos como `Context.GetStatusChange()` o `Card.Status()`. Finalmente, la clase `PCSC` define una serie de constantes cuyos valores son dependientes de la plataforma subyacente, sea Linux (Unix) o Windows, por lo que deben ser mapeados de forma adecuada en una fase de inicialización especialmente dedicada.

Si se desea un mayor detalle en cuanto al contenido y funcionalidad que pueden obtenerse de `jpcsc`, el archivador que se redistribuye junto con nuestra librería contiene la documentación original en formato HTML, así como los ficheros de texto con la información necesaria para su compilación[17]. La decisión de distribuir `jpcsc` junto con nuestra propia librería se debe al hecho que desde hace varios años ha dejado de recibir soporte, encontrándose rotos los enlaces para su descarga en la página de M.U.S.C.L.E. dedicada a los interfaces a otros lenguajes[18]. Por lo que hemos podido averiguar, en la actualidad se recomienda el uso del paquete `javax.smartcardio`[19]

[17] El archivo `README` que se incluye es el que se distribuía con la versión `0.8.0` de la librería `jpcsc`. Sin embargo, como podrá observarse, en realidad se trata del correspondiente a la versión `0.7.7`, o incluso la `0.7.6` de acuerdo a su *changelog*, ya que no fue actualizado por el propio autor.

[18] https://pcsclite.alioth.debian.org/musclecard.com/middle.html

[19] http://docs.oracle.com/javase/6/docs/jre/api/security/smartcardio/spec/javax/smartcardio/

para tal propósito, que será tratado en el siguiente apartado. De todas formas, se ha comprobado que este paquete Java no permite el envío de comandos en forma nativa, por lo que no resulta compatible con las tarjetas Mifare DESFire originales. Por ello se ha decidido proporcionar el código fuente de `jpcsc` junto con nuestra propia librería, aprovechando el hecho de que la licencia que acompaña a la interfaz permite su libre redistribución. Debe mencionarse que las cabeceras de las funciones incluidas en el módulo `winscard.c` de `pcsc-lite` han cambiado desde que se lanzó la versión más reciente de `jpcsc`, por lo que actualmente no es posible su correcta recompilación. Por ello se proporciona el archivo de librería dinámica `libjpcsc.so` precompilado, pero es posible que no sea compatible con determinados sistemas, especialmente los de 64 bits.

4.2.3.2. `javax.smartcardio`

Como se ha mencionado anteriormente, en la actualidad se recomienda el uso del paquete `javax.smartcardio` para el establecimiento de comunicaciones en Java con tarjetas inteligentes a través de lectores compatibles con las especificaciones PC/SC. Este paquete se distribuye con la versión estándar de Java desde el JRE 1.6 y ofrece un interfaz que permite realizar las comunicaciones de un modo similar al de su antecesor `jpcsc`, aunque en este caso su API no es una traslación directa de los métodos que pueden encontrarse en `winscard`/`pcsc-lite`.

La principal diferencia con la implementación `jpcsc` comentada anteriormente es que no se utiliza el concepto de Contexto, sino que se accede directamente a los dispositivos lectores conectados al sistema través de una "factoría de terminales":

```
1 TerminalFactory factory = TerminalFactory.getDefault();
2 List<CardTerminal> terminals = factory.terminals().list();
3 CardTerminal terminal = terminals.get(0);
```

Una vez seleccionado el terminal con el cual se trabajará, se crea un objeto de clase `Card` al realizar la conexión, estableciendo el protocolo adecuado (`"T=0"`, `"T=1"` o `"T=CL"`) o permitiendo el uso de cualquiera que se encuentre disponible (`"*"`). Asimismo, para poder realizar la transmisión es necesario obtener un canal de comunicación:

```
1 Card card = termninal.connect("*");
2 CardChannel channel = card.getBasicChannel();
```

Una de las características más relevantes de la API es que únicamente permite enviar tramas de bytes que sigan la estructura APDU definida en la norma ISO/IEC 7816-4. Esto en la práctica implica que no es posible la transmisión de tramas de longitud inferior a 4 bytes, lo que impide que puedan enviarse muchos de los comandos DESFire en modo nativo. Mientras que en las tarjetas Mifare DESFire EV1 (MF3ICDX1) esto no supone un problema, ya que la comunicación con este tipo de PICCs puede realizarse de forma sencilla insertando los comandos dentro de tramas APDU, el modo *wrapped* de las tarjetas Mifare DESFire originales (MF3ICD40) no

`package-summary.html`

puede emplearse si se desea que la librería también pueda soportar las nuevas funcionalidades introducidas por las EV1.

```
1 ResponseAPDU answer = channel.transmit(new CommandAPDU(command));
```

Finalmente, es necesario realizar la desconexión con la tarjeta:

```
1 card.disconnect(false);
```

4.3. Hardware

Aparte de las tarjetas sin contacto cuyas características comentamos en el Capítulo 3, para poder sacar provecho de la librería implementada es imprescindible disponer de un dispositivo lector apropiado. A lo largo del proceso de desarrollo del presente proyecto se han empleado diversos dispositivos lectores de tarjetas sin contacto. Esto es debido al hecho de que, por desgracia, no todos los dispositivos se construyeron de acuerdo a las especificaciones PC/SC y/o CCID que hemos mencionado en la Sección 4.2. Afortunadamente, cada vez son más los lectores que soportan de forma nativa los requisitos indicados en dichas especificaciones, por lo que si el lector dispone de un dispositivo adquirido en los últimos años o si piensa adquirirlo en un futuro próximo es muy probable que sea perfectamente compatible. En todo caso, es recomendable revisar el listado incluido en la página oficial de `pcsc-lite`[20] antes de adquirir un nuevo dispositivo para asegurar su compatibilidad. Por contra, aquellos que no aparezcan en la lista podrian resultar igualmente compatibles, pero no podría asegurarse hasta su comprobación de forma individual.

Para comprobar si un lector es compatible con las especificaciones se puede recurrir a la aplicación `pcsc_scan` incluida en las `pcsc-tools`, cuya instalación se comenta en el Apéndice A. En la Fig. 4.2 podemos ver cómo se comporta esta sencilla aplicación cuando es ejecutada con dispositivos previamente conectados al equipo. Como se puede comprobar, la aplicación reconoce como dos lectores diferentes las dos ranuras de las que dispone el dispositivo Gemalto Prox-DU, como veremos un poco más adelante. Sin embargo, debemos mencionar que en el momento de hacer la captura de pantalla también se hallaban conectados al equipo los otros dos dispositivos que se trataron de hacer funcionar, un ACS ACR122U y un Philips Mifare Pegoda. En ambos casos la aplicación ni siquiera detecta su presencia, lo que nos indica que o bien no siguen las especificaciones PC/SC o bien su firmware no es compatible con los controladores CCID.

Como hemos mencionado arriba, de los tres lectores de los que se ha dispuesto durante el proceso de elaboración del proyecto, dos han resultado no ser compatibles con el middleware requerido. En el caso del dispositivo Philips Mifare Pegoda MFRD700 (ver Fig. 4.3a), cuya versión actual se denomina NXP Mifare Pegoda CLRD710, este comportamiento era de esperar,

[20] http://pcsclite.alioth.debian.org/ccid/supported.html

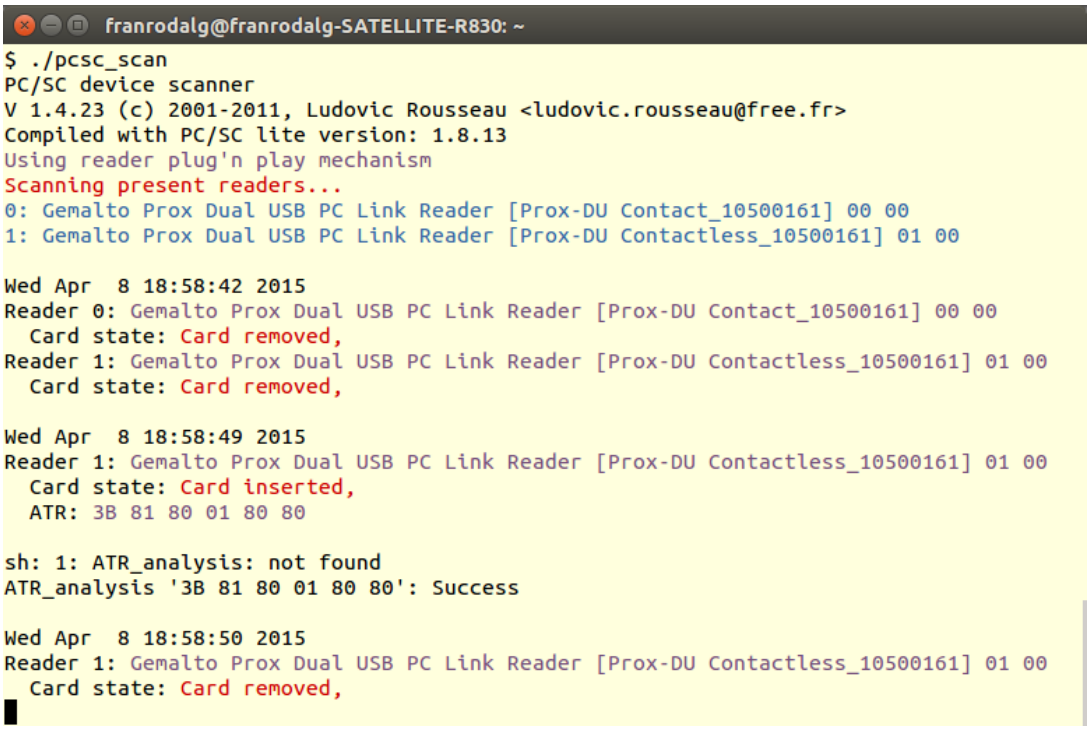


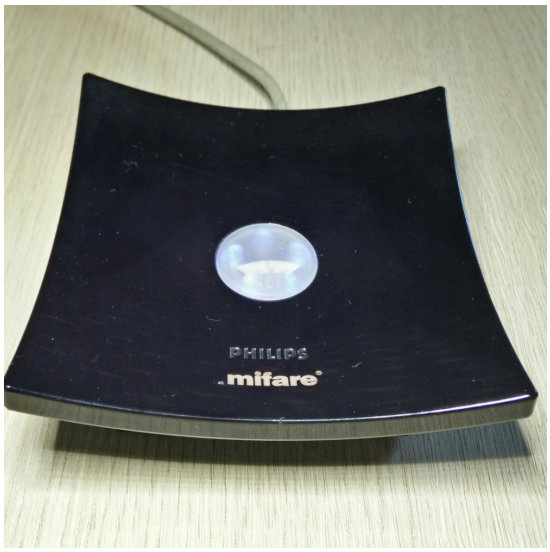
Figura 4.2: Ejemplo de reconocimiento de dispositivo con `pcsc_scan`. De los tres dispositivos conectados, únicamente el Gemalto Prox-DU, con sus dos ranuras, es correctamente reconocido

ya que sus controladores no son estándar y se ofrecen únicamente para plataformas Windows[21]. Por ese motivo, este lector únicamente se ha empleado en un equipo con ese sistema operativo para realizar capturas de tráfico y comprender mejor el contenido de las tramas intercambiadas con las tarjetas.

El hecho de que el dispositivo ACS ACR122U (ver Fig. 4.3b) no fuera compatible sí resultó sorprendente, ya que el fabricante incluye entre las características[22] destacadas de su producto la conformidad con las especificaciones CCID y el soporte de PC/SC. Además, ACS ofrece drivers específicos para el uso del dispositivo mediante PC/SC en Linux/Mac OS X, que han sido instalados de acuerdo con las instrucciones que se les adjuntan. Con versiones anteriores de `pcsc-lite` y de Ubuntu se logró que el *daemon* reconociera el lector, pero la transmisión de tramas nunca pudo realizarse correctamente debido a la aparición de problemas de compatibilidad con el "*Resource Manager*". Como ha podido comprobarse, en las versiones más recientes de la librería y del sistema operativo ni siquiera se reconoce su presencia a pesar de lo anunciado por el fabricante.

Una investigación en mayor profundidad nos permitió determinar que el modelo en cuestión

[21] http://www.mifare.net/en/products/mifare-reader-ic-s/demoboard-site/pegoda/
[22] http://www.acs.com.hk/en/products/3/acr122u-usb-nfc-reader/



(a) Philips Mifare Pegoda

(b) ACS ACR122U

(c) Gemalto Prox-DU

Figura 4.3: Dispositivos lectores empleados durante la elaboración del proyecto

no es completamente compatible con `pcsc-lite`, al menos en las unidades más antiguas[23]. Como pudimos comprobar, la versión del firmware de la que disponía nuestra unidad se encontraba entre las listadas como problemáticas, lo que provocaba la aparición de errores en la comunicación incluso cuando se reconocía su presencia. Actualmente existe una modificación de los drivers

[23] https://pcsclite.alioth.debian.org/ccid/unsupported.html

CCID que, en teoría, permite evitar la aparición de dichos errores[24]. Sin embargo, no existía esa posibilidad cuando nos enfrentamos a dicho contratiempo, por lo que se prefirió optar por adquirir un dispositivo que se encontrara incluido entre los completamente compatibles. En caso de que el lector de esta memoria disponga ya de un dispositivo del modelo ACR122U y desee comprobar si su unidad es de las problemáticas, debe enviar, de acuerdo con lo indicado en la documentación ofrecida por ACS [1], una trama `FF00480000`[25] mediante el software que acompañaba al dispositivo[26]. Debe tenerse en cuenta que, en principio, dicho software únicamente se encuentra disponible en equipos con Microsoft Windows. La respuesta al comando mencionado anteriormente debería incluir la versión del firmware de la que dispone la unidad lectora. Si dicha versión es superior a la `2.06` no deberían producirse errores de comunicación.

Finalmente, el dispositivo con el que se ha podido trabajar sin contratiempos ha sido un Gemalto Prox Dual USB PC Link Reader, habitualmente conocido como Gemalto Prox-DU (ver Fig. 4.3c), aunque recientemente ha sido renombrado a IDBridge CL300. Sin duda alguna, su rasgo más característico es la posibilidad de su uso para la lectura tanto de tarjetas inteligentes sin contacto como con contacto, debido a presencia una ranura distinta para cada tipo. De esta forma el dispositivo puede comunicarse con cualquier tipo de tarjeta sin contacto que siga los estándares ISO 14443 A o B y sea compatible con el protocolo de transmisión de datos `T=CL`, así como con todos los modelos de tarjetas de la familia Mifare lanzados hasta la fecha, se adecúen o no al estándar. No obstante, únicamente se soporta el protocolo `T=1` y no el `T=0`. Del mismo modo, también pueden establecer comunicaciones con aquellas tarjetas que sigan la norma ISO 7816. Esta característica, sin embargo, tiene sus limitaciones ya que, según avisa el propio fabricante, no es posible mantener conexiones activas con tarjetas en ambas ranuras simultáneamente. Además, este modelo de dispositivo no soporta la multiactivación de tarjetas sin contacto, lo que implica que tan sólo se podrá establecer conexión con la primera tarjeta que sea detectada por el dispositivo, siendo el resto ignoradas aunque se encuentren dentro del campo de detección.

A diferencia de lo sucedido con los lectores mencionados anteriormente, los dispositivos Gemalto Prox-DU siguen completamente las especificaciones PC/SC en su versión 2.0, lo que permite su uso mediante el middleware empleado en nuestro proyecto. Además, son compatibles con los controladores CCID para la comunicación con sus dos interfaces, tanto el de contacto como el sin contacto. En este sentido, no es necesario disponer de controladores propietarios, ya que se garantiza su completa adecuación al estándar, por lo que implementaciones de código libre como la del proyecto `pcsc-lite` resultan perfectamente válidas. Lo mismo sucede con el interfaz HID que permite establecer comunicación con el propio lector para, por ejemplo,

[24] http://www.average.org/pcsclite-ACR122U/

[25] Según algunas fuentes (p. ej., http://www.yiersan.org/rdg/RFID), en caso de error puede utilizarse la trama `FF0048000A` con el mismo propósito

[26] http://www.acs.com.hk/en/products/12/acr122u-nfc-contactless-smart-card-reader-software-development-kit/

modificar su configuración si resulta necesario. Sin embargo, los comandos que permiten dicha personalización del comportamiento del lector sí son propios de Gemalto, por lo que en caso de querer emplearlos es necesario acudir al manual de referencia que ofrece el fabricante[27] [13]. Finalmente, resulta interesante mencionar que los lectores Gemalto Prox-DU disponen de una implementación propietaria de las tramas APDU definidas en las normas ISO que les permiten extender su funcionalidad para la transmisión de comandos nativos de las tarjetas Mifare DESFire, lo que reduce las posibilidades de que se produzcan conflictos en el administrador de recursos PC/SC.

4.4. Conclusión

En este capítulo hemos revisado a grandes rasgos los elementos que componen la infraestructura empleada en el desarrollo del proyecto que se presenta en esta memoria. A pesar de los esfuerzos que se han realizado para que la implementación de la librería resulte lo más agnóstica posible a la plataforma sobre la que se emplee, como ha podido comprobarse esto no es posible en muchas ocasiones. Por tanto es recomendable, en la medida de lo posible, asegurarse de disponer de una infraestructura adecuada tanto a nivel de software como de hardware para garantizar que su funcionamiento es el correcto.

Un punto clave es, sin ninguna duda, la elección del interfaz Java de nivel intermedio. De las dos posibilidades que han sido tenidas en cuenta tan sólo debería optarse por la librería `jpcsc` en caso de que resulte imprescindible el uso de tarjetas Mifare DESFire originales y se disponga de un equipo de 32 bits con una instalación de Ubuntu o una distribución similar. En el resto de circunstancias, `javax.smartcardio` es una elección más adecuada. Como veremos en el Capítulo 5, se ha tratado de realizar una clara separación entre nuestra propia librería y el interfaz Java que se esté empleando, de modo que sea posible su reemplazo por otro de forma transparente.

Por último, debemos comentar que, obviamente, la correcta elección del dispositivo lector también es una parte fundamental de la infraestructura necesaria. Aunque a día de hoy la gran mayoría de lectores que pueden obtenerse son compatibles con los estándares sobre los que ha sido construida la librería, es muy aconsejable realizar una comprobación exhaustiva previa para evitar contratiempos como los que nos hemos encontrado durante la elaboración del proyecto. El listado mantenido por el proyecto `pcsc-lite` que hemos comentado con anterioridad es un muy buen punto de partida.

[27] http://support.gemalto.com/fileadmin/user_upload/drivers/Prox-DU_and_Prox-SU/DOC118569D_Prox-DUSU_RefMan.pdf

5 | Implementación de la Librería

En el presente capítulo se describen los contenidos de la librería que ha sido creada para facilitar la implementación de aplicaciones que se comuniquen con tarjetas Mifare DESFire. La librería desarrollada, a la que hemos denominado `dflibrary`, contiene una serie de clases, organizadas en varios paquetes y subpaquetes, cuyos principales cometidos son:

- Proporcionar un interfaz de abstracción común a las librerías de nivel intermedio que se han comentado en la Sección 4.2.3 (paquete `dflibrary.middleware`)
- Permitir el establecimiento de sesiones de comunicación con tarjetas Mifare DESFire y Mifare DESFire EV1, incluyendo implementaciones de los distintos comandos incluidos en la API de las tarjetas en sus distintas versiones (paquete `dflibrary.library`) y de los mecanismos de seguridad que les son propios (subpaquete `dflibrary.library.security`)
- Ofrecer abstracciones de alto nivel a los numerosos elementos necesarios para la configuración de las tarjetas y su uso para la transmisión de información, como pueden ser los permisos de acceso, los distintos tipos de ficheros o las opciones de configuración de claves (subpaquete `dflibrary.library.param`)
- Dotar a la librería de una serie de utilidades para el manejo de tramas de bytes (paquete `dflibrary.utils.ba`) y herramientas criptográficas y de seguridad (paquete `dflibrary.utils.security`), tanto para su uso interno desde la propia librería como externo por parte de un desarrollador de aplicaciones que las necesite

Las siguientes secciones están dedicadas a una explicación en mayor detalle de las funcionalidades que ofrecen cada uno de dichos paquetes. La librería, además, incluye el código fuente de las aplicaciones de demostración desarrolladas en el paquete `dflibrary.samples`, pero su explicación se pospone hasta el Capítulo 6.

5.1. Interfaz de nivel intermedio `ComManager`

Como ya mencionamos en el apartado dedicado al "middleware" del Capítulo 4, para poder acceder a las funcionalidades que nos ofrecen las tarjetas Mifare DESFire mediante el envío de

sus comandos y la recepción de las respuestas correspondientes es necesario el establecimiento de un canal de comunicación a través del dispositivo. Las especificaciones PC/SC, como ya hemos visto, nos proporcionan un modo estándar de realizar dicha comunicación en todos aquellos dispositivos que cumplan con la norma. Sin embargo, no debemos olvidar que el modo de acceso desde Java a las funcionalidades ofrecidas por `pcsc-lite`, la implementación más habitual de las especificaciones en entornos Unix, no se encuentra homogeneizado. De esta forma, si se opta por emplear como capa intermedia la librería `jpcsc` o el paquete `javax.smartcardio`, por ejemplo, las llamadas necesarias para el establecimiento del canal de comunicación y su posterior uso para el envío de comandos serán diferentes. Este hecho nos condujo a concluir que era imprescindible asegurar que el código desarrollado fuera completamente independiente de la implementación de la capa intermedia que se escogiera.

Para lograr el aislamiento propuesto, que permita el reemplazo de una implementación de la capa de nivel intermedio por otra sin necesidad de alterar el código de la librería, se decidió la creción de un interfaz Java que proporciona unas cabeceras homogeneizadas para los métodos más habituales. El uso de un recurso del lenguaje de programación Java como son los interfaces permite describir lo que deben hacer las clases que se encarguen de gestionar las comunicaciones con las capas inferiores sin necesidad de especificar cómo deben hacerlo. Dicho de otra forma, el interfaz `ComManager` describe los requisitos que debe cumplir cualquier clase gestionadora de comunicaciones que se pretenda utilizar con la librería, independientemente de si lo hace mediante la aplicación de las especificaciones PC/SC o no.

A continuación se comentan brevemente las funcionalidades que deben cumplir los métodos que se encuentran definidos en el interfaz:

`void scan()`
: Inicializa la estructura de representación del contexto del sistema si es necesario, determinando qué dispositivos son detectados.

`String[] listReaders()`
: Obtiene los nombres de los dispositivos que se encuentran actualmente conectados al sistema y pueden ser satisfactoriamente detectados por el Administrador de Recursos PC/SC (o su equivalente si se emplea una capa de bajo nivel diferente).

`void select(String readerName)`
: Marca el dispositivo identificado mediante `readerName` como seleccionado para la realización de posteriores operaciones. La cadena de caracteres que representa el nombre del dispositivo a seleccionar debe coincidir exactamente con una de las entradas de la respuesta al método `listReaders`.

`void deselect()`
: Cancela la selección de dispositivo realizada con anterioridad.

`boolean isCardPresent()`
: Nos indica si se detecta una tarjeta dentro del campo de actuación del dispositivo previamente seleccionado.

`void waitCardInsertion()`
: Detiene la ejecución del proceso actual hasta que el dispositivo seleccionado detecte un evento de inserción de tarjeta.

`void waitCardExtraction()`
: Detiene la ejecución del proceso actual hasta que el dispositivo seleccionado detecte un evento de extracción de tarjeta.

`void connect()`
: Establece un canal de comunicación con una tarjeta presente en el dispositivo seleccionado.

`byte[] send(byte[] command)`
: Transmite una trama de bytes a través del canal previamente establecido y devuelve la respuesta obtenida de la tarjeta.

`void reconnect()`
: Reinicia un canal de comunicación previamente establecido.

`void disconnect()`
: Cierra un canal de comunicación previamente establecido.

`void release()`
: Libera, si es necesario, la estructura representativa del contexto del sistema generada mediante una llamada a `scan()`.

`CardType getCardType()`
: Obtiene una representación del modelo de la tarjeta detectada en el dispositivo seleccionado, normalmente a partir de una lectura de su ATR.

Mediante el uso de este pequeño número de métodos es posible realizar las principales operaciones necesarias para reconocer la presencia de una tarjeta en un dispositivo y establecer una comunicación con ella. Asimismo, una ventaja adicional de haber optado por la definición de un interfaz es que permite concentrar en una única instancia de una clase varias funcionalidades relacionadas que habitualmente se encuentran diseminadas en múltiples clases, como vimos en la Sección 4.2.3. Así, es posible escribir el código que nos permita ofrecer un servicio de una forma mucho más compacta y legible. Tómese como ejemplo el código incluido en el Listado 5.1, en el que se puede observar como puede usarse una hipotética implementación del interfaz que hemos denominado `ActualComManager` para el establecimiento, realización y finalización de la comunicación con una tarjeta presente en un dispositivo conectado a nuestro sistema. Como

puede comprobarse, para mejorar la claridad del ejemplo hemos obviado buena parte del código que se requeriría en una aplicación real, como la definición de la clase y el método en el que se encontraría este fragmento de código y los bloques `try-catch` para la captura de posibles excepciones.

Listado de Código 5.1: Ejemplo de uso de una implementación del interfaz `ComManager`

```
1  ComManager cm = new ActualComManager();
2
3  cm.scan();
4  cm.select(cm.listReaders()[0]);
5
6  if(cm.isCardPresent()){
7      cm.connect();
8      System.out.println(cm.getCardType());
9      // byte[] res = cm.send(byte[] com)
10     cm.disconnect();
11 }
12
13 cm.deselect();
14 cm.release();
```

Obviamente, para que el interfaz sea de utilidad, se requiere al menos una implementación concreta de sus métodos. En este sentido, la librería incluye varias clases que cumplen este cometido y que pueden ser empleadas en el desarrollo de aplicaciones reales. La primera de ellas, que hemos denominado `JPCSCComManager`, crea una capa encima de la librería de nivel intermedio `jpcsc` que ya mencionamos anteriormente. Sin embargo, tal y como ya se comentó, dicha librería dejó de ser mantenida hace algunos años, lo que ha provocado que la última versión que se lanzó no pueda ser compilada de nuevo en sistemas actuales. Aunque se dispone de una versión precompilada, ésta no tiene ninguna garantía de poder emplearse en sistemas diferentes a aquél en el que se realizó su compilación original. Por esa razón se decidió incluir una implementación alternativa basada en el paquete Java `javax.smartcardio` que hemos denominado `JSCIOComManager`. De todos modos, no debemos olvidar que las restricciones de dicho paquete con respecto a la estructura de las tramas a enviar impiden que esta implementación pueda ser empleada para la comunicación con tarjetas Mifare DESFire originales. Esto implica que en determinadas circunstancias puede ser necesario el uso de `JPCSCComManager`, aunque en la mayoría de casos es preferible optar por `JSCIOComManager`. Finalmente, también se proporciona una clase llamada `RemoteComManager` cuyo único cometido es la simulación de una comunicación con dispositivos localizados en un sistema remoto. Para tal propósito se ha decidido implementar únicamente un constructor y el método `send`, lo que permite simular el intercambio de datos. El resto de métodos, sin embargo, no resultan de utilidad, por lo que han sido dejados como "stubs". Su utilidad se entenderá mejor en el Capítulo 6 cuando veamos una aplicación concreta que emplea una instancia de esta clase.

Mediante la definición del interfaz y sus implementaciones podemos, en principio, establecer comunicaciones con cualquier tipo de tarjeta inteligente, sea o no sea sin contacto. Sin embargo, si queremos desarrollar aplicaciones que saquen provecho de las funcionalidades que ofrecen las tarjetas Mifare DESFire es necesario el desarrollo de un framework particular. El siguiente apartado describe las clases que han sido creadas para tal propósito.

5.2. Comunicación con tarjetas Mifare DESFire

El verdadero núcleo de la librería que presentamos en este informe lo conforman las clases destinadas al envío y recepción de información durante el mantenimiento de un diálogo entre una aplicación y una tarjeta Mifare DESFire. De entre ellas, la más relevante es, sin duda, la que hemos denominado `DFCard` ya que es donde se incluyen las implementaciones de los comandos DESFire que fueron introducidos en la Sección 3.2. Para poder emplearlos en una aplicación es necesario crear un objeto de dicha clase pasando como parámetro una instancia de `ComManager` *previamente conectada*, tal y como se indicó en el apartado anterior. En este sentido, el Listado 5.2 muestra cómo ampliar el código presentado previamente en el Listado 5.1 para la ejecución de comandos DESFire mediante una instancia de `DFCard`. Como en el caso anterior, se ha prescindido de aquellas líneas de código que hubieran dificultado la comprensión de la estructura propuesta. No debe olvidarse su inclusión en aplicaciones reales.

Listado de Código 5.2: Ejemplo de uso de una instancia de la clase `DFCard` para el envío de un comando DESFire

```
1  ComManager cm = new ActualComManager();
2
3  cm.scan();
4  String[] readers = cm.listReaders();
5  cm.select(readers[0]);
6
7  if(cm.isCardPresent()){
8      cm.connect();
9      if(cm.getCardType() == CardType.MIFARE_DESFIRE){
10         DFCard df = new DFCard(cm);
11         System.out.println(df.getKeyVersion(0));
12     }
13     cm.disconnect();
14 }
15
16 cm.deselect();
17 cm.release();
```

Como puede observarse, la creación de la instancia de clase `DFCard` se produce únicamente en caso de que el `ComManager` reconozca el modelo de la tarjeta conectada como Mifare DESFire.

Para ello hacemos uso del objeto del tipo enumerado `CardType` que se obtiene como respuesta del método `getCardType`. En la clase `CardType` se proporcionan objetos únicos para una serie de modelos de tarjetas sin contacto para las cuales se conoce su ATR. En futuras versiones de la librería podría ser útil la ampliación del listado actual para incluir otros modelos de tarjeta, tanto sin contacto como con contacto.

Debemos hacer notar que no es posible distinguir entre las tarjetas Mifare DESFire originales y las EV1 únicamente a través del ATR, por lo que el objeto obtenido será el mismo para ambos modelos. Recordemos que la distinción entre ambas generaciones es imprescindible ya que la construcción de las tramas de envío de los comandos es diferente entre ellas. Por esta razón el constructor de la clase `DFCard` realiza internamente una llamada a un método definido en las normas ISO que permite determinar si la tarjeta conectada soporta la estructura estándar de las tramas o no. Asimismo, el constructor también inicializa un objeto de clase `DFSession`, cuyo cometido es mantener un seguimiento de una serie de atributos que deben actualizarse a medida que se ejecuten comandos de comunicación, especialmente en materia de seguridad. Los detalles relativos a qué atributos deben ser monitorizados y con qué objetivo únicamente pueden encontrarse en la documentación de carácter confidencial, por lo que es necesario trasladar su explicación al sumario de la API DESFire que se incluye como apéndice en la versión confidencial del presente informe.

El número de métodos que se encuentran a disposición de los desarrolladores, tanto en la mencionada clase `DFCard` como en el resto de las complementarias que se introducirán a continuación, es demasiado elevado como para ser enumerados aquí. Por ese motivo, recomendamos a todo aquél que desee emplear la librería que acuda a la documentación que se le adjunta para saber cómo realizar las llamadas a cada comando particular. Si además se desea conocer detalles del funcionamiento interno de dichos comandos será necesario disponer de la documentación confidencial proporcionada por NXP tras firmar el NDA con ellos. Esto es altamente recomendable, ya que, por ejemplo, gran parte de los requisitos de seguridad necesarios para la satisfactoria ejecución de los comandos únicamente pueden encontrarse en dichas fuentes.

Como puede comprobarse en la documentación adjuntada en el Apéndice C, todos los comandos han sido implementados al menos en dos versiones distintas. Aunque Java permite la coexistencia de múltiples métodos con el mismo nombre si el número y/o tipo de sus argumentos difiere, nuestra intención era disponer de tipos de retorno distintos. Esto no es posible si no se distinguen explícitamente los métodos asignándoles nombres diferentes. Para que dichos nombres fueran lo menos intrusivos posibles en el flujo de programación de las aplicaciones se decidió que se usaría la siguiente convención: los métodos cuyo nombre empieza con mayúscula devuelven directamente la trama de bytes obtenida como respuesta de la tarjeta, mientras que aquellos cuya inicial sea minúscula devuelven un objeto de la clase `DFResponse`, que encapsula abstracciones de los elementos que componen la respuesta recibida.

Tomemos como ejemplo las dos versiones del comando `GetVersion`, que nos devuelve una serie de características relativas a las versiones de hardware y software de la tarjeta conectada. Asumamos que estamos trabajando con una tarjeta Mifare DESFire EV1, y que el objeto `df` es una instancia correctamente inicializada de la clase `DFCard`. En este sentido, mostramos en los Listados 5.3 y 5.4 las salidas obtenidas tras realizar llamadas a los métodos `df.GetVersion()` y `df.getVersion()`, respectivamente.

Listado de Código 5.3: Respuesta obtenida al ejecutar el comando `df.GetVersion()` en una tarjeta Mifare DESFire EV1

```
1 00040101010018050401010104180504794 0B2F82B80BA345755301712
```

Listado de Código 5.4: Respuesta obtenida al ejecutar el comando `df.getVersion()` en una tarjeta Mifare DESFire EV1

```
1  Successful operation.
2  Hardware Info:
3  [Vendor ID: 4,
4  Type: 1,
5  Subtype: 1,
6  Major Version Number: 1,
7  Minor Version Number: 0,
8  Storage Size: 4096 Bytes,
9  Communication Protocol Type: 5],
10 Software Info:
11 [Vendor ID: 4,
12 Type: 1,
13 Subtype: 1,
14 Major Version Number: 1,
15 Minor Version Number: 4,
16 Storage Size: 4096 Bytes,
17 Communication Protocol Type: 5],
18 UID: 0x047940B2F82B80,
19 Production Batch Number: BA34575530,
20 Production Week: 23,
21 Production Year: 18
```

En ambos casos la información recuperada es la misma, aunque claramente mejora su interpretabilidad si se usa un objeto de tipo `DFResponse` para su encapsulamiento. Además, permite el acceso individual a cada uno de los campos de la respuesta, como en el caso anterior, lo que resulta especialmente útil cuando es necesario su uso en posteriores envíos de comandos. Por ejemplo, si no tenemos en cuenta que existe un comando específico para ello en las EV1, podríamos extraer de forma sencilla el UID de la tarjeta y emplearlo para computar la diversificación de una clave sin necesidad de saber en qué posiciones de la trama recibida se encuentra esa información en concreto.

Un objeto `DFResponse` contiene siempre dos atributos. El primero de ellos es de tipo `SC` (siglas de "Status Code") e indica si la ejecución del comando en la tarjeta ha sido satisfactoria o no, reflejando la causa del fallo en su caso. Habitualmente esperaremos que la ejecución haya sido exitosa, pero debemos asegurarnos antes de continuar con el envío de nuevos comandos. Por ello es conveniente llamar al método `isOk()` en la respuesta siempre que ejecutemos un comando. Si deseamos conocer exactamente el código de estado lo podemos hacer mediante el método `getSC()`.

El segundo atributo de los objetos `DFResponse` representa el contenido de la respuesta en sí, en caso de que el comando enviado devuelva algún tipo de información, como sucedía en el ejemplo anterior. Por contra, la respuesta a otros comandos, como podría ser `FormatPICC`, únicamente contiene el código de estado, por lo que el atributo `field` del objeto `DFResponse` toma un valor nulo. Como es de suponer, tanto el contenido como la estructura de la información obtenida a través de cada comando son diferentes, por lo que también lo debe ser el objeto que los representa. Por ese motivo se decidió que el atributo `field` de la clase `DFResponse` fuera de tipo `Object`, de forma que pudiera contener objetos de cualquier clase. Así, si realizamos una llamada al método `getField()` obtendremos un objecto de tipo `Object` que deberemos convertir a su clase particular para poder acceder a su contenido. La clase que corresponde a la respuesta de cada comando se puede encontrar en la documentación de la clase `DFCard`. Por ejemplo, la información obtenida a través de una llamada a `df.getVersion()` es de tipo `PICCVersion`, según se indica en su documentación. Por tanto, la ejecución del código incluido en el Listado 5.5 nos permitirá recuperar la información para su posterior uso, si así se desea. Para mejorar la claridad del ejemplo, no se incluye la comprobación del código de estado de la respuesta ni la captura de posibles excepciones.

Listado de Código 5.5: Recuperación de la clase de objeto original del atributo `field` de una respuesta a través de una operación de "casting"

```
1 DFResponse res = df.getVersion();
2 PICCVersion pv = (PICCVersion) res.getField();
```

Sin embargo, creemos que para este mismo propósito resulta mucho más adecuado el uso de los diferentes métodos que ofrece la clase `DFResponse`, ya que el código resultante es más elegante y legible. Además, esos métodos se encargan de controlar internamente que la conversión se realice adecuadamente. Volviendo al ejemplo de la llamada a `df.getVersion()` introducido anteriormente, el Listado 5.6 muestra cómo podemos emplear el método `getPICCVersion()` para obtener directamente la información acerca de la tarjeta sin necesidad de realizar explícitamente el "casting" de tipos.

Listado de Código 5.6: Recuperación de la clase de objeto original del atributo `field` de una respuesta a través de su método específico

```
1 PICCVersion pv = df.getVersion().getPICCVersion();
```

Independientemente del modo en que obtengamos el objeto en su clase original, tras realizar la conversión podremos aprovechar sus campos y funcionalidades. Por ejemplo, podríamos recuperar el UID de la tarjeta mediante una llamada al método `getUID()` del objeto `pv` creado en los listados anteriores, que devuelve un objeto de la clase `UID`. Tanto `PICCVersion` como `UID` forman parte del subpaquete `dflibrary.library.param`, que contiene todas aquellas clases que proporcionan abstracciones tanto de la información retornada por la tarjeta como de aquellos argumentos que son necesarios en determinados comandos. Por ejemplo, el comando `SelectApplication` puede ejecutarse mediante una llamada al método `DFResponse selectApplication(AID aid)`, donde `AID` es una clase perteneciente a dicho paquete que encapsula un identificador de aplicación. Otras clases de uso habitual pertenecientes a este subpaquete serían `KeySettings`, que representa la configuración de claves (actual o deseada) de una aplicación de la tarjeta, o los tipos de datos que pueden almacenar los ficheros, como `Data` o `Value`. Para agrupar todas aquellas clases que hacen referencia a las propiedades de los distintos tipos de ficheros, incluyendo aquellas comunes entre todos ellos como son los derechos de acceso (`AccessRights`) o los modos de comunicación (`ComSet`), se incluye también el subpaquete `dflibrary.library.param.fileset`. De nuevo remitimos al lector interesado a la documentación de la librería para obtener un listado completo de las clases disponibles y sus características particulares.

También cumple una función similar la clase `DFKey`, que es necesaria para el envío de parámetros en diversos comandos. Sin embargo, debido a su papel fundamental en el cómputo de los mecanismos de seguridad particulares de las tarjetas Mifare DESFire se decidió que formara parte del subpaquete `dflibrary.library.security` junto al tipo enumerado `AuthType` y a la clase `DFCrypto`. Esta última es, sin duda alguna, uno de los pilares más importantes sobre los que se sustenta la librería. Sus métodos estáticos ofrecen las funcionalidades criptográficas y de seguridad que son necesarias para la adecuación de los comandos a sus especificaciones. De este modo es posible establecer comunicaciones seguras con las tarjetas Mifare DESFire. Sin embargo, a diferencia de lo que sucedía con los métodos incluidos en `DFCard`, su utilidad queda limitada al interior de la propia librería. Las funcionalidades de este mismo ámbito de carácter más genérico son ofrecidas como utilidades auxiliares y serán comentadas en la Sección 5.3.

Todos estos subpaquetes que hemos mencionado tienen como principal objetivo ofrecer abstracciones para las versiones "interpretadas" de los comandos DESFire. De esta forma podemos evitar la utilización directa de tramas de bytes que son, en general, mucho menos intuitivas de manejar. El uso de estos métodos de nivel superior permite, además, un mayor control respecto de los argumentos a enviar a los comandos. Por ejemplo, el método `Credit` requiere de dos argumentos, que se corresponden con el identificador de fichero en el cual debe incrementarse el valor registrado y con la trama generada a través de una operación criptográfica derivada del valor a incrementar. Una construcción incorrecta de esta trama puede provocar, en el mejor de los casos, un error de ejecución, y, en el peor, que se produzca un incremento excesivo o insuficiente del valor registrado. Si dicho valor es, por ejemplo, el saldo de una tarjeta monedero, errores de

este tipo pueden ser críticos. En cambio, el uso del método `credit` (nótese la minúscula) garantiza que la construcción de la trama a enviar se realizará de forma consistente, minimizando las posibilidades de aparición de errores de ese tipo. Este mayor control, sumado a la comodidad que proporcionan tanto en su uso como en su interpretación, provoca que creamos conveniente el uso de los métodos "*raw*" (o "en crudo", es decir, aquellos que devuelven directamente las tramas de bytes) exclusivamente para realizar operaciones de depuración de código. Por contra, para el desarrollo de aplicaciones deberían emplearse los métodos que hemos denominado "interpretados".

No debemos perder de vista que ni siquiera siguiendo estas recomendaciones podemos garantizar que las comunicaciones que establezcamos se produzcan sin contratiempos. Como ya hemos comentado anteriormente cuando introducíamos los ejemplos de código, nuestras aplicaciones deben estar siempre preparadas para manejar los posibles errores que puedan aparecer. Por ese motivo, nuestra librería también ofrece la clase `DFLException`, que permite encapsular gran parte de los errores más habituales que podemos encontrarnos. Su atributo `type`, al cual podemos acceder a través del método `getType()`, permite identificar la causa de la excepción de entre un amplio abanico de posibilidades. Muchas de ellas son traslaciones directas de los errores definidos en las especificaciones PC/SC, mientras que otros han sido añadidos específicamente. Para facilitar la captura de las posibles excepciones en las aplicaciones que se desarrollen, es muy recomendable que aquellas que puedan aparecer en las capas inferiores sean convertidas a alguno de los tipos ofrecidos por `DFLException`, como así se hace en las implementaciones del interfaz `ComManager` que se incluyen en la librería.

5.3. Funcionalidades auxiliares

El desarrollo de las clases que hemos comentado con anterioridad ha requerido con frecuencia el manejo de tramas de bytes, especialmente en la construcción final de los comandos a enviar a las tarjetas. Las operaciones de transformación de dichas tramas que creemos que pueden resultar útiles para terceros, incluyendo aquellas de carácter criptográfico, han sido agrupadas en dos paquetes que hemos denominado `dflibrary.utils.ba` y `dflibrary.utils.security`. En las Secciones 5.3.1 y 5.3.2 a continuación describiremos algunas de sus funcionalidades más destacadas, remitiendo a la documentación de la librería al lector interesado en conocer todas las posibilidades que ofrecen.

5.3.1. Utilidades de manejo de cadenas de bytes

Gran parte de los métodos incluidos en las clases que forman parte de la librería requieren el tratamiento de tramas de bytes. En particular, las versiones de los comandos "en crudo" que se encuentran en `DFCard` realizan frecuentemente operaciones de transformacion de dichas tramas, sin olvidar que sus parámetros de entrada y salida toman esa forma. Para minimizar

la repetición de código se decidió el traslado de las operaciones que aparecían recurrentemente a una clase propia que denominamos `BAUtils` ("Byte Array Utils"). El número de métodos estáticos que ofrece ha sido ampliado en varias ocasiones, incluyendo nuevas funcionalidades no contempladas con anterioridad e introduciendo nuevas variantes de los métodos ya existentes. Sus funcionalidades principales podrían agruparse en las siguientes categorías:

Conversión de tipos `toBA`, `toString`, `toInt`, `toBoolean`

Operaciones binarias `twosComp`, `xor`, `and`, `not`

Transformación `padding`, `rotateBA`, `reverseBA`, `extractSubBA`

Combinación `concatenateBAs`, `alignBAs`, `create2dBA`, `join2dBAs`

Comprobación `compareBAs`, `isNeg`

Generación `getRandomBA`

Muchos de estos métodos se ofrecen con varias cabeceras distintas, por lo que resulta conveniente consultar la documentación incluida en la librería para determinar qué combinación de parámetros es la más conveniente en cada caso. En algunas de dichas cabeceras se hace uso de tipos enumerados que han sido definidos en el propio paquete. Uno de ellos, `Direction`, se encuentra dentro de `BAUtil` ya que su utilidad se limita a la especificación de una dirección de rotación en el método `rotateBA`. Sin embargo, los otros tres, `DigitFormat`, `SignMode` y `ByteOrder`, se hallan definidos dentro de una clase complementaria que hemos denominado `DigitUtils`. Su principal cometido es ofrecer métodos para el manejo de dígitos en distintos formatos, especialmente en binario y hexadecimal. En particular, ofrece un gran número de métodos de conversión entre formatos, así como algunas operaciones binarias y métodos de comprobación.

Los métodos ofrecidos por estas dos clases son ampliamente usadas dentro de la librería, y pueden también ser útiles en el desarrollo de aplicaciones. Sin embargo, como ya hemos mencionado anteriormente, se recomienda encarecidamente el uso de las versiones "interpretadas" de los comandos. En este sentido, los métodos de manejo de cadenas de bytes cobran mayor sentido en fases de depuración de código y la ampliación de funcionalidades, tanto en futuras versiones de esta librería como otras similares que puedan ser implementadas.

5.3.2. Utilidades de seguridad

Una de las principales características que hacen interesante el uso de las tarjetas Mifare DESFire es su empleo de los mecanismos estándar de criptografía simétrica. El modo en particular en que dichos mecanismos sirven para la construcción de las tramas a enviar y para garantizar la seguridad de los diálogos con las tarjetas son propios de la API de comunicación DESFire. Además, son confidenciales por expreso deseo de NXP. Por esa razón se han agrupado

en el paquete `dflibrary.library.security` que se ha mencionado anteriormente. Sin embargo, aquellas operaciones de carácter genérico sí pueden ser ofrecidas para su reaprovechamiento por los desarrolladores que lo deseen. Estas han sido agrupadas en tres clases principales:

Crypto Ofrece métodos relacionados con la encriptación de tramas de bytes

CRC Permite calcular el Código de Redundancia Cíclica de una trama, tanto de 16 como de 32 bits, de acuerdo con las especificaciones

MAC Proporciona métodos para la computación de códigos MAC y CMAC

Asimismo, también se incluyen una serie de tipos enumerados que permiten el manejo de las distintas posibilidades de seguridad que requiere la librería: `ChainMode`, `CipAlg`, `CipMode`, `KeyDivMode` y `PaddingMode`.

5.4. Conclusión

En este capítulo se ha pretendido dar una idea a los desarrolladores de las posibilidades que ofrece la librería que hemos implementado. Sin embargo, resulta imposible transmitir aquí todo el trabajo que ha llevado su elaboración. La consulta de la documentación asociada a la librería debería proporcionar una mejor referencia de todo aquello que permite. En cualquier caso, el próximo capítulo presenta de una forma más práctica cómo puede emplearse este conjunto de clases para la creación de aplicaciones reales, incluyendo un ejemplo concreto de un servicio que aprovecha gran parte de las funcionalidades de las tarjetas mediante el uso de la librería.

6 | Desarrollo de aplicaciones

A estas alturas de la presente memoria, ya hemos mencionado en múltiples ocasiones que el principal propósito de la librería que se ha implementado es que resulte útil en el desarrollo de aplicaciones reales. Sin embargo, somos conscientes de que las explicaciones ofrecidas en el Capítulo 5 no dejaban suficientemente claro cómo aprovechar el notable número de líneas de código escrito para la generación de servicios que proporcionen verdadera utilidad al usuario final. En este sentido, se presenta en este capítulo una propuesta de estructura genérica para las aplicaciones que utilicen la librería `dflibrary`, así como un ejemplo concreto más complejo en forma de servicio de venta y gestión de entradas para eventos con pago a través de monedero electrónico.

6.1. Anatomía básica de una aplicación

El Listado de Código 5.2 que incluíamos en el capítulo anterior refleja perfectamente los componentes mínimos que debe tener una aplicación Java para emplear la librería `dflibrary`. El uso de estos componentes genera una estructura muy simple y lógica que, de una forma u otra, deberá aparecer en las aplicaciones que desarrollemos. A continuación enumeramos cada una de las distintas etapas, incluyendo el código que cumple su función en la aplicación simple que sugeríamos en dicho listado:

0. Creación del administrador de comunicaciones:

```
ComManager cm = new ActualComManager();
```

1. Reconocimiento del contexto

```
cm.scan();
```

2. Listado de los dispositivos lectores reconocidos

```
String[] readers = cm.listReaders();
```

3. Selección del lector de trabajo

```
cm.select(readers[0])
```

4. Detección de una tarjeta en el dispositivo lector

```
if(cm.isCardPresent()){ /**/ }
```

5. Establecimiento de un canal de comunicación con la tarjeta detectada

```
cm.connect();
```

6. Comprobación de modelo de tarjeta adecuado

```
if(cm.getCardType() == CardType.MIFARE_DESFIRE){ /**/ }
```

7. Creación de una abstracción de la tarjeta

```
DFCard df = new DFCard(cm);
```

8. Ejecución de la operación deseada mediante el intercambio de comandos

```
df.getKeyVersion(0));
```

9. Desconexión del canal de comunicación

```
cm.disconnect();
```

10. Liberación del lector y el contexto

```
cm.deselect();
cm.release();
```

Obviamente, una aplicación real requiere de una estructura mucho más compleja. Es probable que se necesiten operaciones consistentes en combinaciones de diversos comandos, un interfaz (sea gráfico o textual) que permita la interacción con el usuario, y abstracciones que faciliten el manejo de las estructuras de datos específicas para ofrecer el servicio deseado, entre muchos otros componentes. Para ello se deberá construir una estructura de clases más elaborada, en la que posiblemente los elementos listados anteriormente se encuentren algo (o bastante) más dispersos y ocultos. En este sentido, el apartado siguiente muestra una aplicación de este tipo que hemos creado como demostración de las posibilidades de la librería. En cualquier caso, estos elementos son los fundamentos que nos permiten construir esa aplicación o cualquier otra que se nos ocurra.

6.2. Implementando un servicio “real”

En un principio se creyó conveniente la implementación de varias aplicaciones que mostraran cómo emplear la librería para ofrecer distintos servicios. Sin embargo, al final se consideró que centrándonos en una única aplicación que incluyera diversos servicios relacionados conseguíamos

transmitir con mayor intensidad la sensación de que la librería puede ser empleada en el desarrollo de aplicaciones reales. Asimismo, se lograba demostrar la funcionalidad multiaplicación de las tarjetas Mifare DESFire, una de sus características principales pero que no había podido ponerse en relevancia meramente a través de la implementación de la librería.

Teniendo en cuenta las ventajas que aportan las tarjetas sin contacto con respecto a otras tecnologías se decidió simular un servicio de venta y gestión de entradas de eventos. Aunque esto hubiera sido suficiente para demostrar cómo puede emplearse la librería para desarrollar una aplicación, también incluimos un servicio de pago mediante monedero electrónico, que a su vez servirá para realizar las operaciones de compra de entradas. Más concretamente, en la aplicación propuesta serían las entidades bancarias las que emitirían monederos electrónicos (ligados, o no, a cuentas corrientes de sus clientes) en forma de tarjetas Mifare DESFire, que podrían ser ampliadas mediante aplicaciones de terceros. En estas podrían añadirse una gran variedad de funcionalidades, como podrían ser puntos de fidelidad, cupones de descuento o, como en nuestro caso, los datos referentes a unas entradas a usar con posterioridad en el evento adecuado.

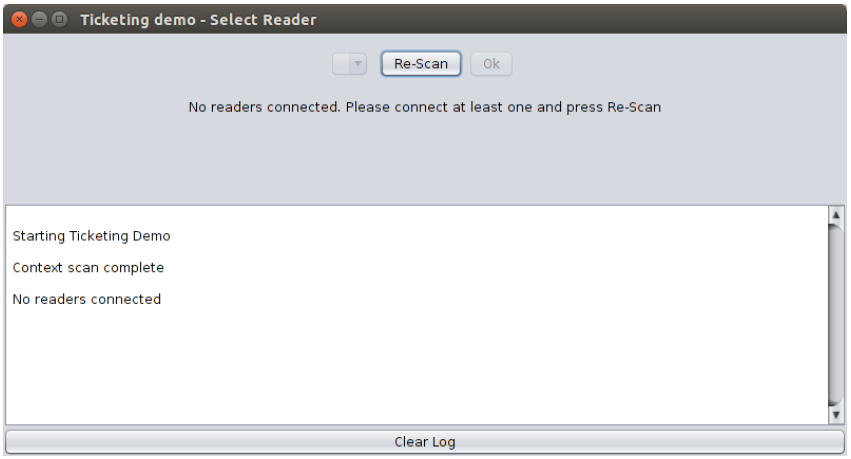
En el siguiente subapartado comentaremos las distintas operaciones que pueden realizarse con la aplicación de demostración, dejando para más adelante las explicaciones acerca de los detalles de su implementación. Creemos que mostrando en primer lugar el funcionamiento de la aplicación, el lector comprenderá mejor su propósito, lo que a su vez le permitirá entender con mayor facilidad la estructura de clases que se propone.

6.2.1. La aplicación de demostración en funcionamiento

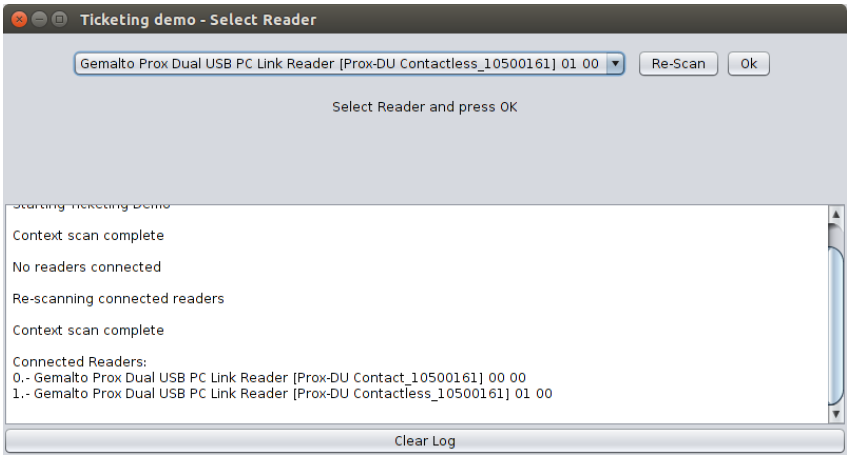
Como veremos, el funcionamiento de la aplicación ha sido pensado para que resulte lo más intuitivo posible, con un interfaz gráfico sencillo que nos guía en cada paso. Sin embargo, antes de lanzar el ejecutable de la aplicación en sí es necesario realizar varias operaciones que es fácil pasar por alto. Incluso nosotros mismos, habiendo diseñado e implementado la operación, las olvidamos a menudo, por lo que si se observa un funcionamiento extraño en la aplicación es probable que se deba a esta circunstancia. En concreto, debemos asegurarnos de que:

- el *daemon* `pcscd`, o su equivalente como Administrador de Recursos si no se emplea `pcsc-lite`, se encuentra en ejecución
- las bases de datos de servicios bancarios y de gestión de entradas se encuentran correctamente construidas y se dispone de acceso al servidor MySQL correspondiente. El Apéndice B incluye ejemplos de código para replicar la base de datos que nosotros hemos empleado en las pruebas de la aplicación
- los servidores bancarios, implementados en la clase `BankServer`, han sido lanzados correctamente (mediante la ejecución de `BankServers`), y se encuentran escuchando en los puertos adecuados

Una vez lanzada la aplicación nos encontramos ante una pantalla como la que se muestra en la Fig. 6.1. En ella se presentan las dos posibilidades que pueden surgir: que no se haya detectado ningún lector compatible en el sistema (6.1a) o que el reconocimiento haya sido satisfactorio (6.1b). En el primer caso deberemos conectar al equipo el dispositivo lector que deseemos emplear y pulsar en el botón `Re-Scan`. Si aún así no aparece ningún dispositivo en la lista desplegable es muy probable que no hayamos iniciado correctamente el Administrador de Recursos (i.e., el *daemon* `pcscd`). De ser así es recomendable detener la ejecución de la aplicación y lanzarla de nuevo una vez se haya comprobado que el *daemon* se encuentra en funcionamiento.



(a) Ningún dispositivo reconocido



(b) Al menos un dispositivo reconocido

Figura 6.1: Selección de lector

Las pantallas que se muestran en la Fig. 6.1 también nos permiten identificar las distintas partes de las que se compone el interfaz de la aplicación. En primer lugar, en la barra de título de la parte superior siempre encontraremos una referencia a la etapa (o "*app*"[1]) en la que nos hallemos en cada momento. En el cuerpo principal de la ventana podemos distinguir, de arriba a abajo, (I) los componentes de interacción con el usuario, (II) el área de mensajes de información, (III) el registro, o *log*, de operaciones realizadas y (IV) un botón de limpieza del registro. Estos elementos se mantienen en todas las pantallas de la aplicación, variando, como es lógico, en los componentes particulares de interacción con el usuario dependiendo de qué tipo de información deba ser intercambiada.

Aunque las transiciones entre *apps* son basante intuitivas, el esquema que se muestra en la Fig. 6.2 puede ayudar a su comprensión. En él se indica, acompañando a cada flecha, la etiqueta del botón que nos traslada de una ventana a otra, así como el nombre de las distintas aplicaciones y subaplicaciones recuadradas. Por otro lado, el botón `Back` que aparece tanto en las tres pantallas de la Fig. 6.3 como en el resto de las que mostraremos más adelante, siempre permite retroceder hasta el interfaz anterior. Es decir, que se recorrerán las flechas del esquema en sentido contrario. Por ejemplo, pulsando `Back` desde la selección de operaciones del monedero electrónico (6.3b) volveremos a la selección de aplicaciones general (6.3a), mientras que si lo hacemos desde este último menú regresaremos a la selección de dispositivo lector que hemos comentado anteriormente (6.1).

Una vez seleccionado el lector con el que queramos operar, la aplicación nos mostará una pantalla como la que se puede encontrar en la Fig. 6.3a. Los botones del área de interacción con el usuario nos permiten escoger la tarea (o grupo de tareas) a realizar. Asumiendo que la tarjeta es "virgen"[2], el primer paso para ofrecer el servicio correctamente es configurar el monedero electrónico, simulando la emisión de la tarjeta por parte de una entidad bancaria. Si tratáramos de configurar la aplicación de *ticketing* antes recibiriamos un mensaje de error. Para acceder a la operación correcta es necesario pulsar el botón `Wallet` del menú anterior, lo que nos llevará a una pantalla como la mostrada en la Fig. 6.3b.

Como ya hemos comentado, el primer paso a realizar es configurar el monedero electrónico en la tarjeta, simulando su emisión por parte de una entidad bancaria. La pantalla que se nos muestra en este caso puede encontrarse en la Fig. 6.4a. Como puede comprobarse, se nos presenta un desplegable con un listado de nombres de bancos y un cuadro de entrada de texto donde se nos pide que introduzcamos la cantidad correspondiente al saldo inicial del que se dispondrá en el monedero.

[1] De ahora en adelante denominaremos "aplicación" al software completo desarrollado, y usaremos el término *app* para referirnos a cada una de las pantallas que permiten realizar una operación concreta

[2] Consideramos una tarjeta como "virgen" si nunca ha sido utilizada tras su adquisición o bien ha sido restaurada su configuración de fábrica mediante un formateo de sus datos registrados y reescritas las claves por defecto

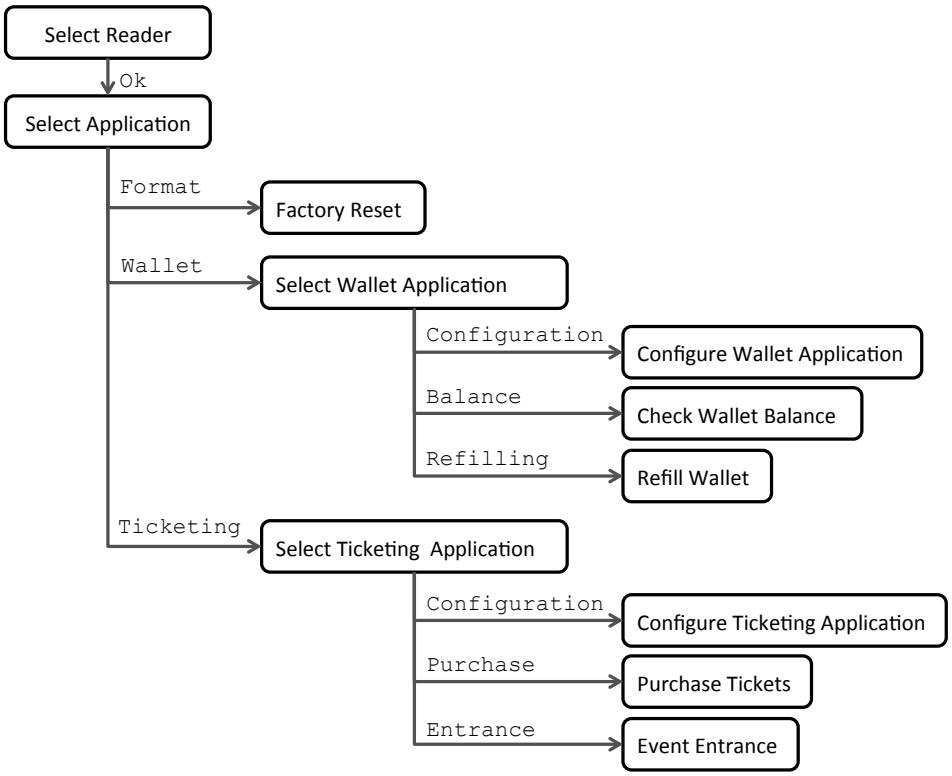
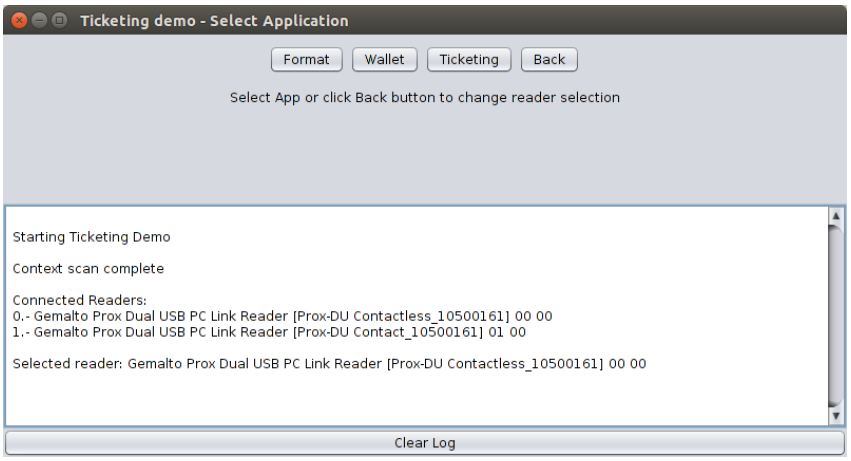


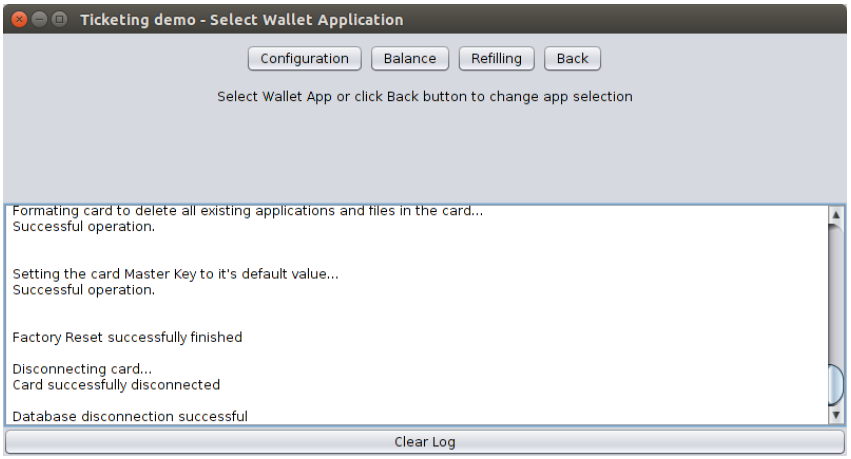
Figura 6.2: Transiciones entre *apps* en la aplicación de demostración

Los bancos incluidos en el desplegable se obtienen de una base de datos específica para tal propósito, de forma similar a como se recomienda en el Apéndice B. Obviamente, ni los nombres ni los identificadores numéricos que les asignemos son relevantes para el usuario de la demostración. De hecho, la elección por parte de éste de una u otra entidad únicamente tendrá importancia si se desea realizar un formateo de fábrica, ya que, como veremos más adelante, la simulación exige que se seleccione el mismo banco en ambos casos. Esto es así ya que las claves almacenadas en la tarjeta, que se utilizarán posteriormente para restringir los permisos que se otorgan en operaciones sensibles, son editadas de forma diferenciada para cada una de las entidades. De esta forma se pretende mostrar que el emisor de la tarjeta puede garantizar que determinadas operaciones puedan ser realizadas únicamente por quienes dispongan de permiso para ello.

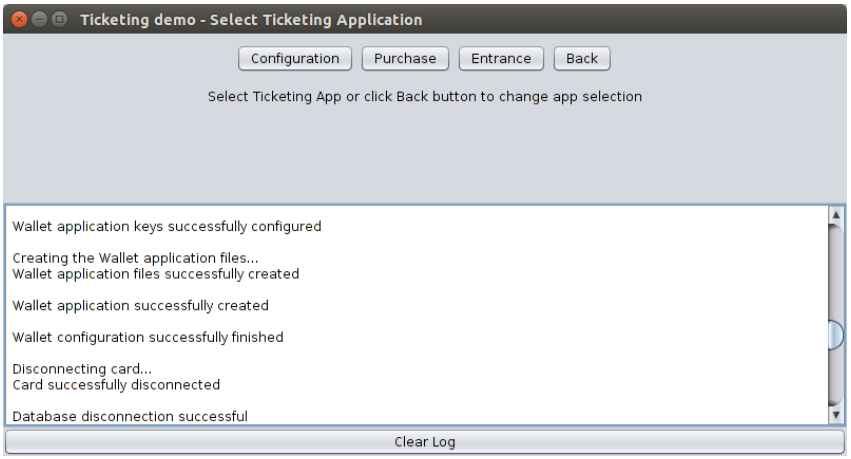
Por su parte, el valor que introduzcamos en la caja de texto se utilizará como saldo inicial del monedero electrónico. Para minimizar la cantidad de intercambios de tramas con la tarjeta, la aplicación comprueba que el texto se corresponda con un número entero entre 0 y 1000, no permitiendo que se intente realizar la operación si no se cumple este requisito. De todas formas, el límite superior es completamente arbitrario y podría ser variado sin demasiados problemas.



(a) Selección de aplicación

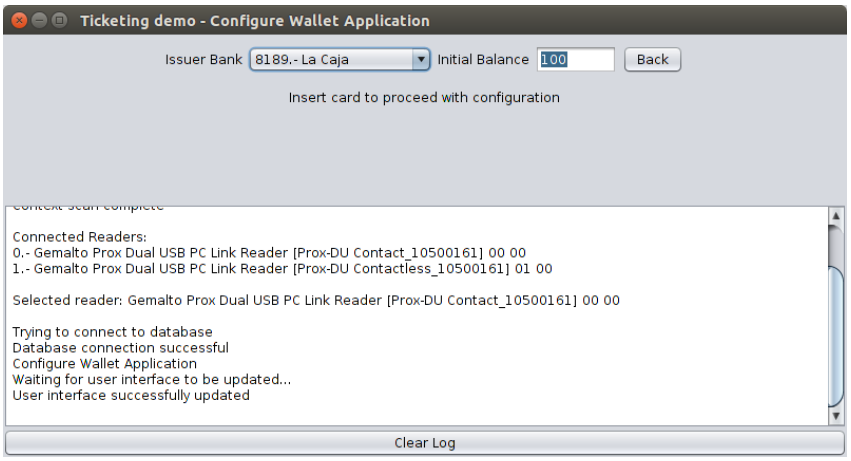


(b) Selección de aplicación de monedero electrónico

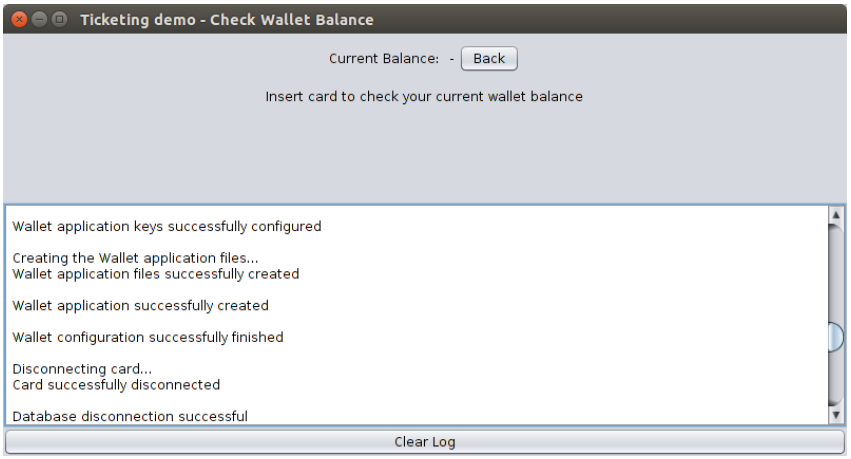


(c) Selección de aplicación de gestión de entradas

Figura 6.3: Ventanas de selección de aplicaciones y subaplicaciones



(a) Configuración de la aplicación



(b) Comprobación de saldo actual

(c) Recarga de saldo

Figura 6.4: Operaciones disponibles en la aplicación de monedero electrónico

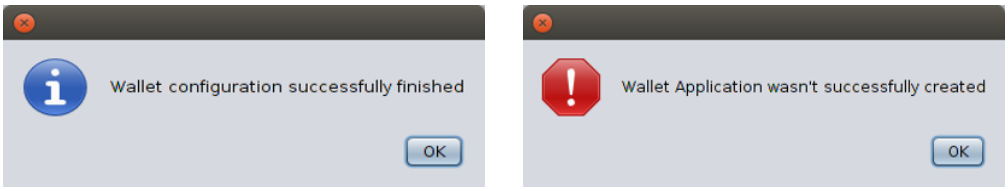
La cantidad con la que decidamos cargar la tarjeta, incluyendo en las operaciones de recarga de saldo que comentaremos más adelante, no afectan al saldo del que disponen los usuarios en sus cuentas corrientes de acuerdo con la base de datos. Aunque este hecho es consecuencia de una decisión deliberada de no complicar la aplicación con funcionalidades no directamente relacionadas con las tarjetas, no puede pasarse por alto que disminuye el grado de realismo del servicio ofrecido. Por este motivo, consideramos que en un futuro debería plantearse la posibilidad de su implementación.

Una vez escogida la entidad bancaria "emisora" y el valor de saldo inicial del monedero electrónico que deseemos, debemos, como nos recuerda el mensaje que aparece en la ventana de la aplicación, introducir la tarjeta en el lector. O mejor dicho, teniendo en cuenta que estamos tratando con tarjetas sin contacto, acercarla al dispositivo. En cualquier caso, para confirmar que deseamos que se realice la operación debemos generar un evento de "inserción de tarjeta". Esto significa que no es suficiente con mantener la tarjeta dentro del campo de actuación del lector, sino que debemos realizar el gesto de acercamiento de forma explícita. El objetivo que persigue esta imposición es evitar que se ejecuten operaciones sin el consentimiento explícito del usuario.

En cuanto se detecta el evento de inserción, se inicia el envío de comandos a la tarjeta para la configuración de la aplicación de monedero electrónico. En concreto, podemos distinguir tres pasos. El primero de ellos consiste en la modificación de la clave maestra de la tarjeta, sustituyendo la clave por defecto que se hallaba registrada previamente por una que solamente el banco emisor conoce. Así impedimos que terceros no autorizados puedan, por ejemplo, formatear la tarjeta o sobreescribir la clave otra vez. Seguidamente, se crea la aplicación en la tarjeta con el identificador apropiado. Este `AID` debe ser independiente de la entidad emisora para que, con posterioridad, pueda ser utilizado por el servicio de venta de entradas uniformemente. Finalmente, se establecen las claves que otorgan los distintos permisos en la aplicación y se crea el fichero contenedor del saldo de la tarjeta con el crédito inicial que hayamos indicado.

Si las tres etapas que hemos comentado son completadas sin inconvenientes, la aplicación nos lo confirmará mostrando un mensaje como el que podemos encontrar en la Fig. 6.5a. Sin embargo, en caso de que se haya producido algún error, la finalización de la ejecución de la operación vendrá acompañada de un mensaje específico que nos avisará de qué etapa ha fallado. Por ejemplo, en la Fig. 6.5b podemos encontrar el mensaje que aparece cuando la creación de la aplicación en la tarjeta no ha podido realizarse. Este aviso concreto es bastante habitual, ya que suele indicar que ya existía previamente una aplicación de monedero electrónico en la tarjeta. En tal caso, podemos escoger si continuar con el resto de operaciones normalmente o si preferimos formatear la tarjeta para iniciar el proceso de nuevo desde el principio.

Tras haber creado y configurado correctamente la aplicación de monedero electrónico ya podemos acceder a las otras dos operaciones que nos ofrecía el menú de la Fig. 6.3b. La primera de ellas, a la que llegaremos a través del botón etiquetado como `Balance`, nos permitirá consultar



(a) Finalización correcta **(b)** Finalización errónea

Figura 6.5: Mensajes de finalización de operación

el saldo que se encuentra actualmente disponible en la tarjeta (Fig. 6.4b). De cara al usuario, su funcionamiento es realmente simple. Nos basta con acercar la tarjeta al lector y un mensaje aparecerá mostrándonos el crédito del que disponemos. Por comodidad, dicho valor también queda reflejado en la ventana principal de la aplicación. Aunque en este caso concreto la ventaja que supone forzar el evento de inserción de tarjeta respecto a la simple detección de la presencia de una PICC en el campo es muy discutible, hemos decicido mantenerlo así por coherencia con el resto de *apps*. Pensamos que el usuario de la demostración agradecerá que todas las operaciones se ejecuten homogéneamente.

Internamente, sin embargo, el funcionamiento de la operación de consulta de saldo no es tan sencillo. La principal causa de esta relativa complejidad es la localización (simulada, obviamente) desde la que se solicita, y se ofrece, cada uno de los distintos servicios. Mientras que la "emisión" de la tarjeta podía entenderse como una operación local, es decir, que se realizaría físicamente dentro de una entidad bancaria, la consulta del saldo disponible debería poder llevarse cabo de forma remota. Siempre que se disponga de la infraestructura adecuada, tanto a nivel de software como de hardware, el acceso a las funcionalidades principales del monedero electrónico debe permitirse incluso desde el propio hogar del usuario. Para lograrlo debemos distinguir dos etapas en la ejecución de dichas operaciones. En la primera, la aplicación comprueba de forma local que la tarjeta se halle adecuadamente configurada y obtiene de ella el identificador del banco emisor. Este identificador se utiliza, entonces, para establecer una conexión "remota" con un servidor bancario, que será el encargado de gestionar el envío de comandos DESFire e interpretar las respuestas por parte de la tarjeta sin necesidad de mostrar las claves correspondientes al interfaz local en ningún caso. De esta forma se evita comprometer la seguridad del servicio. Esto es posible gracias a la implementación del interfaz `ComManager` denominada `RemoteComManager` que ya sugerimos en la Sección 5.1. No obstante, es evidente que su implementación actual es extremadamente simplista, por lo que debería ser mejorada si se deseara emplear en un escenario real, especialmente a nivel de seguridad.

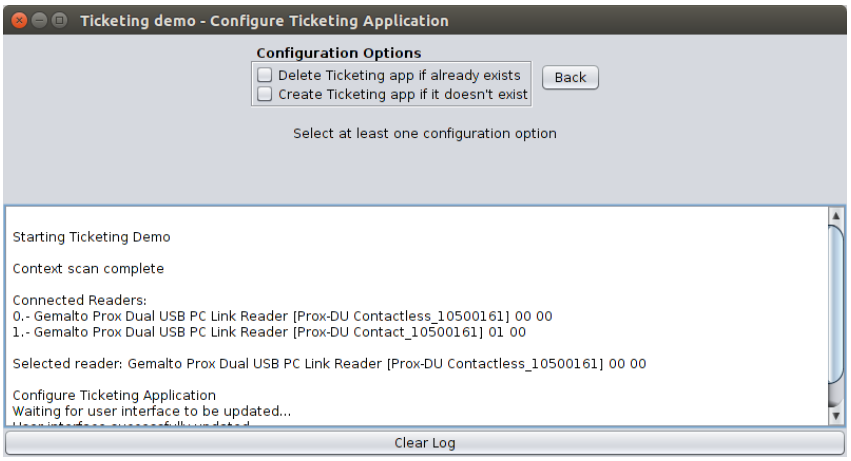
El funcionamiento del sistema de recarga de saldo es esencialmente igual a lo que ya hemos comentado. Una vez accedemos a la pantalla correspondiente (Fig. 6.4c) mediante el botón `Refilling`, únicamente debemos introducir en la caja de texto la cantidad en la que deseemos

incrementar el crédito disponible e insertar la tarjeta para confirmar la operación. Internamente podemos distinguir las mismas dos etapas que mencionamos anteriormente: comprobación local de la configuración y ejecución remota de los comandos. Así se logra realizar la recarga de saldo sin necesidad de disponer localmente de las claves asociadas a la escritura en el fichero que contiene el valor. De todos modos, igual que comentamos en el caso del saldo inicial en la configuración de la aplicación, el hecho de no acceder a la cuenta corriente del usuario para descontarle la cantidad a recargar le quita algo de realismo al servicio ofrecido pero facilita su uso como demostración de las capacidades de la librería.

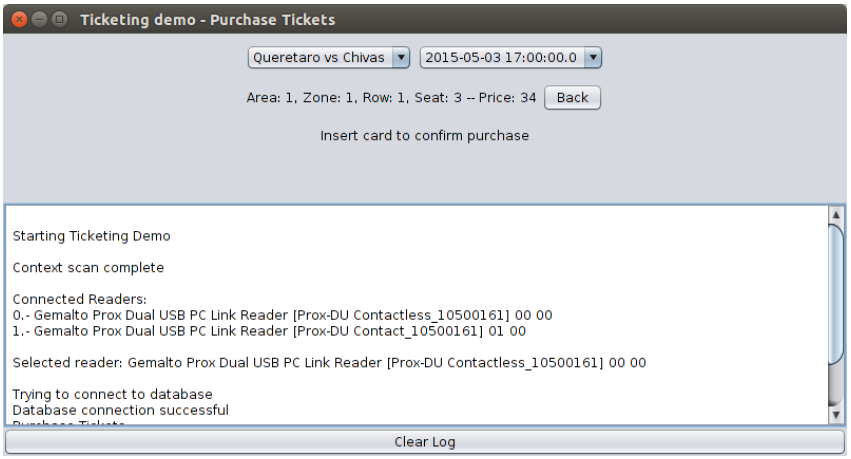
El saldo del que dispongamos en la tarjeta nos permitirá adquirir entradas a través del servicio de *ticketing*. Por descontado, esto requiere que previamente hayamos creado y configurado adecuadamente la aplicación correspondiente en la tarjeta. Para ello, desde el menú general de selección de aplicaciones (Fig. 6.3a) usamos el botón `Ticketing`, accediendo al menú que se nos muestra en la Fig. 6.3c y, entonces, pulsamos el botón etiquetado como `Configuration`. Esto nos llevará a una pantalla como la que encontramos en la Fig. 6.6a. Como puede verse, el usuario debe seleccionar al menos una de las dos opciones que se le ofrecen. A diferencia de lo que sucedía en el caso del monedero electrónico, se ha decidido que la aplicación de `ticketing` pueda ser eliminada y sobreescrita sin necesidad de realizar un formateo completo de la tarjeta. En este sentido, la `app` de configuración permite (I) eliminar una versión preexistente de la aplicación, marcando la caja superior; (II) crear y configurar una nueva versión de la aplicación si no se encuentra ninguna, marcando la caja inferior; y (III) crear y configurar la aplicación, independientemente de si ya existía una versión previa o no, marcando ambas cajas. Con esta diversidad de combinaciones se pretende evitar que se elimine por accidente la aplicación, ya que esto provocaría la pérdida de entradas adquiridas con anterioridad.

Al proceso de compra de las entradas se accede mediante el botón marcado como `Purchase`. La ventana que se nos presenta (Fig. 6.6b) contiene dos desplegables que nos permiten escoger el evento del cual adquirir una o más entradas, especificando su descripción y el horario en el que se ofrece. Debajo se nos indica la localidad que adquiriríamos en caso de aceptar la compra, incluyendo su ubicación y su precio. Sin duda, incluir la posibilidad de seleccionar nosotros mismos la localidad que preferimos sería muy importante en una aplicación real. Pero en el caso de la demo se ha decidido primar la sencillez para el usuario y prescindir de ello. Las diferentes ubicaciones y precios disponibles sí se pueden observar a medida que se adquieren tickets, o si se cambia el evento seleccionado.

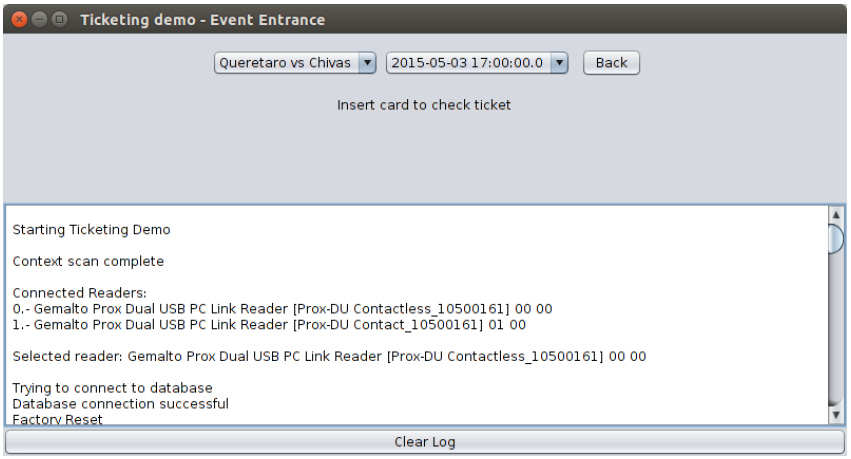
Como siempre, la confirmación de la compra requiere únicamente del acercamiento de la tarjeta, aunque este simple gesto lance la ejecución de múltiples operaciones tras el telón. De los distintos servicios que permite la aplicación de demostración, el proceso de compra es posiblemente el más complejo de todos, ya que además del usuario del sistema requiere de la intervención del proveedor del servicio de gestión de entradas y de la entidad bancaria emisora de la tarjeta. Cada uno de estos agentes conoce un número limitado de claves, disponiendo de



(a) Configuración de la aplicación



(b) Venta de entradas



(c) Control de acceso

Figura 6.6: Operaciones disponibles en la aplicación de gestión de entradas

este modo de permiso para realizar ciertas acciones y otras no. Por ejemplo, el banco es imprescindible para descontar el precio de la entrada del saldo del monedero electrónico, mientras que el proveedor del servicio de gestión de entradas es el único que puede registrar un nuevo ticket en la tarjeta. Para evitar que la complejidad de la aplicación sea excesiva, se decidió no crear un servidor específico para la venta de entradas. De este modo, aunque en una aplicación real la compra remota sería un gran punto a favor, la presencia del servidor bancario ya cumple la función de demostrar las posibilidades de acceso remoto a la tarjeta.

Las distintas acciones que componen esta operación podrían agruparse en las siguientes etapas:

1. Comprobación de la correcta configuración tanto de la aplicación de *ticketing* como del monedero electrónico, incluyendo la determinación del espacio disponible en el fichero de registro de entradas de la tarjeta

2. Verificación de la presencia del identificador de tarjeta en la base de datos del servicio de venta de entradas[3]

3. Acceso al servidor bancario adecuado y realización del pago de forma remota si el saldo disponible es suficiente para cubrir el precio de la entrada

4. Escritura de los datos del ticket adquirido en el fichero correspondiente de la tarjeta

La reducción en el saldo disponible en la tarjeta tras la compra puede comprobarse fácilmente mediante el uso de la *app* `Balance Check` que hemos comentado anteriormente.

En una misma tarjeta pueden registrarse diversas entradas de varios eventos distintos, lo que indudablemente resultaría muy cómodo para los potenciales usuarios del servicio. Por descontado, el espacio disponible es limitado, por lo que en una aplicación real habría que establecer si los tickets deben ser eliminados una vez consumidos o bien si se libera espacio a medida que se necesite.

Una vez adquiridas las entradas, la *app* a la que se accede a través del botón `Entrance` nos permite simular el control de acceso de los eventos. Como puede verse en la Fig. 6.6c, el interfaz consiste únicamente en los mismos dos desplegables que encontrábamos en la pantalla de compra de entradas. De este modo podemos seleccionar el evento al que (supuestamente) queremos acceder. Su funcionamiento es simple. Tras detectar la inserción de una tarjeta, la *app* comprueba que se disponga de fichero de registro de entradas en la aplicación de *ticketing* y lee un ticket tras otro hasta que reconoce alguno que corresponda con el evento actual. En caso de que el ticket no haya sido previamente marcado como consumido, verifica en la base de datos que la localidad correspondiente haya sido asignada a esa misma tarjeta, reduciendo así el riesgo

[3]Este paso es necesario para poder asociar una localidad con un usuario determinado, reduciendo el riesgo de fraude en el control de acceso. Sin embargo, resultaría conveniente que el formulario de registro de tarjeta se realizara dentro de la propia aplicación y no directamente con la base de datos. Este aspecto debe ser mejorado en versiones posteriores de la aplicación de demostración.

de fraude. Finalmente, se edita el fichero interno de la tarjeta para indicar que esa entrada ya ha sido utilizada. Obviamente, si no se encuentra ningún ticket para el evento en cuestión, o todos se encuentran ya marcados como consumidos, se lanza un mensaje de error específico (que, en una situación real, implicaría rechazar el acceso al evento).

El ciclo de la aplicación de demostración termina (o empieza, según se mire) restaurando los valores de fábrica de la tarjeta, permitiendo de esta forma que se repitan todos los pasos explicados con anterioridad. Desde el menú de selección de aplicaciones general (Fig. 6.3a), el botón `Format` nos dirige a la *app* `Factory Reset`, cuyo interfaz de usuario se muestra en la Fig. 6.7. Como puede verse, se incluye un desplegable que muestra los nombres de las distintas entidades bancarias. Este hecho puede sorprender a más de un usuario, ya que el identificador del banco emisor se encuentra almacenado dentro de la tarjeta. Sin embargo, su inclusión atiende a dos razones principales. La primera es que, igual que sucedía en el caso de la *app* de configuración del monedero electrónico, pretende reflejar que el formateo de la tarjeta es una operación tan crucial que probablemente requeriría que el propietario se dirigiera físicamente a una oficina de la entidad bancaria para su realización. La segunda, y quizás más importante, es que permite simular un intento de "fraude". Escoger un banco diferente del emisor sería equivalente a tratar de realizar una operación sensible sin disponer de las claves apropiadas. Como podrá comprobarse, sin ellas no es posible realizar el formateo de la tarjeta, y esto puede extenderse a cualquier otra acción que haya sido protegida con clave.

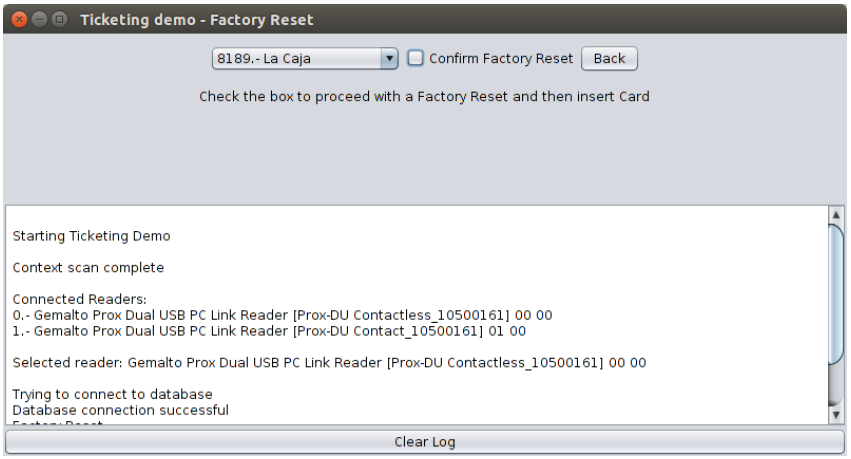


Figura 6.7: *App* de restauración de valores de fábrica

La particularidad más destacada de la *app* de formateo es que no basta con insertar la tarjeta para confirmar la ejecución de la operación. Por contra, la ventana incluye un *checkbox* que es necesario marcar explícitamente para que la orden de formateo se transmita a la tarjeta. En caso contrario no sucederá nada aunque acerquememos la tarjeta al campo del lector. La ejecución de esta operación provoca la eliminación de todas las aplicaciones que hayan sido creadas en la

tarjeta, incluyendo, obviamente, los ficheros que contuvieran. Asimismo, restaura la versión por defecto de la clave maestra. Estas acciones son tan indudablemente sensibles que se ha optado por añadir el *checkbox* incluso tratándose de una simple demostración. Si de todos modos se decide proceder con el formateo, se dispondrá de nuevo de una tarjeta Mifare DESFire "virgen" con la que se podrán seguir realizando pruebas.

El proceso completo que hemos descrito aquí pretende reflejar de forma simplificada las posibilidades que ofrece la librería desarrollada para el desarrollo de servicios reales mediante tarjetas Mifare DESFire. La disponibilidad de versiones interpretadas de los múltiples comandos de la API DESFire ha facilitado enormemente su combinación para la creación de operaciones más complejas. La decisión de ofrecer un interfaz gráfico a los usuarios de la demo, por contra, ha complicado significativamente el diseño de la aplicación, quedando reflejado en las clases implementadas, que serán descritas con mayor detalle en el próximo apartado.

6.2.2. Detalles de implementación

La aplicación de demostración es, evidentemente, mucho más compleja que el ejemplo simple que comentamos en la Sección 6.1. Por ello, requiere de una estructura de clases que sea capaz de manejar esta complejidad, incluyendo los diversos procesos que deben hallarse en ejecución simultáneamente. En concreto, aparte del entorno de interacción con el usuario, son necesarios dos servidores "remotos": el gestor de bases de datos, mediante una implementación de terceros (como MySQL), y al menos un proveedor de servicios bancarios, cuya funcionalidad se haya incluida en la clase `BankServer`. Además, la ejecución de cada una de las distintas operaciones lanza un proceso auxiliar específico, lo que impide que se bloquee el interfaz de usuario. En esta sección describiremos las principales funcionalidades de las clases que hemos desarrollado para realizar sus distintas tareas.

La mayoría de operaciones que pueden realizarse en la aplicación tienen una estructura similar. Por esa razón, y con el objetivo de realizar una explicación lo más completa posible sin caer en una excesiva reiteración, nos centraremos en un caso de uso concreto: la compra de entradas. Asumiremos que la tarjeta tiene correctamente configuradas las aplicaciones de monedero electrónico y de *ticketing*, y que se dispone de saldo suficiente. De este modo, el supuesto usuario lanzaría la demostración y se dirigiría directamente a la *app* `Purchase Tickets`. Esta *app*, como ya indicamos anteriormente, es la que refleja una mayor complejidad ya que requiere de la participación de todos los distintos agentes.

Tras comprobar que el Administrador de Recursos (el *daemon* `pcscd`, por ejemplo) y el servidor de bases de datos se encuentran en funcionamiento, el usuario de la aplicación de demostración debe ejecutar el lanzador de servidores bancarios (`BankServers`). Éste se encarga de iniciar un proceso por cada una de las entidades bancarias que se hallen definidas en la base de datos, abriendo un Socket en el puerto correspondiente a su número de identificación

bancario[4]. A partir de ese momento ya se hallarán en disposición de recibir peticiones por parte de la aplicación cliente, cuya ejecución se inicia desde la clase `WTDemo`.

El cliente ha sido diseñado usando un patrón Model-View-Controller (MVC), de tal forma que la lógica del programa se halla explícitamente separada de su presentación. En particular, el rol de modelo lo toma la clase `WTModel`, mientras que las funcionalidades de vista y controlador son asumidas por la clase `WTView`. En este sentido, el método main de `WTDemo` lanza un hilo para la vista que, en su construcción, genera un objeto `WTModel` con una referencia al propio `WTView`. En otras palabras, tanto la vista como el modelo tienen referencias cruzadas, lo que permite la ejecución mutua de métodos.

El objeto de clase `WTView` creado contiene, aparte de la referencia al modelo, una instancia de la clase interna `GUI`, que representa el interfaz de usuario concreto que se mostrará durante la ejecución de la aplicación. Éste se encarga de contener los distintos componentes gráficos que se le presentan al usuario, representados mediante instancias de múltiples clases internas específicas, incluyendo los *listeners* que detectan los eventos que se produzcan sobre ellos. Además, lanza procesos auxiliares "trabajadores" (`StartWorker`, `AppWorker`) que se encargarán de comunicar al modelo la acción que debe desarrollarse dependiendo de la *app* concreta en la que se encuentre la demostración en cada momento. La coexistencia de distintos procesos ejecutándose de forma simultánea podría causar conflictos de concurrencia. Por esa razón la vista también incluye atributos booleanos que permiten determinar qué operaciones pueden realizarse en cada momento, y que pueden modificarse desde el modelo. El principal de ellos es el que se ha denominado "`lock`".

Por su parte, el modelo, representado como una instancia de la clase `WTModel`, tiene como atributo la referencia a la vista que se le ha pasado a su constructor. Asimismo, es el encargado de comunicarse con el dispositivo lector mediante una implementación de `ComManager`[5], y con las bases de datos (`BankDBManager` y `TickDBManager`). Finalmente, también cuenta con atributos que permiten llevar un seguimiento de qué *app* se encuentra en ejecución y con qué opciones activadas.

La construcción del modelo inicia el proceso comentado en la Sección 6.1, produciéndose el reconocimiento del contexto y la obtención del listado de dispositivos lectores detectados. Este listado se envía a la vista para su presentación al usuario, que puede entonces seleccionar el deseado a través del desplegable o bien forzar una nueva detección de lectores mediante el botón `Re-Scan`. Estos componentes cuentan con *listeners* propios que comunican al modelo la acción que ha realizado el usuario para que se responda adecuadamente. Por ejemplo, al seleccionar un lector en concreto y pulsar sobre el botón `Ok` que se habrá activado, la vista informa al modelo de

[4]Por este motivo, es conveniente el uso de identificadores bancarios que no entren en conflicto con puertos habituales de nuestro sistema. En nuestro caso, por ejemplo, hemos empleado el rango `8189-8192`

[5]En la última versión de la librería se ha generado un objeto de la clase `JSCIOComManager` para tal propósito, aunque podría substituirse por uno de la clase `JPCSCComManager` si se desea utilizar tarjetas Mifare DESFire originales

que dicho lector debe emplearse de ahora en adelante, así como que se debe producir un cambio de *app*.

El funcionamiento de las pantallas correspondientes a los menús de selección de *app*, tanto el genérico primero como el específico de *ticketing* después, es relativamente sencillo. Cada uno de los botones mostrados dispone de un *listener* cuyo cometido es realizar un cambio de *app*. La verdadera complejidad aparece cuando se pretende realizar operaciones con una tarjeta Mifare DESFire. En el ejemplo que nos ocupa lo haremos a través del botón `Purchase`, cuyo *listener* provoca el cambio del panel mostrado para que aparezca el correspondiente a la compra de entradas, y notifica al modelo que debe iniciarse la ejecución de dicha *app*.

En general, cuando se ejecuta una *app* que requiere comunicación con la tarjeta, el modelo funciona esencialmente como una máquina de estados. Sus transiciones vienen determinadas por el estado del atributo `lock` de la vista y por la presencia o no de una tarjeta en el lector, como puede verse en el esquema de la Fig. 6.8. Cada *app* que se inicie lo hará desde el estado `S0`, en el que el modelo se bloqueará hasta que la vista genere el interfaz de usuario pertinente y, a partir de ese momento, se moverá de un estado a otro dependiendo de si el lector detecta o no una tarjeta en su campo hasta alcanzar el estado `S4` en el que se permitirá la realización de la operación que corresponda en dicha *app*.

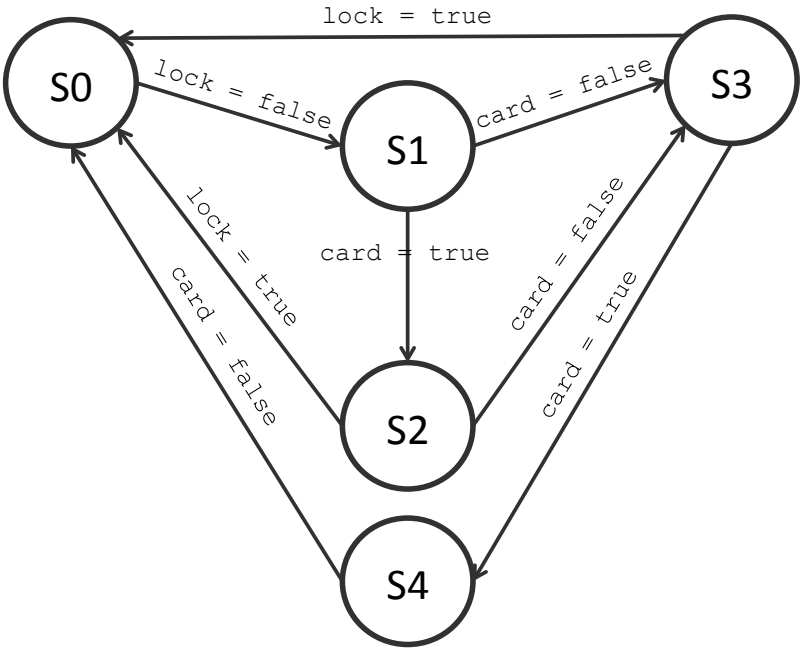


Figura 6.8: Transiciones entre estados en una *app* genérica

En el caso concreto que nos ocupa, el modelo, tras conocer que se desea iniciar la operación de compra de entradas, inicia una conexión con la base de datos de *ticketing* y permanece a la

espera mientras la vista inicia un proceso auxiliar (*worker*) específico para esta *app*. Este *worker* es una instancia de la clase `AppWorker`, que a su vez hereda de la clase `SwingWorker<T, T>` incluida en el paquete `javax.swing`. Su cometido es doble. Por una parte, lanza otro *worker*, esta vez de la clase `FillCompWorker`, que se encarga de rellenar los desplegables con el listado de eventos disponibles, y sus horarios correspondientes. Éste, tras asegurar que el modelo no cambia de estado marcando el `lock` como `true`, solicita la obtención de la información necesaria a la base de datos. Únicamente cuando se haya acabado de refrescar el contenido de los desplegables se liberará de nuevo el `lock`. Por otra parte, el primer *worker* pone en movimiento la máquina de estados del modelo. En otras palabras, es quien provoca que se realice la tarea adecuada en cada estado y se transicione al siguiente cuando se cumplan las condiciones para ello.

Los eventos recuperados de la base de datos son representados como objetos de la clase `Event`, mientras que la información relativa a la primera localidad libre en dicho evento (la que se ofrece para la compra) se encapsula en una instancia de la clase `Ticket`. Estas clases le indican al interfaz gráfico cómo debe presentarle la información recuperada al usuario. Una vez actualizado el contenido de todos los componentes del interfaz es cuando el `lock` vuelve a marcarse como `false` y el modelo pasa al estado `S1`.

El estado `S1` tiene una única labor: determinar si se detecta una tarjeta en el lector. Como explicamos anteriormente, la necesidad de impedir que se realicen operaciones en la tarjeta sin el explícito consentimiento del usuario nos fuerza a asegurar que la tarjeta se detecta porque el usuario la ha acercado al lector y no porque ya se encontrara ahí situada. Aunque el *middleware* habitualmente ofrece una funcionalidad específica para detectar eventos de inserción, en nuestro caso la estructura de la máquina de estados nos permite controlar la presencia o no de tarjeta en cada momento sin preocuparnos de lanzar interrupciones. Además, facilita enormemente la gestión de la interacción del usuario con el contenido de los desplegables (o de cualquier otro componente gráfico del resto de *apps*). Si seguimos lo que nos muestra la Fig. 6.8, podremos ver que desde el estado `S1` nos desplazaremos a `S2` en caso de que haya una tarjeta presente en el lector, y a `S3` en caso contrario. Desde cualquiera de estos dos estados, a su vez, podemos volver al estado inicial si en algún momento se produce un cambio en el interfaz gráfico. Así, cuando el usuario seleccione un evento o un horario distintos a los que se estaban mostrando, los listeners volverán a bloquear el `lock` forzando el retorno al estado `S0`.

Desde el estado `S3`, la detección de la presencia de una tarjeta llevará al modelo al estado `S4`, en el que se procederá a efectuar la operación correspondiente a la *app*. Tras establecer el canal de comunicación con la tarjeta, se realiza una llamada al método concreto que se encargará de dicha tarea. En el caso que nos ocupa, se trata del método `purchase()`. El Listado de Código 6.1 muestra la implementación de dicho método. Tras determinar si hay una localidad disponible para comprar, se inicia el envío de comandos, empezando por las distintas comprobaciones que aseguren que la tarjeta se halla correctamente preparada para la adquisición de una entrada. En ese momento, se obtiene la información acerca de la localidad y el usuario de la tarjeta,

comprobando que se encuentre registrado en la base de datos, y se procede a realizar el pago del importe pertinente. Finalmente, el ticket se marca como perteneciente al usuario de la tarjeta, se guarda en la tarjeta y se registra en la base de datos.

Listado de Código 6.1: Extracto del código de la clase `WTModel` correspondiente al método `purchase`

```
1  private void purchase(){
2
3      if(ticket == null){
4
5          getView().log("No available tickets for the selected event");
6          return;
7
8      }
9      try{
10
11         //Checking if the Wallet Application is present in the card
12         //in order to pay the purchase
13
14         selectMasterApp();
15         checkWalletApp();
16
17         //Checking if the Ticketing Application and the Tickets file
18         //are present in the card
19
20         selectMasterApp();
21         boolean isApp = checkTickApp();
22         if(!isApp)
23             throw new RuntimeException("Ticketing app not present " +
24                                        "in the card");
25
26         selectTickApp();
27         boolean isFile = checkTickFile();
28         if(!isFile)
29             throw new RuntimeException("Tickets file not present " +
30                                        "in the card");
31
32         //Checking if there is enough space in the Tickets file
33         //for another ticket
34
35         boolean freeRecords = checkTicketFileSpace();
36         if(!freeRecords)
37             throw new RuntimeException("There is not enough space " +
38                                        "for another ticket");
39
40         //Recovering the selected ticket
```

```
41
42          Ticket ticket = getTicket();
43
44          //Looking for the userID associated with the UID of the card
45          //at the Ticketing database
46
47          int user_id = getUserID();
48          getView().log("User successfully found in database. User ID: " +
49                          user_id);
50
51          //Proceed with the payment
52
53          int price = ticket.getPrice();
54
55          selectWalletApp();
56          int bankID = getBankID();
57
58          payment(bankID, price);
59
60          //Personalize ticket
61
62          getTicket().setUser(user_id);
63
64          //Write ticket in card
65
66          selectTickApp();
67
68          writeTicketRecord(ticket);
69
70          //Register ticket in database
71
72          registerTicket(ticket);
73
74          getView().log("Ticket successfully bought");
75          getView().show("Ticket successfully bought");
76
77      }catch(DFLException e){
78
79          if(e.getType() == ExType.INVALID_HANDLE){
80              getView().log(e.getMessage());
81              connect();
82              purchase();
83          }
84          else{
85              getView().error(e.getMessage());
86              return;
87          }
```

```
88
89      }catch(Exception e){
90
91          getView().error(e.getMessage());
92          return;
93
94      }
95
96      getView().fillComp(getCurrApp(), false, false);
97
98  }
```

La principal particularidad de las operaciones descritas anteriormente es que no todas son ejecutadas de forma local. Mientras que una operación común, como `writeTicketRecord`, llama directamente a comandos de una abstracción de una tarjeta (la variable `df`), el método correspondiente al pago (`payment`) en realidad se limita a solicitar la conexión con un servidor bancario concreto para que éste sea el encargado de ejecutar la operación de forma remota. En el Listado de Código 6.2 pueden verse claramente las diferencias entre estos dos tipos de métodos.

Listado de Código 6.2: Comparación de métodos con comunicación local (`writeTicketRecord`) y remota (`payment`) con una tarjeta Mifare DESfire, incluidos en la clase `WTModel`

```
1   private void writeTicketRecord(Ticket ticket){
2
3       getView().log("Writing ticket data in the file...");
4
5       DFResponse res = df.writeTicket(ticket);
6       getView().log(res);
7       if(!res.isOk())
8           throw new RuntimeException("Writing ticket data failed");
9       getView().log("");
10
11  }
12
13  private void payment(int bankID, int price){
14
15      getView().log("Connecting with bank number " + bankID);
16      bankConnection(bankID, "DEBIT " + price, df.getLegacy());
17
18  }
```

Al lector quizás le haya extrañado la llamada a `df.writeTicket(ticket)`, ya que no se trata de un comando incluido en la API DESFire. Según dijimos en el Capítulo 5, en la clase `DFCard` se incluyen abstracciones de los comandos nativos DESFire. Sin embargo, en muchas ocasiones es apropiado disponer de un nivel todavía más elevado de abstracción, sea por la conveniencia de agrupar diversos comandos en una única operación (por ejemplo, incluir la

autentificación necesaria antes de enviar el comando deseado) o porque la información se halla representada de forma más compleja que una simple trama de bytes (por ejemplo, una instancia de la clase `Ticket`). Por ello, la clase `WTDFCard` ofrece "comandos" especialmente adecuados para nuestra aplicación. Internamente emplea los comandos nativos DESFire, que envía a través de un `ComManager` del mismo modo que explicamos en el Capítulo 5.

Por otro lado, el método `payment` hace uso de una llamada a `bankConnection` para comunicar a la entidad bancaria adecuada qué cantidad debe ser descontada del saldo de la tarjeta. Con ese objetivo, `bankConnection` abre un `Socket` con la instancia de `BankServer` que se encuentre escuchando en el puerto correspondiente a su identificador bancario. En una aplicación real, obviamente, cada servidor tendría su propia dirección IP y, probablemente, compartirían puerto de escucha. Nuestra representación, sin embargo, cumple perfectamente nuestros propósitos sin complicar demasiado su construcción. A través del canal abierto se envía una trama textual que indica la operación concreta que se solicita (ej. `DEBIT 30`), así como una indicación de si la tarjeta conectada es o no una Mifare DESFire original. De este modo el servidor remoto podrá saber qué comandos deben ser enviados y con qué encapsulamiento. El intercambio de tramas se produce entonces a través de una instancia de `RemoteComManager`, que permite que los comandos se ejecuten a distancia sin necesidad de modificar su implementación interna en `DFCard`.

Una vez completados todos los pasos necesarios para realizar la operación deseada, el interfaz muestra un mensaje que informa al usuario del éxito. A partir de ese momento el modelo permanece en el estado `S4` hasta que la tarjeta se extrae del lector o bien hasta que se solicita la ejecución de una nueva operación (por ejemplo, empleando el botón `Back` de la pantalla), momento en el cual se vuelve al estado inicial `S0`. El ciclo se completa y la aplicación está de nuevo preparada para iniciar el proceso.

Al finalizar el uso de la aplicación de demostración, pulsando el botón de cerrar el interfaz en el borde de la ventana, terminarán las conexiones que hayamos abierto. Los servidores bancarios, sin embargo, permanecerán ejecutándose en segundo plano a menos que los terminemos explícitamente. Por esa razón, si tratamos de lanzarlos de nuevo aparecerá un error indicándonos que sus puertos correspondientes ya se encuentran en uso. Es indiferente optar entre mantener los servidores antiguos o terminar estos e iniciar unos nuevos. En ambos casos la ejecución de la aplicación debería realizarse sin problemas.

6.3. Conclusión

El uso de la librería `dflibrary` nos permite desarrollar aplicaciones que hagan uso de las funcionalidades de las tarjetas Mifare DESFire para ofrecer servicios de interés para los usuarios. El nivel de complejidad de dichas aplicaciones depende en gran medida del servicio concreto que se desee ofrecer y, especialmente, del grado de interactividad que se pretenda. La librería, en cualquier caso, proporciona unos bloques básicos consistentes que permiten la construcción de

aplicaciones tan simples o complejas como desee el desarrollador. La aplicación de demostración que hemos mostrado nos permite comprobar cómo conseguir este objetivo, a pesar de que, sin duda, todavía sean necesarias mejoras sustanciales para que el servicio pueda ser útil en escenarios reales.

7 | Conclusiones y líneas futuras

Este proyecto tuvo desde un principio el propósito de ofrecer a los desarrolladores que lo desearan una librería software que les permitiera generar aplicaciones que aprovecharan las ventajas de las tecnologías de comunicación sin contacto para prestar servicios de interés para los usuarios. Las decisiones en cuanto a infraestructura y diseño que hemos tomado desde su concepción, con su lógico refinamiento a lo largo del camino, nos han conducido a disponer de `dflibrary`, una librería Java que no sólo cumple con los objetivos marcados inicialmente, sino que además lo logra, en nuestra opinión, proporcionando un interfaz de programación práctico e intuitivo.

Buena parte de las ventajas que aporta la librería desarrollada quedan reflejadas en la aplicación de demostración que hemos creado. A pesar de la evidente complejidad estructural que supone una aplicación interactiva[1], aquellos fragmentos de código directamente relacionados con la comunicación con tarjetas Mifare DESFire han resultado mucho más sencillos de lo que cabría esperar gracias al interfaz de programación disponible. Asimismo, los servicios que se simulan en la aplicación permiten mostrar un amplio abanico de funcionalidades de entre las muchas que proporcionan las tarjetas sin contacto en general, y las Mifare DESFire en particular.

Los objetivos marcados se han logrado gracias a un enorme esfuerzo, que esperamos quede reflejado en el resultado que se presenta. Dicho esto, somos plenamente conscientes de que el trabajo no se ha acabado. O, al menos, no debería darse por acabado. Todavía quedan muchos detalles a depurar, muchas mejoras a realizar, y muchas ampliaciones a desarrollar. A lo largo de la presente memoria se han sugerido en múltiples ocasiones aspectos de la implementación que podrían ser mejorados. Sin ánimo de ser exhaustivos, a continuación presentamos una serie de propuestas que creemos que merecería la pena abordar.

Una de las decisiones más relevantes que se tomaron desde un buen principio fue el lenguaje de programación a emplear en el desarrollo de la librería. Java ha sido, sin ninguna duda, uno de los lenguajes más populares durante mucho tiempo. De hecho, en el año 2015 es considerado todavía como el más usado[2]. Por esa razón, su elección permite que la librería creada pueda resultarle útil a un gran número de desarrolladores. No obstante, no debemos obviar que, en

[1] De hecho, podríamos considerarla doblemente interactiva, ya que debe reaccionar tanto a su interfaz gráfico como a las acciones realizadas mediante el lector

[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

los últimos años, su prevalencia está reduciéndose, y otros lenguajes más ligeros, como Python, Javascript o Ruby, son cada vez más populares. Esto nos sugiere que quizás debería estudiarse la posibilidad adaptar nuestra librería a estos, u otros, lenguajes, para de este modo llegar al mayor número de desarrolladores posible.

El sistema operativo sobre el que se ha desarrollado, y testado, la librería también es un aspecto relevante que debería considerarse. Linux, y más concretamente Ubuntu, goza de una popularidad entre los profesionales de la informática que no se extiende al usuario doméstico. Por tanto, si se pretende desarrollar aplicaciones que resulten de utilidad para el gran público es imprescindible extender el alcance de la librería a otros sistemas operativos, principalmente a Windows y Mac OSX. En este último caso, no debería representar un esfuerzo excesivo, ya que su origen Unix y la presencia nativa de una versión de `pcsc-lite` sugieren que su adaptación debería depender únicamente de algunos detalles. Esta facilidad no creemos que se extienda a Windows. Sin embargo, el hecho de haber aislado el código de la librería de la implementación concreta de las capas de nivel inferior mediante el interfaz `ComManager` y que la API `pcsc-lite` sea una traslación casi directa de `winscard.h` nos hace ser relativamente optimistas sobre su viabilidad.

A nivel de hardware también pensamos que sería interesante ampliar el alcance de la librería. Por un lado, debería explorarse la posibilidad de aceptar tarjetas de otros modelos, como las Mifare Plus. Esto, obviamente, requeriría implementar toda su API de comunicación propia, lo que en principio supondría una elevada carga de trabajo. De todos modos, entendemos que el uso del interfaz `ComManager` facilitaría enormemente su adaptación. Por otro lado, no podemos ignorar el hecho que el número de dispositivos móviles con conectividad NFC se encuentra en claro incremento. Por eso pensamos que se debería estudiar la posibilidad de ajustar la librería para que permitiera usar estos dispositivos como lectores de tarjetas Mifare DESFire, de forma similar a como lo hace la aplicación Stratos que mencionamos en la introducción. En este sentido, teniendo en cuenta que Java es el lenguaje de programación escogido por Android para el desarrollo de aplicaciones destinadas a esta plataforma y que su núcleo es Linux, serían estos dispositivos los que, en un principio, deberían requerir un menor esfuerzo de adaptación.

En cuanto a los potenciales servicios a implementar, las posibilidades demasiado númerosas como para listarlas todas. En nuestra opinión, el hilo conductor de todas las posibles aplicaciones individuales debe ser la integración de servicios, de forma similar a cómo sucede en nuestra propia aplicación de demostración. Tanto las capacidades de seguridad del modelo Mifare DESFire como su funcionalidad multiaplicación deberían ser explotadas al máximo para ofrecer una amplia variedad de servicios con la comodidad que supondría necesitar una única tarjeta para todos ellos.

Finalmente, si nos centramos en la aplicación de demostración, sí podemos sugerir algunas mejoras que incrementen su realismo:

- Crear un menú desde el que lanzar y cancelar los distintos servidores bancarios

- Implementar un servidor de *ticketing* remoto
- Permitir el registro de nuevas tarjetas desde el propio servicio de ticketing
- Usar las entradas de la base de datos bancaria para recargar saldo en el monedero desde cuenta corriente de cliente
- Cifrar las comunicaciones con servidores remotos
- Permitir la selección de localidades
- Disponer de un formulario de configuración de preferencias, incluyendo la selección de la implementación del interfaz `ComManager` a utilizar
- Liberar automáticamente los tickets en la base de datos cuando se realiza un Factory Reset

Estamos convencidos de que, a pesar de todas las mejoras queden llevarse a a cabo, la entidad del trabajo realizado es suficiente como para convertirse en una contribución relevante tanto para investigadores académicos como para desarrolladores de software. Esperamos que este proyecto pueda convertirse en el punto de partida de una línea de trabajo concreta, y que en un futuro próximo podamos ver cómo la librería desarrollada puede emplearse en ofrecer servicios útiles e imaginativos a la sociedad.

Apéndices

A | Configuración de la Infraestructura

En este apéndice se pueden encontrar instrucciones para la configuración del sistema necesaria para el uso de la librería que se ha presentado en el presente informe. Estas instrucciones han sido testadas en sistemas Ubuntu 14.04. Deberían ser trasladables a la mayoría de sistemas basados en Unix, aunque es posible que fueran necesarios algunos ajustes en casos particulares.

0. Instalar las librerías `libudev-dev`, `libusb-1.0-0` y `libusb-1.0-0-dev`; por ejemplo, con `sudo apt-get install <lib>`

1. Descargar e instalar la última versión de `pcsc-lite`

   a) Descargar desde Alioth[1]

   b) Descomprimir y entrar en la carpeta generada desde la terminal

   c) Ejecutar el comando `./configure --prefix=/usr`

   d) Ejecutar el comando `sudo make`

   e) Ejecutar el comando `sudo make install`

   f) Comprobar que se haya instalado correctamente el *daemon* con `which pcscd`

2. Descargar e instalar la última versión de `ccid`

   a) Descargar desde Alioth[2]

   b) Descomprimir y entrar en la carpeta generada desde la terminal

   c) Ejecutar el comando `./configure`

   d) Ejecutar el comando `sudo make`

   e) Ejecutar el comando `sudo make install`

   f) Copiar el archivo `src/92_pcscd_ccid.rules` a la carpeta `/etc/rules/rules.d/`

[1] https://alioth.debian.org/frs/download.php/latestzip/39/pcsclite-latest.zip
[2] https://alioth.debian.org/frs/download.php/latestzip/112/ccid-latest.zip

3. Obtener `pcsc-tools` (Opcional)

   a) En este caso puede instalarse directamente desde el repositorio con:

   ```
   $ sudo apt-get install pcsc-tools
   ```

   Sin embargo, es recomendable descargar el código fuente desde la página personal del autor[3] y construir los ejecutables uno mismo con `make` dentro de la carpeta descomprimida. De esta forma nos aseguraremos de que la compilación se realice considerando la versión de `pcsc-lite` efectivamente instalada en nuestro sistema.

   b) Ejecutar el comando `sudo pcscd` para iniciar el *daemon*

   c) Lanzar la aplicación `./pcsc_scan`

   d) Conectar el dispositivo lector a un puerto USB y comprobar que es detectado

   e) Acercar una tarjeta al lector para asegurar que se detecta

4. Instalar `jpcsc` (Opcional)
   Como se ha mencionado en la Sección 4.2.3, en caso de que se desee emplear la librería para establecer comunicaciones con tarjetas Mifare DESFire originales es necesario recurrir a interfaces que permitan en envío de comandos en forma nativa, como `jpcsc`. De todas formas, es importante recordar que esta librería ya no tiene soporte y sus enlaces oficiales de descarga se encuentran rotos. Por tanto, debería ser evitada en la medida de lo posible. Asimismo, la última versión que se lanzó ya no puede compilarse con las versiones recientes de `pcsc-lite`, por lo que es necesario recurrir a una precompilación, que puede no ser compatible con el sistema del que disponga el usuario. En caso de que sea imprescindible su uso, deberá procederse del siguiente modo:

   a) Extraer el archivo `libjpcsc.so` proporcionado en la carpeta `/usr/lib`

   b) Mover el paquete `jpcsc.jar` al directorio `jre/lib/ext` en la ubicación de la variable de entorno `JAVA_HOME`, o alternativamente, ejecutar las aplicaciones usando `java -jar` incluyendo la localización del paquete en cuestión

[3] http://ludovic.rousseau.free.fr/softwares/pcsc-tools/

B | Configuración de las Bases de Datos

Con el objetivo de permitir a los usuarios probar el funcionamiento de la aplicación de demostración que se ha desarrollado, se incluye en este apéndice código sugerido para la reconstrucción de las dos bases de datos que se requieren: Servicios Bancarios y Venta de Entradas. En ambos casos se trata de estructuras muy sencillas que no pretenden representar las necesidades de una eventual aplicación real. Como podrá comprobarse, se propone una estructura con algunas tablas que no son actualmente empleadas en la aplicación, pero que deberían ser útiles para implementar las ampliaciones de funcionalidad que se han propuesto en el Capítulo 7.

B.1. Base de Datos Servicios Bancarios

El Listado de código B.1 puede ser usado para reconstruir la base de datos bancaria que se ha empleado en la aplicación de demostración de la librería.

Listado de Código B.1: Estructura de la Base de Datos de Servicios Bancarios

```
1  CREATE DATABASE IF NOT EXISTS bank_db;
2
3  USE bank_db;
4
5  DROP TABLE IF EXISTS banks;
6
7  CREATE TABLE banks(
8      bank_id int(11) unsigned NOT NULL,
9      name varchar(128) NOT NULL,
10     PRIMARY KEY (bank_id)
11 ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
12
13 DROP TABLE IF EXISTS clients;
14
15 CREATE TABLE clients(
16     user_id int(11) NOT NULL AUTO_INCREMENT,
17     username varchar(16) NOT NULL,
```

```
18      pwd varchar(16) NOT NULL,
19      PRIMARY KEY (user_id),
20      UNIQUE KEY user_id (user_id),
21      UNIQUE KEY username (username)
22  ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
23
24  DROP TABLE IF EXISTS accounts;
25
26  CREATE TABLE accounts(
27      id int(11) NOT NULL AUTO_INCREMENT,
28      bank_id int(11) unsigned NOT NULL,
29      user_id int(11) unsigned NOT NULL,
30      balance int(11) NOT NULL DEFAULT '0',
31      PRIMARY KEY (id)
32  ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
33
34  GRANT ALL ON bank_db.*
35      TO 'bankusr'@'localhost'
36      IDENTIFIED BY 'bankpwd';
```

Por su parte, el Listado B.2 muestra un ejemplo de población de la base de datos para que la aplicación de demostración pueda ejecutarse correctamente.

Listado de Código B.2: Ejemplo de población de la Base de Datos de Servicios Bancarios

```
1   USE bank_db;
2
3   TRUNCATE TABLE banks;
4
5   INSERT INTO banks (bank_id, name)
6       VALUES
7       (8189, "La Caja"),
8       (8190, "Banco Terrassa"),
9       (8191, "Banco Cantabria"),
10      (8192, "BVAA");
11
12  TRUNCATE TABLE clients;
13
14  INSERT INTO clients (username, pwd)
15      VALUES
16      ("fran", "1234"),
17      ("paco", "4321");
18
19  TRUNCATE TABLE accounts;
20
21  INSERT INTO accounts (bank_id, user_id, balance)
22      VALUES
23      (8189, 1, 1000),
```

```
24      (8190, 1, 500);
```

B.2. Base de Datos Servicios de Venta de Entradas

El código sugerido para la reconstrucción de la base de datos del servicio de venta de entradas se presenta en el Listado B.3.

Listado de Código B.3: Estructura de la Base de Datos de Servicios de Venta de Entradas

```
1  CREATE DATABASE IF NOT EXISTS chivas_db;
2
3  USE chivas_db;
4
5  DROP TABLE IF EXISTS events;
6
7  CREATE TABLE events (
8    event_id int(11) NOT NULL AUTO_INCREMENT,
9    description varchar(200),
10   date datetime NOT NULL,
11   PRIMARY KEY (event_id)
12 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
13
14 DROP TABLE IF EXISTS tickets;
15
16 CREATE TABLE tickets (
17   ticket_id int(11) NOT NULL AUTO_INCREMENT,
18   event_id int(11) NOT NULL,
19   user_id int(11) NOT NULL,
20   not_before datetime NOT NULL,
21   not_after datetime NOT NULL,
22   area int(11) unsigned  NOT NULL,
23   zone int(11) unsigned NOT NULL,
24   row int(11) unsigned NOT NULL,
25   seat int(11) unsigned NOT NULL,
26   price int(11) unsigned NOT NULL,
27   state int(11) NOT NULL,
28   PRIMARY KEY (ticket_id)
29 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
30
31 DROP TABLE IF EXISTS users;
32
33 CREATE TABLE users (
34   user_id int(11) NOT NULL AUTO_INCREMENT,
35   first_name varchar(40),
36   last_name varchar(40),
37   card_id varchar(14),
```

```
38    PRIMARY KEY (user_id)
39  ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
40
41  GRANT ALL ON chivas_db.*
42      TO 'chivasweb'@'localhost'
43      IDENTIFIED BY 'mysqlchivasweb';
```

En este caso, sin embargo, se presentan dos listados distintos para la población de esta base de datos, ya que se ha creído conveniente implementar un script Python que genere un gran número de tickets para que puedan ser consumidos sin necesidad de resetear la base de datos continuamente. En este sentido el Listado B.4 contiene el código SQL sugerido para poblar las tablas de eventos y usuarios, mientras que el Listado B.5 muestra el código Python sugerido para la generación de multitud de entradas.

Listado de Código B.4: Ejemplo de población de las tablas de eventos y usuarios de la Base de Datos de Servicios de Venta de Entradas

```
1  USE chivas_db;
2
3  TRUNCATE TABLE events;
4
5  INSERT INTO events(description, date)
6      VALUES
7      ('Queretaro vs Chivas','2015-05-03 17:00:00'),
8      ('Chivas vs Tijuana','2015-05-09 17:00:00'),
9      ('Tigre vs Chivas','2015-05-16 19:00:00'),
10     ('Queretaro vs Chivas','2015-05-20 21:00:00');
11
12 TRUNCATE TABLE users;
13
14 INSERT INTO users(first_name, last_name, card_id)
15     VALUES
16     ('Paco', 'Gomez Fernandez','047940B2F82B80'),
17     ('Alberto','Perez Diaz','04271B4190D030'),
18     ('Juan','Olmo Vazquez','04262F4190D030'),
19     ('Isabel','Lopez Garcia','04271C4190D030'),
20     ('Federico','Villar Arcos','04240F4190D030');
```

Listado de Código B.5: Ejemplo de población de la tabla de tickets de la Base de Datos de Servicios Bancarios

```
1  import MySQLdb
2  from random import randint
3  from datetime import timedelta
4
5  MIN_PRICE = 20
6  MAX_PRICE = 50
```

```
7
8  MIN_AREAS = 1
9  MAX_AREAS = 2
10 MIN_ZONES = 1
11 MAX_ZONES = 4
12 MIN_ROWS = 5
13 MAX_ROWS = 10
14 MIN_SEATS = 5
15 MAX_SEATS = 10
16
17 DB = 'chivas_db'
18 USER = 'chivasweb'
19 PWD = 'mysqlchivasweb'
20
21
22 def _merge_dicts(d1, d2):
23
24     merged = d1.copy()
25     merged.update(d2)
26     return merged
27
28
29 def connect_db(host = 'localhost'):
30
31     db = MySQLdb.connect(host = host,
32                  user = USER,
33                  passwd = PWD,
34                  db = DB)
35
36     return db, db.cursor()
37
38
39 def get_events(cur):
40
41     cur.execute('SELECT * FROM events')
42
43     return {event_id: event_date for event_id, event_date in
44         [(int(row[0]), row[2]) for row in cur.fetchall()]}
45
46 def create_row():
47
48     seats = []
49
50     for seat_num in range(1, randint(MIN_SEATS, MAX_SEATS) + 1):
51         seats.append({'seat': seat_num})
52
53     return seats
```

```
54
55
56  def create_zone():
57
58      zone_price = randint(MIN_PRICE, MAX_PRICE)
59      seats = []
60
61      for row_num in range(1, randint(MIN_ROWS, MAX_ROWS) + 1):
62          row = create_row()
63          row = [_merge_dicts({'price': zone_price, 'row': row_num}, seat)
64                  for seat in row]
65          seats += row
66
67      return seats
68
69
70  def create_area():
71
72      seats = []
73
74      for zone_num in range(1, randint(MIN_ZONES, MAX_ZONES) + 1):
75          zone = create_zone()
76          zone = [_merge_dicts({'zone': zone_num}, row)
77                  for row in zone]
78          seats += zone
79
80      return seats
81
82
83  def create_seats():
84
85      seats = []
86
87      for area_num in range(1, randint(MIN_AREAS, MAX_AREAS) + 1):
88          area = create_area()
89          area = [_merge_dicts({'area': area_num}, zone)
90                  for zone in area]
91          seats += area
92
93      return seats
94
95
96  def create_event_tickets(event_id, event_date):
97
98      not_before = event_date - timedelta(hours = 2)
99      not_after = event_date + timedelta(hours = 2)
100
```

```
101     tickets_seats = create_seats()
102
103     tickets_meta = {'event_id': event_id,
104                     'user_id': 0,
105                     'not_before': not_before,
106                     'not_after': not_after,
107                     'state': 0}
108
109     tickets = []
110
111     for seat in tickets_seats:
112         ticket = _merge_dicts(tickets_meta, seat)
113         tickets.append(ticket)
114
115     return tickets
116
117
118 def create_tickets(events):
119
120     tickets = []
121
122     for event_id in events.keys():
123         tickets += create_event_tickets(event_id, events[event_id])
124
125     return tickets
126
127
128 def delete_tickets(cur):
129
130     cur.execute('TRUNCATE TABLE tickets')
131
132
133 def ticket_to_str(ticket):
134
135     return "(%s, %s, '%s', '%s', %s, %s, %s, %s, %s, %s)" %\
136                 (ticket['event_id'],
137                     ticket['user_id'],
138                     ticket['not_before'],
139                     ticket['not_after'],
140                     ticket['state'],
141                     ticket['area'],
142                     ticket['zone'],
143                     ticket['row'],
144                     ticket['seat'],
145                     ticket['price'])
146
147
```

```
148 def insert_tickets(cur, tickets):
149 
150     query = 'INSERT INTO tickets (' +\
151                 'event_id, user_id, not_before, not_after, state, ' +\
152                 'area, zone, row, seat, price) ' +\
153                 'VALUES ' +\
154                 ' ,'.join([ticket_to_str(ticket) for ticket in tickets]) +\
155                 ';'
156     cur.execute(query)
157 
158 
159 if __name__=='__main__':
160 
161     db, cur = connect_db()
162 
163     events = get_events(cur)
164 
165     tickets = create_tickets(events)
166 
167     delete_tickets(cur)
168 
169     insert_tickets(cur, tickets)
170 
171     db.commit()
```

C | Documentación de la clase `DFCard`

De entre las clases que se han implementado para la construcción de la librería, sin duda es `DFCard` la que contiene los métodos más relevantes para el desarrollo de aplicaciones que se comuniquen con tarjetas Mifare DESFire. Aunque muchos detalles acerca de la implementación de los comandos incluidos en `DFCard` no pueden ser revelados debido al acuerdo de confidencialidad firmado, los desarrolladores deben poder tener acceso a las cabeceras de los métodos que tienen disponibles. Por ello creemos conveniente incluir la documentación del código correspondiente a dicha clase como apéndice de la memoria del proyecto. Como podrá comprobarse, la documentación ha sido redactada en inglés para una mayor difusión.

C.1. Package dflibrary.library

C.1.1. Class DFCard

Provides an abstraction of a Mifare DESFire Card, allowing to send native commands and receive the card responses through its methods

C.1.1.1. Declaration

```
1 public class DFCard
2  extends java.lang.Object
```

C.1.1.2. Constructor summary

DFCard(ComManager) Constructs an instance of class DFCard

C.1.1.3. Method summary

abortTransaction() Executes the interpreted version of the Abort Transaction command, as defined in the Mifare DESFire API.

AbortTransaction() Executes the raw version of the Abort Transaction command, as defined in the Mifare DESFire API.

Authenticate(byte[], byte[]) Executes the raw version of the Authenticate command, as defined in the Mifare DESFire API.

authenticate(int, byte[]) Executes the interpreted version of the Authenticate command, as defined in the Mifare DESFire API.

authenticate(int, DFKey) Executes the interpreted version of the Authenticate command, as defined in the Mifare DESFire API.

AuthenticateAES(byte[], byte[]) Executes the raw version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API.

authenticateAES(int, byte[]) Executes the interpreted version of the Authenticate AES command, as defined in the Mifare DESFire EV1 API.

authenticateAES(int, DFKey) Executes the interpreted version of the Authenticate AES command, as defined in the Mifare DESFire EV1 API.

AuthenticateISO(byte[], byte[]) Executes the raw version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API.

authenticateISO(int, byte[]) Executes the interpreted version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API.

authenticateISO(int, DFKey) Executes the interpreted version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API.

ChangeFileSettings(byte[], byte[]) Executes the raw version of the Change File Settings command, as defined in the Mifare DESFire API.

ChangeFileSettings(byte[], byte[], byte[]) Executes the raw version of the Change File Settings command, as defined in the Mifare DESFire API.

changeFileSettings(FID, ComSet, AccessRights, AccessRights) Executes the interpreted version of the Change File Settings command, as defined in the Mifare DESFire API.

ChangeKey(byte[], byte[]) Executes the raw version of the Change Key command, as defined in the Mifare DESFire API.

changeKey(int, DFKey, DFKey) Executes the interpreted version of the Change Key command, as defined in the Mifare DESFire API.

ChangeKeySettings(byte[]) Executes the raw version of the Change Key Settings command, as defined in the Mifare DESFire API.

changeKeySettings(KeySettings) Executes the interpreted version of the Change Key Settings command, as defined in the Mifare DESFire API.

ClearRecordFile(byte[]) Executes the raw version of the Clear Record File command, as defined in the Mifare DESFire API.

clearRecordFile(FID) Executes the interpreted version of the Clear Record File command, as defined in the Mifare DESFire API.

commitTransaction() Executes the interpreted version of the Commit Transaction command, as defined in the Mifare DESFire API.

CommitTransaction() Executes the raw version of the Commit Transaction command, as defined in the Mifare DESFire API.

createApplication(AID, KeySettings, int) Executes the interpreted version of the Create Application command, as defined in the Mifare DESFire API.

createApplication(AID, KeySettings, int, boolean, CipAlg) Executes the interpreted version of the Create Application command, as defined in the Mifare DESFire EV1 API.

createApplication(AID, KeySettings, int, boolean, CipAlg, ISOFileID, DFName) Executes the interpreted version of the Create Application command, as defined in the Mifare DESFire EV1 API.

CreateApplication(byte[], byte[], byte[]) Executes the raw version of the Create Application command, as defined in the Mifare DESFire API.

CreateApplication(byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Application command, as defined in the Mifare DESFire EV1 API.

CreateBackupDataFile(byte[], byte[], byte[], byte[]) Executes the raw version of the Create Backup Data File command, as defined in the Mifare DESFire API.

CreateBackupDataFile(byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Backup Data File command, as defined in the Mifare DESFire API.

createBackupDataFile(FID, ComSet, AccessRights, Size) Executes the interpreted version of the Create Backup Data File command, as defined in the Mifare DESFire API.

createBackupDataFile(FID, ISOFileID, ComSet, AccessRights, Size) Executes the interpreted version of the Create Backup Data File command, as defined in the Mifare DESFire API.

CreateCyclicRecordFile(byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Cyclic Record File command, as defined in the Mifare DESFire API.

CreateCyclicRecordFile(byte[], byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Cyclic Record File command, as defined in the Mifare DESFire API.

createCyclicRecordFile(FID, ComSet, AccessRights, Size, int) Executes the interpreted version of the Create Cyclic Record File command, as defined in the Mifare DESFire API.

createCyclicRecordFile(FID, ISOFileID, ComSet, AccessRights, Size, int) Executes the interpreted version of the Create Cyclic Record File command, as defined in the Mifare DESFire EV1 API.

CreateLinearRecordFile(byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Linear Record File command, as defined in the Mifare DESFire API.

CreateLinearRecordFile(byte[], byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Linear Record File command, as defined in the Mifare DESFire EV1 API.

createLinearRecordFile(FID, ComSet, AccessRights, Size, int) Executes the interpreted version of the Create Linear Record File command, as defined in the Mifare DESFire API.

createLinearRecordFile(FID, ISOFileID, ComSet, AccessRights, Size, int) Executes the interpreted version of the Create Linear Record File command, as defined in the Mifare DESFire EV1 API.

CreateStdDataFile(byte[], byte[], byte[], byte[]) Executes the raw version of the Create Standard Data File command, as defined in the Mifare DESFire API.

CreateStdDataFile(byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Standard Data File command, as defined in the Mifare DESFire API.

createStdDataFile(FID, ComSet, AccessRights, Size) Executes the interpreted version of the Create Standard Data File command, as defined in the Mifare DESFire API.

createStdDataFile(FID, ISOFileID, ComSet, AccessRights, Size) Executes the interpreted version of the Create Standard Data File command, as defined in the Mifare DESFire API.

CreateValueFile(byte[], byte[], byte[], byte[], byte[], byte[], byte[]) Executes the raw version of the Create Value File command, as defined in the Mifare DESFire EV1 API.

createValueFile(FID, ComSet, AccessRights, Value, Value, Value, boolean) Executes the interpreted version of the Create Value File command, as defined in the Mifare DESFire API.

createValueFile(FID, ComSet, AccessRights, Value, Value, Value, boolean, boolean) Executes the interpreted version of the Create Value File command, as defined in the Mifare DESFire EV1 API.

Credit(byte[], byte[]) Executes the raw version of the Credit command, as defined in the Mifare DESFire API.

credit(FID, Value, ComSet, AccessRights) Executes the interpreted version of the Credit command, as defined in the Mifare DESFire API.

credit(FID, Value, ValueFileSettings) Executes the interpreted version of the Credit command, as defined in the Mifare DESFire API.

Debit(byte[], byte[]) Executes the raw version of the Debit command, as defined in the Mifare DESFire API.

debit(FID, Value, ComSet, AccessRights) Executes the interpreted version of the Debit command, as defined in the Mifare DESFire API.

debit(FID, Value, ValueFileSettings) Executes the interpreted version of the Debit command, as defined in the Mifare DESFire API.

deleteApplication(AID) Executes the interpreted version of the Delete Application command, as defined in the Mifare DESFire API.

DeleteApplication(byte[]) Executes the raw version of the Delete Application command, as defined in the Mifare DESFire API.

DeleteFile(byte[]) Executes the raw version of the Delete File command, as defined in the Mifare DESFire API.

deleteFile(FID) Executes the interpreted version of the Delete File command, as defined in the Mifare DESFire API.

formatPICC() Executes the interpreted version of the Format PICC command, as defined in the Mifare DESFire API.

FormatPICC() Executes the raw version of the Format PICC command, as defined in the Mifare DESFire API.

freeMemory() Executes the interpreted version of the Free Memory command, as defined in the Mifare DESFire EV1 API.

FreeMemory() Executes the raw version of the Free Memory command, as defined in the Mifare DESFire EV1 API.

getApplicationIDs() Executes the interpreted version of the Get Application IDs command, as defined in the Mifare DESFire API.

GetApplicationIDs() Executes the raw version of the Get Application IDs command, as defined in the Mifare DESFire API.

getCardType()

getCardUID() Executes the interpreted version of the Get Card UID command, as defined in the Mifare DESFire EV1 API.

GetCardUID() Executes the raw version of the Get Card UID command, as defined in the Mifare DESFire EV1 API.

getDFNames() Executes the interpreted version of the Free Memory command, as defined in the Mifare DESFire EV1 API.

GetDFNames() Executes the raw version of the Free Memory command, as defined in the Mifare DESFire EV1 API.

getFileIDs() Executes the interpreted version of the Get File IDs command, as defined in the Mifare DESFire API.

GetFileIDs() Executes the raw version of the Get File IDs command, as defined in the Mifare DESFire API.

GetFileSettings(byte[]) Executes the raw version of the Get File Settings command, as defined in the Mifare DESFire API.

getFileSettings(FID) Executes the interpreted version of the Get File Settings command, as defined in the Mifare DESFire API.

getISOFileIDs() Executes the interpreted version of the Get ISO File IDs command, as defined in the Mifare DESFire EV1 API.

GetISOFileIDs() Executes the raw version of the Get ISO File IDs command, as defined in the Mifare DESFire EV1 API.

getKeySettings() Executes the interpreted version of the Get Key Settings command, as defined in the Mifare DESFire API.

GetKeySettings() Executes the raw version of the Get Key Settings command, as defined in the Mifare DESFire API.

GetKeyVersion(byte[]) Executes the raw version of the Get Key Version command, as defined in the Mifare DESFire API.

getKeyVersion(int) Executes the interpreted version of the Get Key Version command, as defined in the Mifare DESFire API.

getSession()

GetValue(byte[]) Executes the raw version of the Get Value command, as defined in the Mifare DESFire API.

getValue(FID, ComSet, AccessRights, boolean) Executes the interpreted version of the Get Value command, as defined in the Mifare DESFire EV1 API.

getValue(FID, ValueFileSettings) Executes the interpreted version of the Get Value command, as defined in the Mifare DESFire API.

getVersion() Executes the interpreted version of the Get Version command, as defined in the Mifare DESFire API.

GetVersion() Executes the raw version of the Get Version command, as defined in the Mifare DESFire API.

isoSelect() Performs a ISO Select command, needed for determining if Mifare DESFire API commands should be sent in wrapped or native format

ISOSelect() Performs a ISO Select command, needed for determining if Mifare DESFire API commands should be sent in wrapped or native format

LimitedCredit(byte[], byte[]) Executes the raw version of the Limited Credit command, as defined in the Mifare DESFire API.

limitedCredit(FID, Value, ComSet, AccessRights) Executes the interpreted version of the Limited Credit command, as defined in the Mifare DESFire API.

limitedCredit(FID, Value, ValueFileSettings) Executes the interpreted version of the Limited Credit command, as defined in the Mifare DESFire API.

ReadData(byte[], byte[], byte[]) Executes the raw version of the Read Data command, as defined in the Mifare DESFire API.

readData(FID, Size, Size, ComSet, AccessRights) Executes the interpreted version of the Read Data command, as defined in the Mifare DESFire API.

readData(FID, Size, Size, DataFileSettings) Executes the interpreted version of the Read Data command, as defined in the Mifare DESFire API.

ReadRecords(byte[], byte[], byte[]) Executes the raw version of the Read Records command, as defined in the Mifare DESFire API.

readRecords(FID, Size, Size, RecordFileSettings) Executes the interpreted version of the Read Records command, as defined in the Mifare DESFire API.

readRecords(FID, Size, Size, Size, int, ComSet, AccessRights) Executes the interpreted version of the Read Records command, as defined in the Mifare DESFire API.

resetSession() Re-starts the current communication session

selectApplication(AID) Executes the interpreted version of the Select Application command, as defined in the Mifare DESFire API.

SelectApplication(byte[]) Executes the raw version of the Select Application command, as defined in the Mifare DESFire API.

setCardType(CardType)

setCardType(PICCVersion)

SetConfiguration(byte[], byte[]) Executes the raw version of the Set Configuration command, as defined in the Mifare DESFire EV1 API.

setConfiguration(ConfigOption) Executes the interpreted version of the Set Configuration command, as defined in the Mifare DESFire EV1 API.

setSession(AuthType, int, byte[], boolean) Configures the current communication session

WriteData(byte[], byte[], byte[], byte[]) Executes the raw version of the Write Data command, as defined in the Mifare DESFire API.

writeData(FID, Size, Data, ComSet, AccessRights) Executes the interpreted version of the Write Data command, as defined in the Mifare DESFire API.

writeData(FID, Size, Data, DataFileSettings) Executes the interpreted version of the Write Data command, as defined in the Mifare DESFire API.

WriteRecord(byte[], byte[], byte[], byte[]) Executes the raw version of the Write Record command, as defined in the Mifare DESFire API.

writeRecord(FID, Size, Data, ComSet, AccessRights) Executes the interpreted version of the Write Record command, as defined in the Mifare DESFire API.

writeRecord(FID, Size, Data, RecordFileSettings) Executes the interpreted version of the Write Record command, as defined in the Mifare DESFire API.

C.1.1.4. Constructors

- **DFCard**

```
1 public DFCard(dflibrary.middleware.ComManager cm)
```

  - **Description**

    Constructs an instance of class DFCard

  - **Parameters**
    - `cm` – a previously connected ComManager instance

C.1.1.5. Methods

- **abortTransaction**

```
1 public DFResponse abortTransaction()
```

  - **Description**

    Executes the interpreted version of the Abort Transaction command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **AbortTransaction**

```
1 public byte[] AbortTransaction()
```

  - **Description**

    Executes the raw version of the Abort Transaction command, as defined in the Mifare DESFire API.

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **Authenticate**

```
1 public byte[] Authenticate(byte[] keyNum,byte[] keyData)
```

  - **Description**

    Executes the raw version of the Authenticate command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `keyNum` – a byte array representing the key number within the selected card application with which we want to authenticate
    - `keyData` – a byte array containing the key identified by `keyNum`
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **authenticate**

```
1 public DFResponse authenticate(int keyNum,byte[] keyData)
```

  - **Description**

    Executes the interpreted version of the Authenticate command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `keyNum` – the key number within the selected card application with which we want to authenticate
    - `keyData` – a byte array containing the key identified by `keyNum`
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **authenticate**

```
1 public DFResponse authenticate(int keyNum,security.DFKey key)
```

  - **Description**

    Executes the interpreted version of the Authenticate command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `keyNum` – the key number within the selected card application with which we want to authenticate
  - `key` – an instance of class `DFKey` containing the data about the key identified by `keyNum`
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **AuthenticateAES**

```
1 public byte[] AuthenticateAES(byte[] keyNum,byte[] keyData)
```

- **Description**

  Executes the raw version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.
- **Parameters**
  - `keyNum` – a byte array representing the key number within the selected card application with which we want to authenticate
  - `keyData` – a byte array containing the key identified by `keyNum`
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **authenticateAES**

```
1 public DFResponse authenticateAES(int keyNum,byte[] keyData)
```

- **Description**

  Executes the interpreted version of the Authenticate AES command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.
- **Parameters**
  - `keyNum` – the key number within the selected card application with which we want to authenticate. Response only includes the Status Code returned by the card.
  - `keyData` – a byte array containing the key identified by `keyNum`
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **authenticateAES**

```
1 public DFResponse authenticateAES(int keyNum,security.DFKey key)
```

  - **Description**

    Executes the interpreted version of the Authenticate AES command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

  - **Parameters**

    - `keyNum` – the key number within the selected card application with which we want to authenticate. Response only includes the Status Code returned by the card.
    - `key` – an instance of class `DFKey` containing the data about the key identified by `keyNum`

  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **AuthenticateISO**

```
1 public byte[] AuthenticateISO(byte[] keyNum,byte[] keyData)
```

  - **Description**

    Executes the raw version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

  - **Parameters**

    - `keyNum` – a byte array representing the key number within the selected card application with which we want to authenticate
    - `keyData` – a byte array containing the key identified by `keyNum`

  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **authenticateISO**

```
1 public DFResponse authenticateISO(int keyNum,byte[] keyData)
```

- **Description**

  Executes the interpreted version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `keyNum` – the key number within the selected card application with which we want to authenticate
  - `keyData` – a byte array containing the key identified by `keyNum`
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **authenticateISO**

```
1  public DFResponse authenticateISO(int keyNum,security.DFKey key)
```

- **Description**

  Executes the interpreted version of the Authenticate ISO command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `keyNum` – the key number within the selected card application with which we want to authenticate
  - `key` – an instance of class `DFKey` containing the data about the key identified by `keyNum`
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **ChangeFileSettings**

```
1  public byte[] ChangeFileSettings(byte[] fid,byte[] encFileSettings)
```

- **Description**

  Executes the raw version of the Change File Settings command, as defined in the Mifare DESFire API. This version of the command is needed if the current access rights require a previous authentication to be changed.

- **Parameters**

- `fid` – an instance of class `FID` representing the file identifier
- `encFileSettings` – a byte array containing the new file settings, transformed for security as required

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

ChangeFileSettings

```
1 public byte[] ChangeFileSettings(byte[] fid,byte[] comSet,byte[] ar)
```

- **Description**

  Executes the raw version of the Change File Settings command, as defined in the Mifare DESFire API. This version of the command will only succeed if the current access rights allow its change freely.

- **Parameters**
  - `fid` – a byte array representing the file identifier
  - `comSet` – a byte array representing the new communication settings to be set
  - `ar` – a byte array representing the new access rights to be set
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

changeFileSettings

```
1 public DFResponse changeFileSettings(param.FID fid,param.fileset.ComSet
    comSet,param.fileset.AccessRights newAR,param.fileset.AccessRights
    oldAR)
```

- **Description**

  Executes the interpreted version of the Change File Settings command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier
  - `comSet` – an instance of class `ComSet` representing the new communication settings to be set
  - `newAR` – an instance of class `AccessRights` representing the new access rights to be set

- `oldAR` – an instance of class `AccessRights` representing the current access rights of the file

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ ChangeKey

```
1 public byte[] ChangeKey(byte[] keyNum,byte[] encKeyData)
```

- **Description**

  Executes the raw version of the Change Key command, as defined in the Mifare DESFire API.

- **Parameters**

  - `keyNum` – a byte array representing the key number within the selected card application with which we want to authenticate
  - `encKeyData` – a byte array containing the new key, transformed for security as required

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ changeKey

```
1 public DFResponse changeKey(int keyNum,security.DFKey newKey,security.DFKey
      oldKey)
```

- **Description**

  Executes the interpreted version of the Change Key command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**

  - `keyNum` – the key number within the selected card application that we want to modify
  - `newKey` – an instance of class `DFKey` containing the new data about the key identified by `keyNum`
  - `oldKey` – an instance of class `DFKey` containing the old data about the key identified by `keyNum`

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **ChangeKeySettings**

```
1 public byte[] ChangeKeySettings(byte[] cipKeySettings)
```

  - **Description**

    Executes the raw version of the Change Key Settings command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `cipKeySettings` – a byte array containing the new key settings of the currently selected card application, transformed for security as required
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **changeKeySettings**

```
1 public DFResponse changeKeySettings(param.KeySettings ks)
```

  - **Description**

    Executes the interpreted version of the Change Key Settings command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `ks` – an instance of class `KeySettings` representing the new key settings of the currently selected card application
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **ClearRecordFile**

```
1 public byte[] ClearRecordFile(byte[] fid)
```

  - **Description**

    Executes the raw version of the Clear Record File command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `fid` – a byte array representing the file identifier of the file to be written in

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **clearRecordFile**

```
1 public DFResponse clearRecordFile(param.FID fid)
```

  - **Description**

    Executes the interpreted version of the Clear Record File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `fid` – an instance of class `FID` representing the file identifier of the file to be cleared
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **commitTransaction**

```
1 public DFResponse commitTransaction()
```

  - **Description**

    Executes the interpreted version of the Commit Transaction command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **CommitTransaction**

```
1 public byte[] CommitTransaction()
```

  - **Description**

    Executes the raw version of the Commit Transaction command, as defined in the Mifare DESFire API.

  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **createApplication**

```
1 public DFResponse createApplication(param.AID aid,param.KeySettings ks,int
    numOfKeys)
```

  - **Description**

    Executes the interpreted version of the Create Application command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `aid` – an instance of class `AID` representing the application identifier of the new application to be created in the card
    - `ks` – an instance of class `KeySettings` representing the desired key settings of the new application to be created
    - `numOfKeys` – the number of keys of the new application to be created
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **createApplication**

```
1 public DFResponse createApplication(param.AID aid,param.KeySettings ks,int
    numOfKeys,boolean ISOFidAllow,dflibrary.utils.security.CipAlg alg)
```

  - **Description**

    Executes the interpreted version of the Create Application command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `aid` – an instance of class `AID` representing the application identifier of the new application to be created in the card
    - `ks` – an instance of class `KeySettings` representing the desired key settings of the new application to be created
    - `numOfKeys` – the number of keys of the new application to be created
    - `ISOFidAllow` – a boolean indicating whether ISO file identifiers are allowed or not in the new application
    - `alg` – an instance of class `CipAlg representing the criptographic algorithm to be used in the security of the new application`

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **createApplication**

```
public DFResponse createApplication(param.AID aid,param.KeySettings ks,int
    numOfKeys,boolean ISOFidAllow,dflibrary.utils.security.CipAlg alg,param
    .ISOFileID isoFid,param.DFName name)
```

- **Description**

  Executes the interpreted version of the Create Application command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `aid` – an instance of class `AID` representing the application identifier of the new application to be created in the card
  - `ks` – an instance of class `KeySettings` representing the desired key settings of the new application to be created
  - `numOfKeys` – the number of keys of the new application to be created
  - `ISOFidAllow` – a boolean indicating whether ISO file identifiers are allowed or not in the new application
  - `alg` – an instance of class `CipAlg representing the criptographic algorithm to be used in the security of the new application`
  - `isoFid` – an instance of class `ISOFileID` representing the ISO file identifier of the new application to be created in the card
  - `name` – an instance of class `DFName` representing the DF name of the new application to be created in the card
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **CreateApplication**

```
public byte[] CreateApplication(byte[] aid,byte[] keySettings,byte[]
    numOfKeys)
```

- **Description**

  Executes the raw version of the Create Application command, as defined in the Mifare DESFire API.

- **Parameters**
  - `aid` – a byte array representing the application identifier of the new application to be created in the card
  - `keySettings` – a byte array representing the desired key settings of the new application to be created
  - `numOfKeys` – a byte array representing the number of keys of the new application to be created
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

CreateApplication

```
1 public byte[] CreateApplication(byte[] aid,byte[] ks,byte[] ks2,byte[] fid,
    byte[] name)
```

- **Description**

  Executes the raw version of the Create Application command, as defined in the Mifare DESFire EV1 API.
- **Parameters**
  - `aid` – a byte array representing the application identifier of the new application to be created in the card
  - `ks` – a byte array representing the desired key settings of the new application to be created
  - `ks2` – a byte array representing the number of keys of the new application to be created, as well as whether ISO file identifiers are allowed or not in the new application and the criptographic algorithm to be used
  - `fid` – a byte array representing the ISO file identifier of the new application to be created in the card
  - `name` – a byte array representing the DF name of the new application to be created in the card
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

CreateBackupDataFile

```
1 public byte[] CreateBackupDataFile(byte[] fid,byte[] comSet,byte[] ar,byte
    [] fileSize)
```

- **Description**

  Executes the raw version of the Create Backup Data File command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the new file
  - `comSet` – a byte array representing the communication settings to be set in the new file
  - `ar` – a byte array representing the new access rights to be set in the new file
  - `fileSize` – a byte array representing the size of the new file to be created
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **CreateBackupDataFile**

```
1 public byte[] CreateBackupDataFile(byte[] fid,byte[] isoFid,byte[] comSet,
    byte[] ar,byte[] fileSize)
```

- **Description**

  Executes the raw version of the Create Backup Data File command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the new file
  - `isoFid` – a byte array representing the ISO file identifier of the new file
  - `comSet` – a byte array representing the communication settings to be set in the new file
  - `ar` – a byte array representing the new access rights to be set in the new file
  - `fileSize` – a byte array representing the size of the new file to be created
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **createBackupDataFile**

```
1 public DFResponse createBackupDataFile(param.FID fid,param.fileset.ComSet
    comSet,param.fileset.AccessRights ar,param.Size fileSize)
```

- **Description**

  Executes the interpreted version of the Create Backup Data File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `fileSize` – an instance of class `Size` representing the size of the new file to be created
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **createBackupDataFile**

```
1 public DFResponse createBackupDataFile(param.FID fid,param.ISOFileID isoFid
    ,param.fileset.ComSet comSet,param.fileset.AccessRights ar,param.Size
    fileSize)
```

- **Description**

  Executes the interpreted version of the Create Backup Data File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file
  - `isoFid` – an instance of class `ISOFileID` representing the ISO file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `fileSize` – an instance of class `Size` representing the size of the new file to be created
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **CreateCyclicRecordFile**

```
1 public byte[] CreateCyclicRecordFile(byte[] fid,byte[] comSet,byte[] ar,
    byte[] recSize,byte[] maxNumOfRecords)
```

  - **Description**

    Executes the raw version of the Create Cyclic Record File command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `fid` – a byte array representing the file identifier of the new file
    - `comSet` – a byte array representing the communication settings to be set in the new file
    - `ar` – a byte array representing the new access rights to be set in the new file
    - `recSize` – a byte array representing the size of each record in the file
    - `maxNumOfRecords` – a byte array representing the maximum number of records that can be kept within the file at one time
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **CreateCyclicRecordFile**

```
1 public byte[] CreateCyclicRecordFile(byte[] fid,byte[] isoFid,byte[] comSet
    ,byte[] ar,byte[] recSize,byte[] maxNumOfRecords)
```

  - **Description**

    Executes the raw version of the Create Cyclic Record File command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `fid` – a byte array representing the file identifier of the new file
    - `isoFid` – a byte array representing the ISO file identifier of the new file
    - `comSet` – a byte array representing the communication settings to be set in the new file
    - `ar` – a byte array representing the new access rights to be set in the new file
    - `recSize` – a byte array representing the size of each record in the file
    - `maxNumOfRecords` – a byte array representing the maximum number of records that can be kept within the file at one time

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **createCyclicRecordFile**

```
1 public DFResponse createCyclicRecordFile(param.FID fid,param.fileset.ComSet
      comSet,param.fileset.AccessRights ar,param.Size recSize,int
    maxNumOfRecords)
```

- **Description**

  Executes the interpreted version of the Create Cyclic Record File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `recSize` – an instance of class `Size` representing the size of each record in the file
  - `maxNumOfRecords` – an integer representing the maximum number of records that can be kept within the file at one time
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **createCyclicRecordFile**

```
1 public DFResponse createCyclicRecordFile(param.FID fid,param.ISOFileID
    isoFid,param.fileset.ComSet comSet,param.fileset.AccessRights ar,param.
    Size recSize,int maxNumOfRecords)
```

- **Description**

  Executes the interpreted version of the Create Cyclic Record File command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file

- `isoFid` – an instance of class `ISOFileID` representing the ISO file identifier of the new file
- `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
- `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
- `recSize` – an instance of class `Size` representing the size of each record in the file
- `maxNumOfRecords` – an integer representing the maximum number of records that can be kept within the file at one time

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

■ **CreateLinearRecordFile**

```
1 public byte[] CreateLinearRecordFile(byte[] fid,byte[] comSet,byte[] ar,
    byte[] recSize,byte[] maxNumOfRecords)
```

- **Description**

  Executes the raw version of the Create Linear Record File command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the new file
  - `comSet` – a byte array representing the communication settings to be set in the new file
  - `ar` – a byte array representing the new access rights to be set in the new file
  - `recSize` – a byte array representing the size of each record in the file
  - `maxNumOfRecords` – a byte array representing the maximum number of records that can be created within the file
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

■ **CreateLinearRecordFile**

```
1 public byte[] CreateLinearRecordFile(byte[] fid,byte[] isoFid,byte[] comSet
    ,byte[] ar,byte[] recSize,byte[] maxNumOfRecords)
```

- **Description**

  Executes the raw version of the Create Linear Record File command, as defined in the Mifare DESFire EV1 API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the new file
  - `isoFid` – a byte array representing the ISO file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `recSize` – an instance of class `Size` representing the size of each record in the file
  - `maxNumOfRecords` – an integer representing the maximum number of records that can be created within the file
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **createLinearRecordFile**

```
1 public DFResponse createLinearRecordFile(param.FID fid,param.fileset.ComSet
      comSet,param.fileset.AccessRights ar,param.Size recSize,int
    maxNumOfRecords)
```

- **Description**

  Executes the interpreted version of the Create Linear Record File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `recSize` – an instance of class `Size` representing the size of each record in the file
  - `maxNumOfRecords` – an integer representing the maximum number of records that can be created within the file
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **createLinearRecordFile**

```
1 public DFResponse createLinearRecordFile(param.FID fid,param.ISOFileID
    isoFid,param.fileset.ComSet comSet,param.fileset.AccessRights ar,param.
    Size recSize,int maxNumOfRecords)
```

  - **Description**

    Executes the interpreted version of the Create Linear Record File command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `fid` – an instance of class `FID` representing the file identifier of the new file
    - `isoFid` – an instance of class `ISOFileID` representing the ISO file identifier of the new file
    - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
    - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
    - `recSize` – an instance of class `Size` representing the size of each record in the file
    - `maxNumOfRecords` – an integer representing the maximum number of records that can be created within the file
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **CreateStdDataFile**

```
1 public byte[] CreateStdDataFile(byte[] fid,byte[] comSet,byte[] ar,byte[]
    fileSize)
```

  - **Description**

    Executes the raw version of the Create Standard Data File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `a` – byte array representing the file identifier of the new file
    - `comSet` – a byte array representing the communication settings to be set in the new file

- `ar` – a byte array representing the new access rights to be set in the new file
- `fileSize` – in instance of class `Size` representing the size of the new file to be created

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

CreateStdDataFile

```
1 public byte[] CreateStdDataFile(byte[] fid,byte[] isoFid,byte[] comSet,byte
    [] ar,byte[] fileSize)
```

- **Description**

  Executes the raw version of the Create Standard Data File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the new file
  - `isoFid` – a byte array representing the ISO file identifier of the new file
  - `comSet` – a byte array representing the communication settings to be set in the new file
  - `ar` – a byte array representing the new access rights to be set in the new file
  - `fileSize` – a byte array representing the size of the new file to be created
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

createStdDataFile

```
1 public DFResponse createStdDataFile(param.FID fid,param.fileset.ComSet
    comSet,param.fileset.AccessRights ar,param.Size fileSize)
```

- **Description**

  Executes the interpreted version of the Create Standard Data File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file

- `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
- `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
- `fileSize` – an instance of class `Size` representing the size of the new file to be created

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

createStdDataFile

```
1 public DFResponse createStdDataFile(param.FID fid,param.ISOFileID isoFid,
    param.fileset.ComSet comSet,param.fileset.AccessRights ar,param.Size
    fileSize)
```

- **Description**

  Executes the interpreted version of the Create Standard Data File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file
  - `isoFid` – an instance of class `ISOFileID` representing the ISO file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `fileSize` – an instance of class `Size` representing the size of the new file to be created
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

CreateValueFile

```
1 public byte[] CreateValueFile(byte[] fid,byte[] comSet,byte[] ar,byte[]
    lowerLimit,byte[] upperLimit,byte[] value,byte[] limCredEnFreeGetVal)
```

- **Description**

  Executes the raw version of the Create Value File command, as defined in the Mifare DESFire EV1 API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the new file
  - `comSet` – a byte array representing the communication settings to be set in the new file
  - `ar` – a byte array representing the new access rights to be set in the new file
  - `lowerLimit` – a byte array representing the lower possible value in the file
  - `upperLimit` – a byte array representing the upper possible value in the file
  - `value` – a byte array representing the initial value in the file
  - `limCredEnFreeGetVal` – a byte array representing both whether Limited Credit operations are allowed or not and whether free read access to the current value is permited or not
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **createValueFile**

```
1 public DFResponse createValueFile(param.FID fid,param.fileset.ComSet comSet
    ,param.fileset.AccessRights ar,param.Value lowerLimit,param.Value
    upperLimit,param.Value value,boolean limCredEn)
```

- **Description**

  Executes the interpreted version of the Create Value File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `lowerLimit` – an instance of class `Value` representing the boundary which must not be passed by a Debit calculation on the current value. It may be negative, and is usually set to 0

- `upperLimit` – an instance of class `Value` representing the boundary which must not be passed by a Credit calculation on the current value. It has to be higher or equal to the lower limit
- `value` – an instance of class `Value` representing the initial value to be set in the file
- `limCredEn` – a boolean indicating whether Limited Credit operations are allowed in this file or not

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

createValueFile

```
1  public DFResponse createValueFile(param.FID fid,param.fileset.ComSet comSet
       ,param.fileset.AccessRights ar,param.Value lowerLimit,param.Value
       upperLimit,param.Value value,boolean limCredEn,boolean freeGetVal)
```

- **Description**

  Executes the interpreted version of the Create Value File command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the new file
  - `comSet` – an instance of class `ComSet` representing the communication settings to be set in the new file
  - `ar` – an instance of class `AccessRights` representing the new access rights to be set in the new file
  - `lowerLimit` – an instance of class `Value` representing the boundary which must not be passed by a Debit calculation on the current value. It may be negative, and is usually set to 0
  - `upperLimit` – an instance of class `Value` representing the boundary which must not be passed by a Credit calculation on the current value. It has to be higher or equal to the lower limit
  - `value` – an instance of class `Value` representing the initial value to be set in the file
  - `limCredEn` – a boolean indicating whether Limited Credit operations are allowed in this file or not
  - `freeGetValue` – a boolean indicating whether free read access to the current value is permited or not

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **Credit**

```
1 public byte[] Credit(byte[] fid,byte[] data)
```

  - **Description**

    Executes the raw version of the Credit command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
    - `data` – a byte array representing the amount to increase, transformed for security as required
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **credit**

```
1 public DFResponse credit(param.FID fid,param.Value value,param.fileset.
    ComSet comSet,param.fileset.AccessRights ar)
```

  - **Description**

    Executes the interpreted version of the Credit command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
    - `value` – an instance of class `Value` representing the amount to increase
    - `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
    - `ar` – an instance of `AccessRights` representing the current Access Rights of the file
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **credit**

```
1 public DFResponse credit(param.FID fid,param.Value value,param.fileset.
    ValueFileSettings fileSet)
```

- **Description**

  Executes the interpreted version of the Credit command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
  - `value` – an instance of class `Value` representing the amount to increase
  - `fileSet` – an instance of class `ValueFileSettings` representing the current File Settings of the file, as obtained by the Get File Settings command
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **Debit**

```
1 public byte[] Debit(byte[] fid,byte[] data)
```

- **Description**

  Executes the raw version of the Debit command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
  - `data` – a byte array representing the amount to decrease, transformed for security as required
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **debit**

```
1 public DFResponse debit(param.FID fid,param.Value value,param.fileset.
    ComSet comSet,param.fileset.AccessRights ar)
```

- **Description**

  Executes the interpreted version of the Debit command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**

- `fid` – an instance of class `FID` representing the file identifier of the file to be edited
- `value` – an instance of class `Value` representing the amount to decrease
- `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
- `ar` – an instance of `AccessRights` representing the current Access Rights of the file

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

debit

```
1 public DFResponse debit(param.FID fid,param.Value value,param.fileset.
    ValueFileSettings fileSet)
```

- **Description**

  Executes the interpreted version of the Debit command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
  - `value` – an instance of class `Value` representing the amount to decrease
  - `fileSet` – an instance of class `ValueFileSettings` representing the current File Settings of the file, as obtained by the Get File Settings command
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

deleteApplication

```
1 public DFResponse deleteApplication(param.AID aid)
```

- **Description**

  Executes the interpreted version of the Delete Application command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `aid` – an instance of class `AID` representing the application identifier of the new application to be created in the card

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **DeleteApplication**

```
1 public byte[] DeleteApplication(byte[] aid)
```

  - **Description**

    Executes the raw version of the Delete Application command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `aid` – a byte array representing the application identifier of the new application to be created in the card
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **DeleteFile**

```
1 public byte[] DeleteFile(byte[] fid)
```

  - **Description**

    Executes the raw version of the Delete File command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `fid` – a byte array representing the file identifier of the file to be deleted
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **deleteFile**

```
1 public DFResponse deleteFile(param.FID fid)
```

  - **Description**

    Executes the interpreted version of the Delete File command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

  - **Parameters**

  - `fid` – an instance of class `FID` representing the file identifier of the file to be deleted
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **formatPICC**

```
1 public DFResponse formatPICC()
```

  - **Description**

    Executes the interpreted version of the Format PICC command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **FormatPICC**

```
1 public byte[] FormatPICC()
```

  - **Description**

    Executes the raw version of the Format PICC command, as defined in the Mifare DESFire API.
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **freeMemory**

```
1 public DFResponse freeMemory()
```

  - **Description**

    Executes the interpreted version of the Free Memory command, as defined in the Mifare DESFire EV1 API. Response includes an object of class `Size`, which can be retrieved with its method `getFreeMemory()`.
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **FreeMemory**

```
1 public byte[] FreeMemory()
```

- **Description**

  Executes the raw version of the Free Memory command, as defined in the Mifare DESFire EV1 API.

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **getApplicationIDs**

```
1 public DFResponse getApplicationIDs()
```

- **Description**

  Executes the interpreted version of the Get Application IDs command, as defined in the Mifare DESFire API. Response includes an object of class `AIDS`, which can be retrieved with its method `getAIDs()`.

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **GetApplicationIDs**

```
1 public byte[] GetApplicationIDs()
```

- **Description**

  Executes the raw version of the Get Application IDs command, as defined in the Mifare DESFire API.

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **getCardType**

```
1 public CardType getCardType()
```

- **Returns** – the current card type

- **getCardUID**

```
1 public DFResponse getCardUID()
```

- **Description**

  Executes the interpreted version of the Get Card UID command, as defined in the Mifare DESFire EV1 API. Response includes an object of class `UIDRes`, which can be retrieved with its method `getUIDRes()`. The card UID can be retrieved directly with the method `getUID()`.

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

GetCardUID

```
1 public byte[] GetCardUID()
```

- **Description**

  Executes the raw version of the Get Card UID command, as defined in the Mifare DESFire EV1 API.

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

getDFNames

```
1 public DFResponse getDFNames()
```

- **Description**

  Executes the interpreted version of the Free Memory command, as defined in the Mifare DESFire EV1 API. Response includes an object of class `DFNamesRes`, which can be retrieved with its method `getDFNames()`.

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

GetDFNames

```
1 public byte[][] GetDFNames()
```

- **Description**

  Executes the raw version of the Free Memory command, as defined in the Mifare DESFire EV1 API.

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **getFileIDs**

```
1 public DFResponse getFileIDs()
```

  - **Description**

    Executes the interpreted version of the Get File IDs command, as defined in the Mifare DESFire API. Response includes an object of class `FIDS`, which can be retrieved with its method `getFIDs()`.

  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **GetFileIDs**

```
1 public byte[] GetFileIDs()
```

  - **Description**

    Executes the raw version of the Get File IDs command, as defined in the Mifare DESFire API.

  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **GetFileSettings**

```
1 public byte[] GetFileSettings(byte[] fid)
```

  - **Description**

    Executes the raw version of the Get File Settings command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `fid` – a byte array representing the file identifier
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **getFileSettings**

```
1 public DFResponse getFileSettings(param.FID fid)
```

- **Description**

  Executes the interpreted version of the Get File Settings command, as defined in the Mifare DESFire API. Response includes an object of class `FileSettings`, which can be retrieved with its method `getFileSettings()`. Depending on the type of file, this object can be casted to `DataFileSettings`, `ValueFileSettings`, or `RecordFileSettings`.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

■ **getISOFileIDs**

```
1 public DFResponse getISOFileIDs()
```

- **Description**

  Executes the interpreted version of the Get ISO File IDs command, as defined in the Mifare DESFire EV1 API. Response includes an object of class `ISOFileIDs`, which can be retrieved with its method `getISOFileIDs()`.

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

■ **GetISOFileIDs**

```
1 public byte[] GetISOFileIDs()
```

- **Description**

  Executes the raw version of the Get ISO File IDs command, as defined in the Mifare DESFire EV1 API.

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

■ **getKeySettings**

```
1 public DFResponse getKeySettings()
```

- **Description**

  Executes the interpreted version of the Get Key Settings command, as defined in the Mifare DESFire API. Response includes an object of class `KeySettingsRes`,

which contains an instance of `KeySettings` that can be retrieved with its method `getKeySettings()`, and the number of keys that can be obtained with its method `getNumOfKeys()`.

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **GetKeySettings**

```
1 public byte[] GetKeySettings()
```

  - **Description**

    Executes the raw version of the Get Key Settings command, as defined in the Mifare DESFire API.

  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **GetKeyVersion**

```
1 public byte[] GetKeyVersion(byte[] keyNum)
```

  - **Description**

    Executes the raw version of the Get Key Version command, as defined in the Mifare DESFire API.

  - **Parameters**
    - `keyNum` – a byte array representing the key number within the selected card application of which we want to retrieve its current version
  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **getKeyVersion**

```
1 public DFResponse getKeyVersion(int keyNum)
```

  - **Description**

    Executes the interpreted version of the Get Key Version command, as defined in the Mifare DESFire API. Response includes an object of class `KeyVersion`, which can be retrieved with its method `getKeyVersion()`

- **Parameters**
  - `keyNum` – the key number within the selected card application of which we want to retrieve its current version
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **getSession**

```
1 public DFSession getSession()
```

- **Returns** – the current session

▪ **GetValue**

```
1 public byte[] GetValue(byte[] fid)
```

- **Description**

  Executes the raw version of the Get Value command, as defined in the Mifare DESFire API.
- **Parameters**
  - `fid` – a byte array representing the file identifier of the file to be read from
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **getValue**

```
1 public DFResponse getValue(param.FID fid,param.fileset.ComSet comSet,param.
    fileset.AccessRights ar,boolean freeGetValue)
```

- **Description**

  Executes the interpreted version of the Get Value command, as defined in the Mifare DESFire EV1 API. Response includes an object of class `ValueRes`, which can be retrieved with its method `getValueRes()`
- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be read from

- `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
- `ar` – an instance of `AccessRights` representing the current Access Rights of the file
- `freeGetValue` – a boolean indicating whether free read access to the current value is permited or not

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

getValue

```
1 public DFResponse getValue(param.FID fid,param.fileset.ValueFileSettings
    fileSet)
```

- **Description**

  Executes the interpreted version of the Get Value command, as defined in the Mifare DESFire API. Response includes an object of class `ValueRes`, which can be retrieved with its method `getValueRes()`

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be read from
  - `fileSet` – an instance of class `ValueFileSettings` representing the current File Settings of the file to be read from, as obtained by the Get File Settings command
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

getVersion

```
1 public DFResponse getVersion()
```

- **Description**

  Executes the interpreted version of the Get Version command, as defined in the Mifare DESFire API. Response includes an object of class `PICCVersion`, which can be retrieved with its method `getPICCVersion()`. The card UID can be retrieved directly with the method `getUID()`.

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **GetVersion**

```
1 public byte[] GetVersion()
```

  - **Description**

    Executes the raw version of the Get Version command, as defined in the Mifare DESFire API.

  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **isoSelect**

```
1 public DFResponse isoSelect()
```

  - **Description**

    Performs a ISO Select command, needed for determining if Mifare DESFire API commands should be sent in wrapped or native format

  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **ISOSelect**

```
1 public byte[] ISOSelect()
```

  - **Description**

    Performs a ISO Select command, needed for determining if Mifare DESFire API commands should be sent in wrapped or native format

  - **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **LimitedCredit**

```
1 public byte[] LimitedCredit(byte[] fid,byte[] data)
```

  - **Description**

    Executes the raw version of the Limited Credit command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
  - `data` – a byte array representing the amount to increase, transformed for security as required
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **limitedCredit**

```
1 public DFResponse limitedCredit(param.FID fid,param.Value value,param.
    fileset.ComSet comSet,param.fileset.AccessRights ar)
```

- **Description**

  Executes the interpreted version of the Limited Credit command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.
- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
  - `value` – an instance of class `Value` representing the amount to increase
  - `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
  - `ar` – an instance of `AccessRights` representing the current Access Rights of the file
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **limitedCredit**

```
1 public DFResponse limitedCredit(param.FID fid,param.Value value,param.
    fileset.ValueFileSettings fileSet)
```

- **Description**

  Executes the interpreted version of the Limited Credit command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.
- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be edited
  - `value` – an instance of class `Value` representing the amount to increase

- `fileSet` – an instance of class `ValueFileSettings` representing the current File Settings of the file, as obtained by the Get File Settings command

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

ReadData

```
1 public byte[] ReadData(byte[] fid,byte[] offset,byte[] length)
```

- **Description**

  Executes the raw version of the Read Data command, as defined in the Mifare DESFire API.

- **Parameters**

  - `fid` – a byte array representing the file identifier of the file to be read from
  - `offset` – a byte array representing the initial position where the data should be read from the file
  - `length` – a byte array representing the number of data bytes to be read

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

readData

```
1 public DFResponse readData(param.FID fid,param.Size offset,param.Size
    length,param.fileset.ComSet comSet,param.fileset.AccessRights ar)
```

- **Description**

  Executes the interpreted version of the Read Data command, as defined in the Mifare DESFire API. Response includes an object of class `DataRes`, which can be retrieved with its method `getDataRes()`.

- **Parameters**

  - `fid` – an instance of class `FID` representing the file identifier of the file to be read from
  - `offset` – an instance of class `Size` representing the initial position where the data should be read from the file
  - `length` – an instance of class `Size` representing the number of data bytes to be read

- `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
- `ar` – an instance of `AccessRights` representing the current Access Rights of the file

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

■ **readData**

```
1 public DFResponse readData(param.FID fid,param.Size offset,param.Size
    length,param.fileset.DataFileSettings fileSet)
```

- **Description**

  Executes the interpreted version of the Read Data command, as defined in the Mifare DESFire API. Response includes an object of class `DataRes`, which can be retrieved with its method `getDataRes()`

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be read from
  - `offset` – an instance of class `Size` representing the initial position where the data should be read from the file
  - `length` – an instance of class `Size` representing the number of data bytes to be read
  - `fileSet` – an instance of class `DataFileSettings` representing the current File Settings of the file to be read from, as obtained by the Get File Settings command
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

■ **ReadRecords**

```
1 public byte[] ReadRecords(byte[] fid,byte[] offset,byte[] length)
```

- **Description**

  Executes the raw version of the Read Records command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the file to be read from

- `offset` – a byte array representing the initial record where the data should be read
- `length` – a byte array containing the number of records to be read

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **readRecords**

```
1 public DFResponse readRecords(param.FID fid,param.Size offset,param.Size
    length,param.fileset.RecordFileSettings fileSet)
```

- **Description**

  Executes the interpreted version of the Read Records command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be read from
  - `offset` – an instance of class `Size` representing the initial record where the data should be read
  - `length` – an instance of class `Data` containing the number of records to be read
  - `fileSet` – an instance of class `RecordFileSettings` representing the current File Settings of the file to be read from, as obtained by the Get File Settings command
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **readRecords**

```
1 public DFResponse readRecords(param.FID fid,param.Size offset,param.Size
    length,param.Size recSize,int currNumOfRecords,param.fileset.ComSet
    comSet,param.fileset.AccessRights ar)
```

- **Description**

  Executes the interpreted version of the Read Records command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be read from

- `offset` – an instance of class `Size` representing the initial record where the data should be read
- `length` – an instance of class `Data` containing the number of records to be read
- `recSize` – an instance of class `Size` representing the size of the records stored in the file
- `currNumOfRecords` – an integer indicating the current number of records stored in the file
- `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
- `ar` – an instance of `AccessRights` representing the current Access Rights of the file

- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **resetSession**

```
1 public void resetSession()
```

- **Description**

  Re-starts the current communication session

- **selectApplication**

```
1 public DFResponse selectApplication(param.AID aid)
```

- **Description**

  Executes the interpreted version of the Select Application command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `aid` – an instance of class `AID` representing the application identifier of the application to be selected
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **SelectApplication**

```
1 public byte[] SelectApplication(byte[] aid)
```

- **Description**

  Executes the raw version of the Select Application command, as defined in the Mifare DESFire API.

- **Parameters**
  - `aid` – a byte array representing the application identifier of the application to be created in the card
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **setCardType**

```
1 public void setCardType(CardType ct)
```

- **Parameters**
  - `ct` – the current card type

▪ **setCardType**

```
1 public void setCardType(param.PICCVersion version)
```

- **Parameters**
  - `the` – version of the card

▪ **SetConfiguration**

```
1 public byte[] SetConfiguration(byte[] opt,byte[] encData)
```

- **Description**

  Executes the raw version of the Set Configuration command, as defined in the Mifare DESFire EV1 API.

- **Parameters**
  - `opt` – a byte array representing the configuration option to be set in the card
  - `encData` – a byte array representing the data to be set in the card, transformed for security as required

- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

- **setConfiguration**

```
1 public DFResponse setConfiguration(param.ConfigOption opt)
```

  - **Description**

    Executes the interpreted version of the Set Configuration command, as defined in the Mifare DESFire EV1 API. Response only includes the Status Code returned by the card.

  - **Parameters**
    - `opt` – an instance of class `ConfigOption including the configuration option to be set in the card and the corresponding data`
  - **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

- **setSession**

```
1 public void setSession(security.AuthType auth,int authKeyNum,byte[]
    sessionKey,boolean PICCAuth)
```

  - **Description**

    Configures the current communication session

  - **Parameters**
    - `auth` – an instance of `AuthType` indicating the current type of authentication
    - `authKeyNum` – an integer indicating the number of the key currently authenticated
    - `sessionKey` – a byte array containing the current session key
    - `PICCAuth` – a boolen indicating whether the card is currently successfully authenticated or not

- **WriteData**

```
1 public byte[] WriteData(byte[] fid,byte[] offset,byte[] length,byte[] data)
```

  - **Description**

    Executes the raw version of the Write Data command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the file to be written in
  - `offset` – a byte array representing the initial position where the data should be written within the file
  - `length` – a byte array representing the number of data bytes to be written
  - `data` – a byte array containing the data bytes to be written, transformed for security as required
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **writeData**

```
1 public DFResponse writeData(param.FID fid,param.Size offset,param.Data data
    ,param.fileset.ComSet comSet,param.fileset.AccessRights ar)
```

- **Description**

  Executes the interpreted version of the Write Data command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.
- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be written in
  - `offset` – an instance of class `Size` representing the initial position where the data should be written within the file
  - `data` – an instance of class `Data` containing the data bytes to be written
  - `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
  - `ar` – an instance of `AccessRights` representing the current Access Rights of the file
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **writeData**

```
1 public DFResponse writeData(param.FID fid,param.Size offset,param.Data data
    ,param.fileset.DataFileSettings fileSet)
```

- **Description**

  Executes the interpreted version of the Write Data command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be written in
  - `offset` – an instance of class `Size` representing the initial position where the data should be written within the file
  - `data` – an instance of class `Data` containing the data bytes to be written
  - `fileSet` – an instance of class `DataFileSettings` representing the current File Settings of the file to be written in, as obtained by the Get File Settings command
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **WriteRecord**

```
1 public byte[] WriteRecord(byte[] fid,byte[] offset,byte[] length,byte[]
    data)
```

- **Description**

  Executes the raw version of the Write Record command, as defined in the Mifare DESFire API.

- **Parameters**
  - `fid` – a byte array representing the file identifier of the file to be written in
  - `offset` – a byte array representing the initial position where the data should be written within the new record
  - `length` – a byte array representing the number of data bytes to be written
  - `data` – a byte array containing the data bytes to be written, transformed for security as required
- **Returns** – a byte array representing the response obtained from the Mifare DESFire card to the transmitted command

▪ **writeRecord**

```
1 public DFResponse writeRecord(param.FID fid,param.Size offset,param.Data
    data,param.fileset.ComSet comSet,param.fileset.AccessRights ar)
```

- **Description**

  Executes the interpreted version of the Write Record command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be written in
  - `offset` – an instance of class `Size` representing the initial position where the data should be written within the new record
  - `data` – an instance of class `Data` containing the data bytes to be written
  - `comSet` – an instance of `ComSet` representing the current Communication Settings of the file
  - `ar` – an instance of `AccessRights` representing the current Access Rights of the file
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

▪ **writeRecord**

```
1 public DFResponse writeRecord(param.FID fid,param.Size offset,param.Data
    data,param.fileset.RecordFileSettings fileSet)
```

- **Description**

  Executes the interpreted version of the Write Record command, as defined in the Mifare DESFire API. Response only includes the Status Code returned by the card.

- **Parameters**
  - `fid` – an instance of class `FID` representing the file identifier of the file to be written in
  - `offset` – an instance of class `Size` representing the initial position where the data should be written within the new record
  - `data` – an instance of class `Data` containing the data bytes to be written
  - `fileSet` – an instance of class `RecordFileSettings` representing the current File Settings of the file to be written in, as obtained by the Get File Settings command
- **Returns** – an instance of class `DFResponse` containing the response obtained from the Mifare DESFire card to the transmitted command

Referencias

[1] Advanced Card Systems Limited. *ACR122U NFC Reader. Application Programming Interface [1.2]*, August 2008.

[2] Advanced Card Systems Limited. *ACR122U NFC Reader. SDK User Manual [1.7]*, October 2008.

[3] Advanced Card Systems Limited. *ACR122U NFC Reader. Technical Specification [1.0]*, August 2008.

[4] Avoine, G. RFID Security. In *Encyclopedia of Cryptography and Security*, H. C. A. v. Tilborg and S. Jajodia, Eds. Springer, 2011.

[5] Barclay, K., and Savage, J. *Object-Oriented Design with UML and Java*. Elsevier Butterworth-Heinemann, 2004.

[6] Block, J. *Effective Java*, 2 ed. Addison-Wesley Press, Inc., 2008.

[7] Coskun, V., Ozdenizci, B., and Ok, K. A Survey on Near Field Communication (NFC) Technology. *Wireless Personal Communications 71*, 3 (2013), 2259–2294.

[8] De Soete, M. Smart Card. In *Encyclopedia of Cryptography and Security*, H. C. A. v. Tilborg and S. Jajodia, Eds. Springer, 2011.

[9] Dobkin, D. M. *The RF in RFID. Passive UHF RFID in Practice*. Communictions Engineering Series. Newnes, 2008.

[10] Dworkin, M. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. National Institute of Standards and Technology (NIST), May 2005.

[11] Finkenzeller, K. *RFID Handbook. Fundamentals and Applications in Contactless Smart Cards and Identification*, 2 ed. John Wiley & Sons, Inc., 2003.

[12] Gemalto. *Prox-DU & Prox-SU. Dual Interface USB Smart Card Reader. PC/SC Guide [Release A]*, February 2011.

[13] Gemalto. *Prox-DU & Prox-SU. Dual Interface USB Smart Card Reader. Reference Manual [Release D]*, February 2011.

[14] Horstmann, C. S., and Cornell, G. *Core Java 2*, 7 ed. Pearson Prentice Hall, 2006.

[15] Huseman, D. Standards in the Smart Card World. *Computer Networks 36*, 4 (July 2001), 473–487.

[16] Monperrus, M. How to Write Good Object-oriented API Documentation. http://www.monperrus.net/martin/how-to-write-good-API-documentation.

[17] NXP Semiconductors. *AN10787. Mifare Application Directory (MAD) [Rev 0.7]*, July 2010.

[18] NXP Semiconductors. *AN10922. Symmetric Key Diversifications [Rev 1.3]*, March 2010.

[19] NXP Semiconductors. *MF3ICDx21_41_81. Mifare DESFire EV1 Contactless Multi-application IC. Product Short Data Sheet [Rev 3.1]*, December 2010.

[20] NXP Semiconductors. *AN10833. Mifare Type Identification Procedure [Rev 3.2]*, August 2011.

[21] NXP Semiconductors. *AN10957. Generic Control Data Model [Rev 1.1]*, March 2011.

[22] NXP Semiconductors. *AN10969. System Level Security Measures of Mifare Installations [Rev 1.0]*, August 2011.

[23] NXP Semiconductors. *NFC Tags. A Technical Introduction, Applications and Products [Rev 1.3]*, December 2011.

[24] NXP Semiconductors. *AN11004. Mifare DESFire as Type 4 Tag [Rev 2.2]*, January 2012.

[25] Oswald, D., and Paar, C. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems* (Berlin, Heidelberg, 2011), CHES'11, Springer-Verlag, pp. 207–222.

[26] Philips Semiconductors. Mifare DESFire Security.

[27] Philips Semiconductors. *Mifare DESFire MF3 IC D40. Product Short Form Specification [Rev 3.0]*, April 2004.

[28] Philips Semiconductors. Mifare DESFire & ISO 14443, 2006.

[29] Philips Semiconductors. Mifare DESFire Commands, 2006.

[30] Philips Semiconductors. Mifare DESFire Functionality, 2006.

[31] Philips Semiconductors. Mifare DESFire Product Overview, 2006.

[32] Rankl, W., and Effing, W. *Smart Card Handbook*, 3 ed. John Wiley & Sons, Inc., 2003.

[33] Vauclair, M. Contactless Cards. In *Encyclopedia of Cryptography and Security*, H. C. A. v. Tilborg and S. Jajodia, Eds. Springer, 2011, p. 249.

[34] Vauclair, M. NFC. In *Encyclopedia of Cryptography and Security*, H. C. A. v. Tilborg and S. Jajodia, Eds. Springer, 2011.