Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

DISEÑO DE UN LENGUAJE DE DOMINIO ESPECÍFICO PARA LA GESTIÓN DE MAPAS DE MEMORIA DE TARJETAS SIN CONTACTO

Trabajo específico propuesto por el alumno

Autor: GABRIEL GUERRA FERNÁNDEZ

Tutor: RUBÉN HERADIO GIL

[Página en blanco]

Resumen

El trabajo se centra en la tecnología de tarjetas sin contacto, especialmente en las tarjetas MiFare, ampliamente utilizadas en diversos sectores por su facilidad de uso y seguridad. Sin embargo, el acceso y edición de los datos en estas tarjetas pueden ser complejos y requieren conocimientos especializados. Para abordar esto, se propone la parametrización de la relación de datos entre diferentes tarjetas mediante un lenguaje específico de dominio (DSL). Este enfoque permite establecer una correspondencia entre los datos de las tarjetas y tipos de datos definidos, brindando un formato más comprensible para trabajar con ellas.

En este contexto, se ha desarrollado una aplicación que utiliza el DSL para la edición de datos de tarjetas MiFare, proporcionando una interfaz gráfica intuitiva y sencilla para los usuarios sin experiencia en programación. Además, el software permite realizar respaldos y clonar datos entre diferentes tarjetas. Esta solución mejora la eficiencia, productividad y flexibilidad en el desarrollo de software, así como la comprensión y mantenimiento de la solución implementada.

El enfoque de parametrización en forma de DSL ofrece una solución innovadora y efectiva para trabajar con tarjetas MiFare, proporcionando un mayor control y facilidad de uso en la manipulación de datos. Con esta herramienta, se logra una gestión más segura y conveniente de la información almacenada en las tarjetas, lo que representa una ventaja significativa en diversos sectores que utilizan esta tecnología.

Palabras clave

Tarjeta, MiFare, DESFire, Memoria, Edición.

[Página en blanco]

Contenido

Lista de figuras

Lista de tablas

Capítulo 1. Introducción

En este capítulo introductorio, se analizará a fondo un problema identificado junto con la solución propuesta para abordarlo. En primer lugar, se explorará la definición del problema en cuestión, examinando sus implicaciones y desafíos. Posteriormente, se presentará la solución, detallando su enfoque y características clave. Además, se resumirán los resultados obtenidos a través de la implementación de esta solución, destacando los logros y contribuciones significativas. Por último, se proporcionará información sobre el software desarrollado, incluyendo un enlace al repositorio público correspondiente, lo que permitirá a los interesados acceder y explorar más a fondo la implementación y funcionalidades de la solución.

Sección 1.1. Definición del problema

La tecnología de las tarjetas sin contacto se ha vuelto cada vez más popular en diferentes sectores debido a su facilidad de uso y seguridad. En particular, las tarjetas MiFare como las que se muestran en la Figura 1 son ampliamente utilizadas en aplicaciones como el transporte público, la identificación y la banca debido a su capacidad de almacenamiento, velocidad de lectura y escritura, y su capacidad de encriptación. Sin embargo, la lectura y la edición de los datos de estas tarjetas puede ser un proceso complejo y requiere conocimientos especializados en programación y criptografía.

Figura 1 Tarjetas MiFare

Una tarjeta MiFare es una tarjeta inteligente basada en tecnología de radiofrecuencia (RFID) que utiliza el estándar ISO 14443A. Estas tarjetas funcionan mediante la comunicación inalámbrica entre una antena integrada en la tarjeta y un lector compatible. Cuando una tarjeta MiFare se acerca al lector, se establece una comunicación inalámbrica a corta distancia. La tarjeta MiFare contiene una memoria interna organizada en bloques, donde se almacenan los datos. Estos bloques pueden ser de diferentes tipos, como bloques de datos, bloques de autenticación y bloques de control (tráiler).

Para acceder a los datos almacenados en las tarjetas MiFare, es necesario autenticarse utilizando un método de autenticación específico. Esto implica que el lector/escritor y la tarjeta deben intercambiar información de autenticación para verificar que tienen los permisos necesarios para acceder a los datos. Existiendo diferentes claves para acceder a los mismos datos se puede establecer diferentes niveles de acceso y autorización a los datos. Esto permite una gestión más granular y controlada de quién puede acceder a qué áreas de memoria o que funcionalidades puede acometer contra la tarjeta (por ejemplo: sólo lectura, lectura-escritura cambio de claves).

En sistemas donde se necesita una mayor seguridad estas claves podrían no ser fijas, sino que podrían ser claves diversificadas las cuales se calculan mediante algoritmos criptográficos a partir de una semilla, como por ejemplo el identificador único de la tarjeta (UID). Así se brinda de una capa adicional de protección individual a cada tarjeta. Esto significa que incluso si un atacante lograse obtener la clave de una tarjeta particular, no podrá utilizarla para acceder a otras tarjetas del sistema. Estos algoritmos criptográficos se suelen alojar en los denominados SAM (Secure Access Module, por sus siglas en inglés), un SAM es un módulo de acceso seguro utilizado para proteger y gestionar las claves criptográficas en sistemas de tarjetas inteligentes. Proporciona una capa adicional de seguridad y facilita operaciones criptográficas de forma segura entre el sistema y la tarjeta inteligente.

Dentro de tarjetas MiFare existen diferentes tipos de claves según la familia.

- Mifare Classic y MiFare Plus: Utilizan claves de 6 bytes. Existiendo la clave A y la clave B para cada sector, cada sector contiene un número determinado de bloques de datos.
- MiFare Ultralight: No es necesario ningún tipo de autenticación para acceder a los bloques de datos.
- MiFare Desfire: Utilizan claves de 16 bytes. Existiendo hasta 16 diferentes claves para cada aplicación, cada aplicación contiene un número determinado de ficheros de datos.

En la Tabla 1 se muestran las diferentes características de los tipos de tarjetas.

	Memoria	Organización	Tipo memoria	UID	Baudrate	Claves de acceso	Seguridad	Generador número aleatorio
Mifare Classic 1K	1.024 bytes	16 sectores con 64 bytes	EEPROM 100.000 ciclos	4 bytes (NUID) 7 bytes (UID)	106 kbits/segundo	2 claves por sector	Crypto1	Sí
Mifare Classic 4K	4.096 bytes	32 sectores con 64 bytes y 8 sectores con 256 bytes	EEPROM 100.000 ciclos	4 bytes (NUID) 7 bytes (UID)	106 kbits/segundo	2 claves por sector	Crypto1	Sí
Mifare Ultralight	64 byte (32 bits OTP)	16 páginas con 4 bytes	EEPROM 10.000 ciclos	7 bytes (UID)	106 kbits/segundo	No	No	No
Mifare Ultralight C	192 byte (32 bits OTP)	48 páginas con 4 bytes	EEPROM 10.000 ciclos	7 bytes (UID)	106 kbits/segundo	1 clave	Autentificación DES-3DES	Sí
Mifare Plus S 2K	2.048 bytes	32 sectores con 64 bytes	EEPROM 200.000 ciclos	4 bytes (NUID) 7 bytes (UID)	Desde 106 a 848 kbits/segundo	2 claves por sector	Crypto1 o AES	Sí
Mifare Plus S 4K	4.096 bytes	32 sectores con 64 bytes y 8 sectores con 256 bytes	EEPROM 200.000 ciclos	4 bytes (NUID) 7 bytes (UID)	Desde 106 a 848 kbits/segundo	2 claves por sector	Crypto1 o AES	Sí
Mifare Plus X 2K	2.048 bytes	32 sectores con 64 bytes	EEPROM 200.000 ciclos	4 bytes (NUID) 7 bytes (UID)	Desde 106 a 848 kbits/segundo	2 claves por sector	Crypto1 o AES	Sí
Mifare Plus X 4K	4.096 bytes	32 sectores con 64 bytes y 8 sectores con 256 bytes	EEPROM 200.000 ciclos	4 bytes (NUID) 7 bytes (UID)	Desde 106 a 848 kbits/segundo	2 claves por sector	Crypto1 o AES	Sí
Mifare DESFire EV1 2K	2.048 bytes	Sistema de archivos	EEPROM 500.000 ciclos	7 bytes (UID)	Desde 106 a 848 kbits/segundo	14 claves por aplicación	DES, 3DES, AES 128	Sí
Mifare DESFire EV1 4K	4.096 bytes	Sistema de archivos	EEPROM 500.000 ciclos	7 bytes (UID)	Desde 106 a 848 kbits/segundo	14 claves por aplicación	DES, 3DES, AES 128	Sí
Mifare DESFire EV1 8K	8.192 bytes	Sistema de archivos	EEPROM 500.000 ciclos	7 bytes (UID)	Desde 106 a 848 kbits/segundo	14 claves por aplicación	DES, 3DES, AES 128	Sí

Tabla 1 Tipos de tarjetas MiFare

Existen diferentes tipos de software disponibles para acceder al contenido de las tarjetas MiFare. Estos incluyen el software proporcionado por los fabricantes de lectores de tarjetas MiFare, las librerías y SDK de desarrollo ofrecidos por fabricantes y desarrolladores de software, y el software propietario desarrollado por terceros. Algunos ejemplos populares de software son las aplicaciones móviles "Mifare Classic Tool" y "NFC Tools". Sin embargo, todas estas opciones permiten acceder y modificar los datos en bruto de la tarjeta sin un formato amigable.

Sección 1.2. Solución propuesta

Para abordar la limitación anteriormente descrita, se propone la parametrización de la relación de los datos de diferentes tarjetas mediante un lenguaje específico de dominio. Esto implicaría establecer una correspondencia entre el contenido de las tarjetas y una colección de tipos de datos estándar y definidos. Con el uso de este lenguaje, se podría codificar esta relación y facilitar un formato más comprensible para trabajar con los datos de las tarjetas MiFare.

El diseño de cualquier lenguaje específico de dominio (DSL) pretende mejorar la eficiencia, la productividad y la flexibilidad del desarrollo de software al adaptarse a las diferentes necesidades y características actuales y futuras. Del mismo modo facilita la comprensión y el mantenimiento de la solución software.

Sección 1.3. Resultados obtenidos

Se ha empleado el DSL en cinco escenarios para mapear la información de la memoria de tarjetas de transporte. Cada escenario abarca un caso en el que se utiliza el DSL para decodificar y representar la información de distintos tipos de tarjetas. Inicialmente, se muestra el contenido de manera cruda, sin conocimiento del esquema de memoria, seguido por la representación del contenido después de ser mapeado mediante el esquema codificado en el DSL.

La sección 4.1 aborda el contenido de una tarjeta MiFare Ultralight. Las secciones 4.2, 4.3 y 4.4 exponen los datos de diferentes tarjetas MiFare Classic. Y en la sección 4.5 se presenta el contenido de una tarjeta MiFare DESFire.

Sección 1.4. Software desarrollado

El software desarrollado permite la edición de los datos de tarjetas MiFare, mediante la parametrización en forma de DSL. El aplicativo construye una interfaz gráfica, de acuerdo a los parámetros previamente establecidos, donde el usuario puede visualizar y/o manipular los diferentes datos de las tarjetas de manera intuitiva y sencilla, sin necesidad de tener conocimientos de programación. En la Figura 2 se muestra un boceto de la funcionalidad.

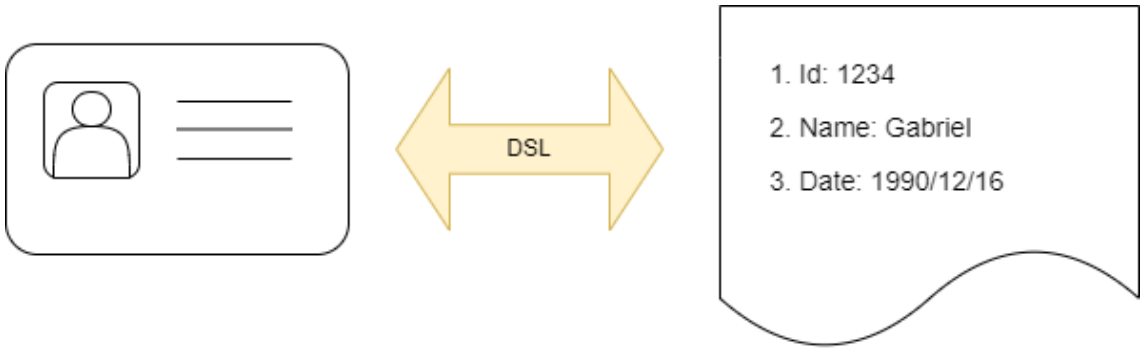

Figura 2 Transformación de datos.

Asimismo, el software también permite volcar el contenido de las tarjetas en un fichero del mismo modo que permite realizar la operación inversa, es decir, grabar las tarjetas con los datos de dichos ficheros. Mediante esta funcionalidad se puede hacer respaldo de los datos o incluso se podría realizar un clonado de datos entre diferentes tarjetas.

La solución software se ha almacenado en un repositorio público de GitHub al cual se puede acceder mediante el siguiente enlace.

Repositorio público: https://github.com/gguerra10/Cardamatic-public

Consta de varias librerías, entre las que cabe destacar las siguientes:

- GGuerra.Cardamatic.WinForm: contiene toda la lógica relacionada con la interfaz gráfica.
- GGuerra.Cardamatic.CardReader: varias librerías que definen la interfaz e implementación de un lector de tarjetas sin contacto usando el protocolo PC/SC.
- GGuerra.Cardamatic.Encoding: consta de varias librerías que se encargan de la codificación/decodificación de los datos de las tarjetas.
- GGuerra.Cardamatic.App: Aplicación ejecutable que hace uso de todas las librerías anteriores mediante la inyección de dependencias.

Los puntos clave de la arquitectura y los detalles se extienden en las secciones 3.2 y 3.3 de este documento.

Capítulo 2. Estado del arte

El desarrollo de software relativo a tarjetas inteligentes ha sido objeto de investigación y desarrollo en diversos estudios previos. En la literatura académica y técnica, se han abordado diferentes enfoques y técnicas para trabajar con estos dispositivos de almacenamiento de datos.

Investigadores de la Universidad de Augsburg publicaron varios artículos donde proponen un enfoque basado en desarrollo dirigido a modelo, el cual facilita el desarrollo de aplicaciones críticas de seguridad basadas en protocolos criptográficos [1] [2]. Su enfoque integra de manera fluida la generación de código y los métodos formales. Además, investigadores de la Universidad de Shiraz también proponen un marco de trabajo similar basado igualmente en la arquitectura dirigida a modelos [3]. Ambos estudios parten de un modelo UML independiente a la plataforma de desarrollo logran generar código ejecutable en Java Card. No obstante, el producto de ambos trabajos es el software que reside en las tarjetas sin contacto, dejando de lado la manipulación de los datos de las mismas.

Sin embargo, Xu JunWu y Xie Fang han señalado que la comunicación con las tarjetas es una parte crucial pero desafiante, ya que cada tarjeta se comunica de manera diferente [4]. Su trabajo se centra en proporcionar una comprensión básica de la comunicación PC/SC con un lector y una tarjeta, dejando la comunicación específica con las tarjetas como un aspecto abierto a la variabilidad.

En el contexto del transporte público, investigadores del departamento de electrónica del Laboratorio Tecnológico de Uruguay (LATU) presentaron un trabajo en el cual presentan los procedimientos y el equipo de laboratorio desarrollados para evaluar las características de las tarjetas RFID de transporte [5]. En dicho trabajo proporcionaron una herramienta de software a la entidad gestora del servicio de transporte público para realizar sus propias pruebas sobre sus tarjetas inteligentes.

En el ámbito educativo, un grupo del departamento de ingeniería eléctrica y tecnología de la información de la Universidad de Gadjah Mada (Indonesia) presentan un prototipo de sistema de asistencia en línea para instituciones educativas en Indonesia [6]. El sistema utiliza tarjetas inteligentes Mifare 1K como identificación de estudiantes y una Raspberry Pi como lector de tarjetas y dispositivo de conexión. Introducen dos aplicaciones de software, una para la gestión de tarjetas y otra para la lectura de asistencia.

Además, investigadores de la Universidad de Mapua publicaron un artículo científico donde detallan los riesgos de la tecnología sin contacto [7]. Realizan un ataque sobre una tarjeta MiFare Classic utilizando la aplicación móvil MiFare Classic Tool.

Si bien estos estudios han abordado la creación y/o manipulación de datos en tarjetas MiFare, aún existen algunas limitaciones en términos de flexibilidad, usabilidad y aplicabilidad en entornos diversos. Actualmente existen varios tipos de software disponibles para acceder al contenido de las tarjetas MiFare. Algunos de ellos son:

- Software proporcionado por el fabricante del lector de tarjetas: Muchos fabricantes de lectores de tarjetas MiFare ofrecen su propio software de lectura y escritura que es compatible con sus dispositivos.
- Librerías y SDK de desarrollo: Algunos fabricantes y desarrolladores de software proporcionan librerías y kits de desarrollo de software (SDK) que permiten a los

desarrolladores integrar la funcionalidad de lectura y escritura de tarjetas MiFare en sus propias aplicaciones personalizadas.

- Software propietario: software desarrollado por terceros que es compatible con lectores de tarjetas MiFare. Estos programas suelen ofrecer una interfaz de usuario más amigable y funcionalidades adicionales. Algunos ejemplos populares pueden ser "Mifare Classic Tool" y "NFC Tools" que permiten la lectura y escritura de tarjetas MiFare.

Estas opciones anteriores permitirían obtener y modificar los datos en crudo de la tarjeta a lo sumo, sin ningún tipo de formato amigable. Para solventar esta carencia se propone parametrizar la relación de los datos fuentes como sería el contenido de diferentes tarjetas a diferentes hacia una colección de tipos de datos (estándar y definidos) y viceversa. Para tal fin se propone la invención de un lenguaje especifico de dominio que permita codificar tales correspondencias.

	Software lector	SDK desarrollo	MiFare Classic Tool	DSL
Ajeno a programación	SÍ	NO	SÍ	SÍ
Compatibilidad diferentes lectores	NO	-	NO*	NO**
Confidencialidad claves	NO	-	NO	SÍ
Compatible con MiFare Classic	SÍ	SÍ	SÍ	SÍ
Compatible con MiFare DESFire	SÍ	SÍ	NO	SÍ
Formato de los datos personalizado	NO	-	NO	SÍ
Adaptabilidad	NO	-	NO	SÍ

Tabla 2 Tabla comparativa de aproximaciones

A la vista de la Tabla 2 el software que podría incluir el fabricante del lector es ajeno a la programación sin embargo podría no ser compatible con otros lectores, lo que limita su alcance. Por otro lado, los SDK de desarrollo ofrecen flexibilidad, pero se necesitan conocimientos de programación para explotarlos. MiFare Classic Tool es un software popular que permite la lectura y escritura de tarjetas MiFare Classic. Sin embargo, no es compatible con MiFare DESFire y no ofrece la posibilidad de decodificar el mapa de la tarjeta más allá del crudo hexadecimal. En contraste, la creación de un lenguaje especifico de dominio permite la creación de un formato de datos personalizado y ofrece confidencialidad de claves. Además, el DSL puede ser ampliado para un futuro ser compatible con nuevas tecnologías.

Capítulo 3. Exposición

En el presente apartado de exposición se presentarán en profundidad el enfoque y los componentes clave de la solución, del mismo modo, se explorarán los detalles técnicos y funcionales de la solución software objetivo en los siguientes subapartados:

- Lectura y escritura de las tarjetas sin contacto.
- Definición DSL.
- Interpretación DSL.
- Decodificadores DSL.

La solución aportada sigue el patrón de diseño de inyección de dependencias para abstraer las dependencias entre los diferentes componentes.

Las ventajas de utilizar la inyección de dependencias son:

- **Desacoplamiento**: Permite separar la creación y gestión de objetos de sus dependencias, lo que resulta en un código más modular y fácil de mantener. Los objetos no necesitan conocer los detalles de cómo se crean o se obtienen sus dependencias.
- **Flexibilidad y reutilización**: Al ser las dependencias proporcionadas externamente, es más fácil cambiar las implementaciones de las dependencias o reutilizar componentes en diferentes contextos sin modificar el código del objeto dependiente.
- **Testabilidad**: Facilita la realización de pruebas unitarias, ya que se pueden proporcionar dependencias simuladas u objetos 'mock' durante las pruebas. Esto permite aislar el objeto bajo prueba y probarlo de manera independiente.
- **Modularidad**: La inyección de dependencias fomenta el diseño de componentes más pequeños y especializados, lo que facilita la comprensión, el mantenimiento y la evolución del código.

Sección 3.1. Antecedentes: lectura y escritura de las tarjetas sin contacto

La comunicación con las tarjetas inteligentes se realiza mediante el intercambio de mensajes **APDU** (Application Protocol Data Unit). Un APDU consiste en dos partes principales: el encabezado y los datos. El encabezado contiene información sobre el tipo de comando que se envía a la tarjeta (comando de envío) o la respuesta de la tarjeta al comando (comando de respuesta). Los datos, si están presentes, contienen información adicional que se transmite junto con el comando o la respuesta.

Existen dos tipos de APDU:

1- Comando APDU (**ApduCommand**): Es utilizado por el dispositivo lector para enviar comandos a la tarjeta inteligente. El comando APDU consta de los siguientes elementos:
   - CLA (Class): Especifica la clase del comando y proporciona información sobre el tipo de operación a realizar.
   - INS (Instruction): Indica la instrucción o acción específica que se va a realizar en la tarjeta.
   - P1 y P2 (Parameter 1 y Parameter 2): Son parámetros opcionales utilizados para proporcionar información adicional al comando.

- Lc (Length of Command Data): Indica la longitud de los datos que se envían junto con el comando.
- Command Data: Son los datos que se envían junto con el comando, si es necesario.
- Le (Length Expected): Indica la longitud esperada de la respuesta de la tarjeta.

2- Respuesta APDU (**ApduResponse**): Es utilizado por la tarjeta inteligente para enviar una respuesta al dispositivo lector después de recibir un comando. La respuesta APDU consta de los siguientes elementos:
   - Data: Son los datos de respuesta devueltos por la tarjeta.
   - SW1 y SW2 (Status Word 1 y Status Word 2): Proporcionan información sobre el estado de la operación realizada por la tarjeta.

Para intercambiar los APDU y acceder al contenido de las tarjetas sin contacto se ha desarrollado un adaptador de lector de tarjetas. Las interfaces definidas para implementar el adaptador del hardware se presentan en la Figura 3.

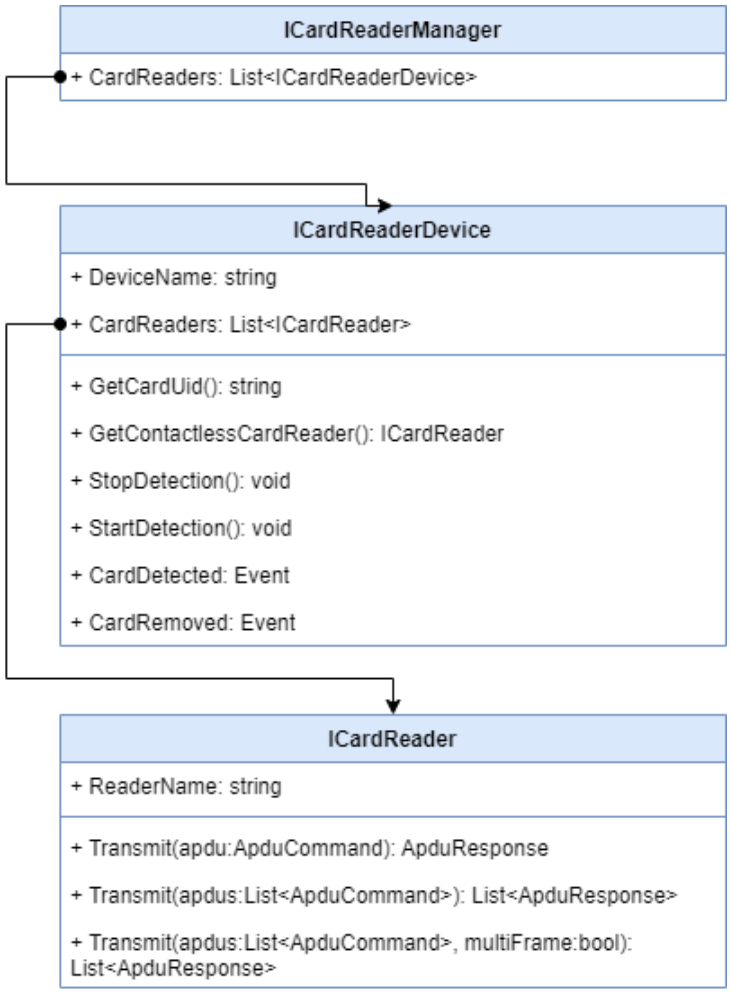

Figura 3 Interfaces del adaptador lector de tarjetas

La interfaz **ICardReaderManager** es un gestor de lectores de tarjetas cuyo cometido es listar los diferentes dispositivos conectados a la máquina.

La interfaz **ICardReaderDevice** representa un dispositivo lector de tarjetas que a su vez puede disponer de más de un lector, comúnmente un lector para las tarjetas sin contacto y uno o varios lectores para los módulos de acceso seguro. Dispone de los siguientes métodos:

- **GetContactlessCardReader**: devuelve la instancia del lector sin contacto del dispositivo.
- **GetCardUid**: envía un comando APDU al lector de tarjetas sin contacto del dispositivo para obtener el identificador único de la tarjeta si esta está presente.
- **StartDetection**: habilita la detección de tarjeta sin contacto hasta que se presente una.
- **StopDetection**: deshabilita la detección de tarjeta sin contacto.

Posee los eventos **CardDeteted** / **CardRemoved** que se lanzan cuando la detección de tarjeta sin contacto está habilitada.

Y en el nivel más bajo se encuentra la interfaz **ICardReader** representa un lector de tarjetas con el cual se pueden intercambiar los mensajes APDU.

La implementación del adaptador de lector se ha desarrollado apoyándose en la API “PC/SC” (Personal Computer/Smart Card) esta define la comunicación y la interacción entre la máquina y las tarjetas inteligentes. Esta API esta incorporada en los sistemas operativos Windows en la librería winscard. También existe una implementación libre, de código abierto, llamada “PC/SC Lite” para sistemas operativos Linux.

Un nivel por encima del adaptador también se han definido dos interfaces que servirán de abstracción para comunicarse con los diferentes tipos de tarjetas **IMifareClassicService** (para las tarjetas MiFare Classic y MiFare Ultralight) y **IDesfireService** (para las tarjetas MiFare Desfire).

La interfaz definida **IMifareClassicService** se muestra en la Figura 4.

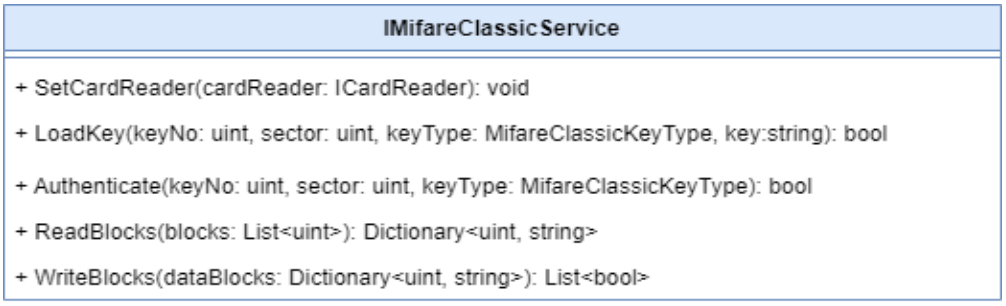

Figura 4 Interfaz del protocolo de comunicaciones con tarjetas MiFare Classic

- **SetCardReader**: Este método establece el lector de tarjetas tomando un objeto **ICardReader** como parámetro que se utilizará para enviar los comando.
- **LoadKey**: Este método carga una clave de autenticación en memoria. Recibe el número de clave, el sector de la tarjeta, el tipo de clave y la propia clave. Devuelve un valor booleano que indica si la operación de carga de la clave fue exitosa.
- **Authenticate**: Este método autentica con la clave previamente cargada en la memoria. Toma el número de clave, el sector de la tarjeta y el tipo de clave. Devuelve un valor booleano que indica si la autenticación fue exitosa.

- **ReadBlocks**: Este método lee bloques de una tarjeta MiFare. Recibe una colección de números de bloque (blocks) y devuelve un diccionario que mapea los números de bloque con los datos leídos de la tarjeta.
- **WriteBlocks**: Este método escribe bloques en una tarjeta MiFare. Recibe un diccionario que mapea los números de bloque con los datos que se deben escribir en la tarjeta. Devuelve una colección de valores booleanos que indican si la operación de escritura para cada bloque fue exitosa.

La implementación de esta interfaz se ha realizado de acuerdo a las instrucciones especificadas en la documentación del fabricante del lector utilizado (ACR1252U). A continuación, se detallan algunos de los comandos que se han utilizado.

1- Comando APDU para carga de clave en el lector:
   - CLA: 0xFF
   - INS: 0x82
   - P1, P2: P1 con valor 0x00 para cargar la clave en memoria, y P2 para indicar la dirección de memoria donde cargar la clave. Este lector en concreto reserva dos direcciones (0x00, 0x01)
   - Lc: 0x06
   - Command Data: 6 bytes con la clave.

2- Comando APDU de autenticación:
   - CLA: 0xFF
   - INS: 0x86
   - P1, P2: sin uso ambos con valores 0x00, 0x00
   - Lc: 0x05
   - Command Data: 5 bytes con la siguiente estructura:
     - Version: 0x01
     - Sin uso
     - Numero de bloque: bloque de memoria que se desea autenticar.
     - Tipo de clave: tipo de clave que se desea autenticar TipoA (0x60) o TipoB (0x61)
     - Numero de clave. Dirección de memoria donde se ha alojado la clave anteriormente.

3- Comando APDU de lectura de datos:
   - CLA: 0xFF
   - INS: 0xB0
   - P1, P2: P1 sin uso con valor 0x00, y P2 para indicar el bloque de memoria al que realizar la lectura.
   - Le: número de bytes que se desea leer. Su valor ha de ser múltiplo de 16 ya que es el tamaño de bloque.

4- Comando APDU de escritura de datos:
   - CLA: 0xFF
   - INS: 0xB0
   - P1, P2: P1 sin uso con valor 0x00, y P2 para indicar el bloque de memoria al que realizar la lectura.
   - Lc: número de bytes que se desea escribir. Solo se admiten múltiplos de 16 ya que es el tamaño de bloque.
   - Command Data: los datos que se desean escribir cuya longitud se ha indicado antes.

Por otro lado, la interfaz del protocolo **IDesfireService** tiene ciertas similitudes con la anterior, algunos de sus métodos son:

- **SetCardReader**: Este método establece el lector de tarjetas tomando un objeto **ICardReader** como parámetro que se utilizará para enviar los comando.
- **SetSamServcie**: Este método establece el servicio de modulo acceso seguro tomando como entrada una instancia de **ISamService**, este se utilizará para las operaciones criptográficas de diversificación de claves.
- **SelectApplication**: Este método selecciona una aplicación en la tarjeta Desfire. Recibe el identificador de la aplicación y devuelve un valor booleano que indica si la selección de la aplicación fue exitosa.
- **Authenticate**: Este método autentifica con claves fijas la aplicación anteriormente seleccionada. Toma como argumentos el número de clave, la propia clave y el tipo de autentificación. Devuelve un valor booleano que indica si la autentificación fue exitosa.
- **Authenticate**: Este método autentifica con claves diversificadas por el UID de la tarjeta. Toma el número de clave, la propia clave, el tipo de autentificación y el UID de la tarjeta. Devuelve un valor booleano que indica si la autentificación fue exitosa.
- **ReadFile**: Este método lee el contenido de un archivo de una tarjeta Desfire. Recibe el identificador de archivo, el tipo de archivo, el desplazamiento y la longitud de los datos a leer. Devuelve una cadena que representa el contenido leído del archivo.
- **WriteFile**: Este método escribe contenido en un archivo de una tarjeta Desfire. Recibe los datos a escribir, el identificador de archivo, el tipo de archivo, el desplazamiento y la longitud de los datos a escribir. Devuelve un valor booleano que indica si la operación de escritura fue exitosa.

La implementación se ha realizado siguiendo documentación confidencial de la empresa propietaria (NXP); de esta manera no se introduce de forma explícita detalles de este protocolo en el presente documento.

La interfaz **ISamService** posee un único método **GetKey** el cual retorna la clave diversificada tomando como parámetros una clave fija y el UID de la tarjeta. Aunque se ha desarrollado una implementación que busca remplazar el módulo SAM mediante software los detalles internos y las operaciones criptográficas no se introducirán en esta memoria.

Ambos protocolos se pueden dividir en dos fases, una primera fase de autenticación y otra segunda de lectura y/o escritura. Para tener éxito en la autenticación contra la tarjeta inteligente es necesario disponer de las claves correctas para cada tipo de tarjeta.

Para parametrizar los diferentes tipos de claves y tipos de autenticación se define una clase que mediante serialización soporte diferentes configuraciones de claves y puedan ser cargados en tiempo de ejecución de la aplicación. Los diferentes juegos de claves se encontrarán en ficheros encriptados de cara a garantizar la confidencialidad y la integridad de esta información sensible. Para poner en común ambos tipos de claves en las tarjetas MiFare Classic los sectores se representarán como direcciones de memoria, mientras que en las tarjetas MiFare Desfire se hará lo mismo con las aplicaciones.

A continuación, se ilustran sendos ejemplos de parametrización de claves en Figura 5 y Figura 6.

```
{
  "Description": "Mifare keys sample",
  "MifareClassicKeys": [
    {
      "Key": "000000000000",
      "KeyNo": "0",
      "Address": "0",
      "KeyType": "TypeA"
    },
    {
      "Key": "FFFFFFFFFFFF",
      "KeyNo": "0",
      "Address": "0",
      "KeyType": "TypeB"
    },
    {
      "Key": "A0A1A2A3A4A5",
      "KeyNo": "1",
      "Address": "0",
      "KeyType": "TypeA"
    },
    {
      "Key": "B0B1B2B3B4B5",
      "KeyNo": "1",
      "Address": "0",
      "KeyType": "TypeB"
    }
  ]
}
```

Figura 5 Ejemplo de claves MiFare Classic

```
{
  "Description": "Desfire keys sample",
  "DesfireKeys": [
    {
      "Key": "00000000000000000000000000000000",
      "KeyNo": "0",
      "Address": "000001",
      "ApplicationCrypto": "2"
    },
    {
      "Key": "0A0B0C0D0E0F1A1B1C1D1E1F2A2B2C2D",
      "KeyNo": "0",
      "Address": "010001",
      "ApplicationCrypto": "2"
    }
  ],
  "SamService": "SamSoftware"
}
```

Figura 6 Ejemplo de claves MiFare Desfire

La aplicación ofrecerá la capacidad de listar juegos de claves previamente configurados en una carpeta correspondiente para que el usuario pueda seleccionar las claves convenientes para cada tipo de tarjeta. Una vez que el usuario haya seleccionado un juego de claves, la aplicación realizará el proceso de desencriptado del fichero y cargará las claves de manera segura en la memoria. Estas claves estarán listas para ser utilizadas en las operaciones de autenticación y acceso a las tarjetas.

Una vez superada la fase de autenticación con la tarjeta, el lector ya puede leer y/o escribir datos en los bloques de memoria de la tarjeta. Por ejemplo, se pueden almacenar información como identificaciones, saldos, claves de acceso, entre otros.

Un mapa de memoria de una tarjeta sin contacto se puede representar visualmente como un diccionario cuyas claves serían los bloques (o direcciones) de memoria de la tarjeta y los valores el contenido mismo. A continuación, se muestra un ejemplo de una tarjeta MiFare Classic 1K en la Figura 7.

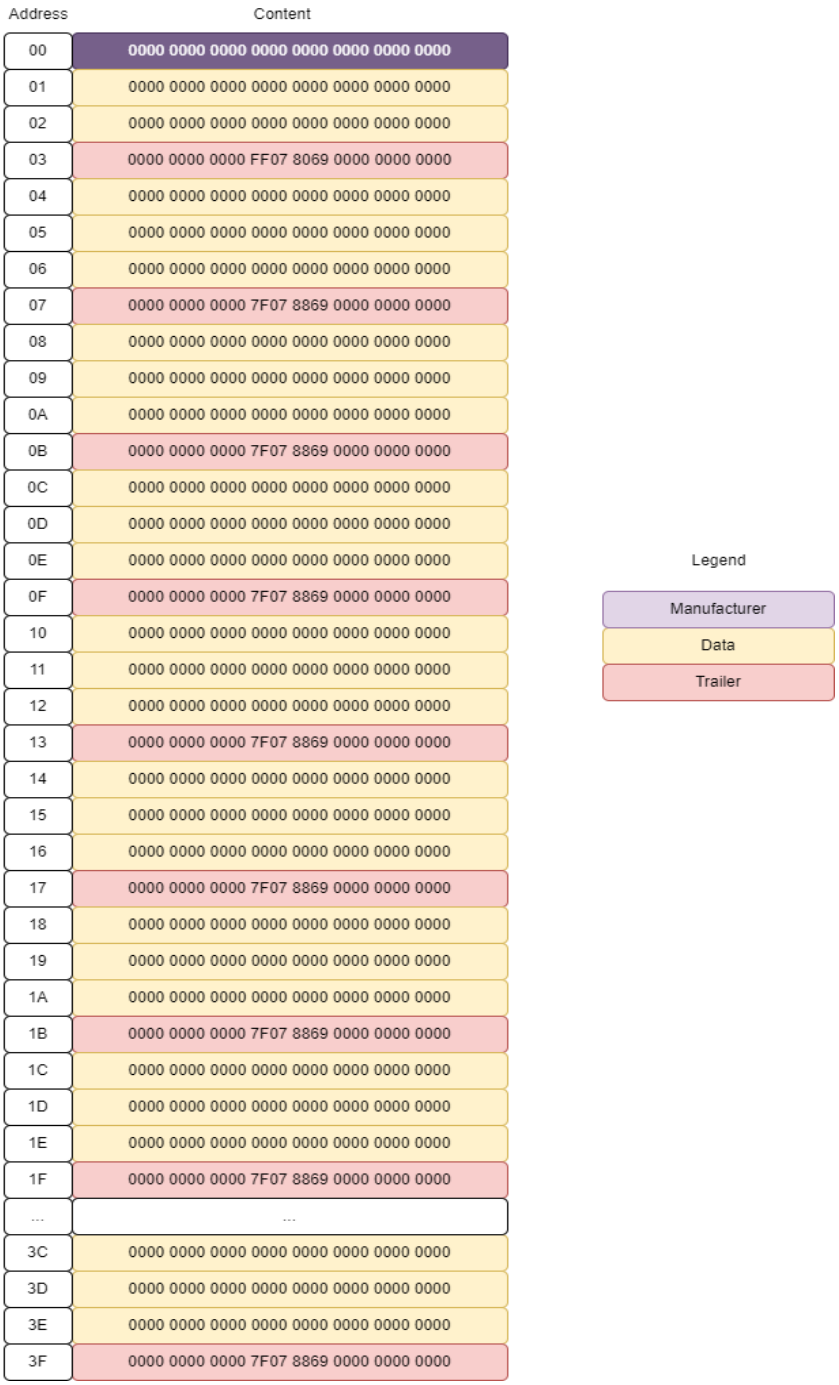

Figura 7 Mapa de memoria tarjeta MiFare Classic 1K

Sección 3.2. Definición del DSL

El propósito del DSL es proporcionar significado al contenido crudo de los mapas de memoria al transformar el contenido binario en una interfaz gráfica compuesta por pestañas, columnas y propiedades. Esta interfaz permitirá al usuario final visualizar y editar el contenido del mapa de memoria de manera más intuitiva y accesible. El DSL utiliza la codificación JSON y será la su interpretación la responsable de generar la interfaz gráfica. De esta manera cada esquema dispondrá de una descripción y una lista de pestañas, cada pestaña a su vez tendrá una descripción y una lista de columnas, cada columna será una lista propiedades y por último cada propiedad, que será la entidad encargada de realizar la transformación de formato binario al formato destino. El diagrama de clases se presenta en la siguiente Figura 8.

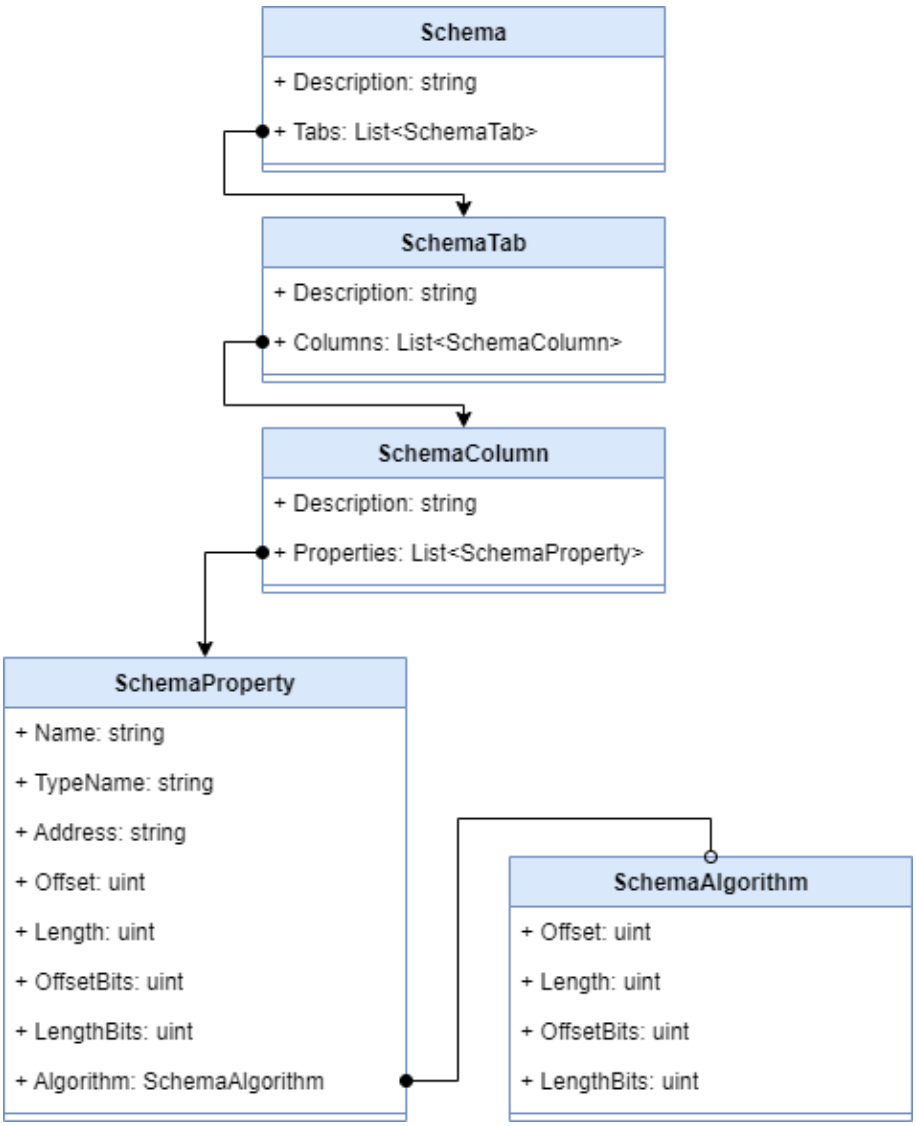

Figura 8 Diagrama de clases DSL

Cada una de las propiedades tendrá los siguientes campos obligatorios:

- **Name**: que servirá de identificador del dato en cuestión.
- **Address**: dirección del mapa de memoria de la tarjeta donde se aloja el dato.
- **Offset**: posición del dato en bytes dentro de la dirección de memoria.
- **Length**: longitud del dato en bytes dentro de la dirección de memoria.
- **TypeName**: será el nombre completo de una clase registrada que contendrá la lógica de representación, codificación y decodificación del contenido binario de la porción de mapa de memoria.

También hay cabida para los siguientes campos opcionales:

- **OffsetBits**: posición del dato en bits dentro de la dirección de memoria. Para ajustar aquellos datos que no estén alineados a nivel byte.
- **LengthBits**: longitud del dato en bits dentro de la dirección de memoria. Para ajustar aquellos datos cuya longitud no sean bytes enteros.

Por último, un campo opcional **Algorithm** el cual se utilizará para aquellos datos de la tarjeta que no sean editables por el usuario, sino que su cálculo se hace mediante un algoritmo utilizando datos de la propia tarjeta, como por ejemplo una suma de control. El campo **Algorithm** a su vez dispone de los campos **Offset** y **Length** (y los opcionales **OffsetBits** y **LengthBits**) para capturar la parte de memoria con la que realizar el cálculo.

En la próxima Figura 9, se muestra un escenario donde se han definido 4 propiedades utilizando el DSL y la transformación de datos para un mapa de memoria de entrada.

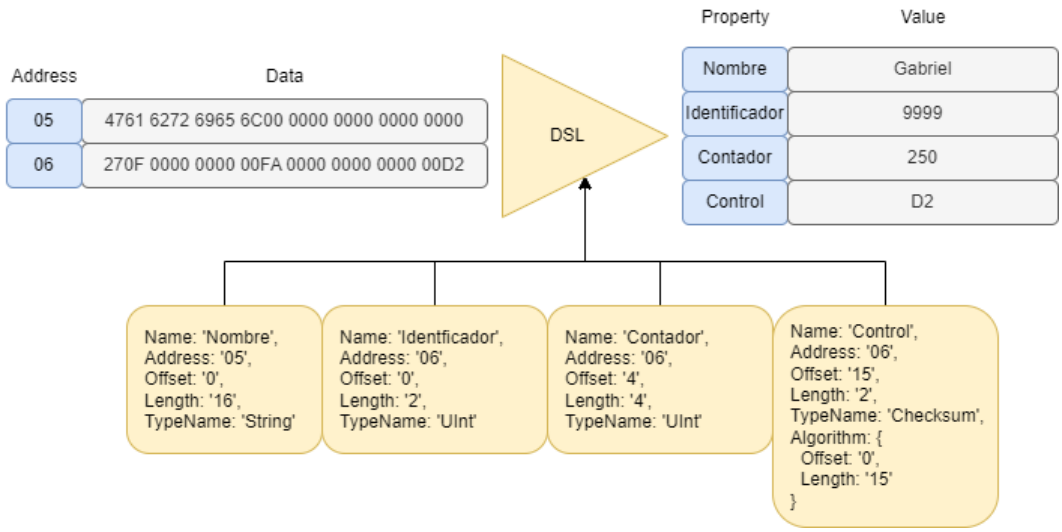

Figura 9 Detalle de transformación mediante el DSL.

En la siguiente Figura 10, se muestra el escenario inverso, tras una hipotética edición de los datos partiendo del escenario anterior, se realiza la transformación inversa obteniendo un nuevo mapa de memoria.

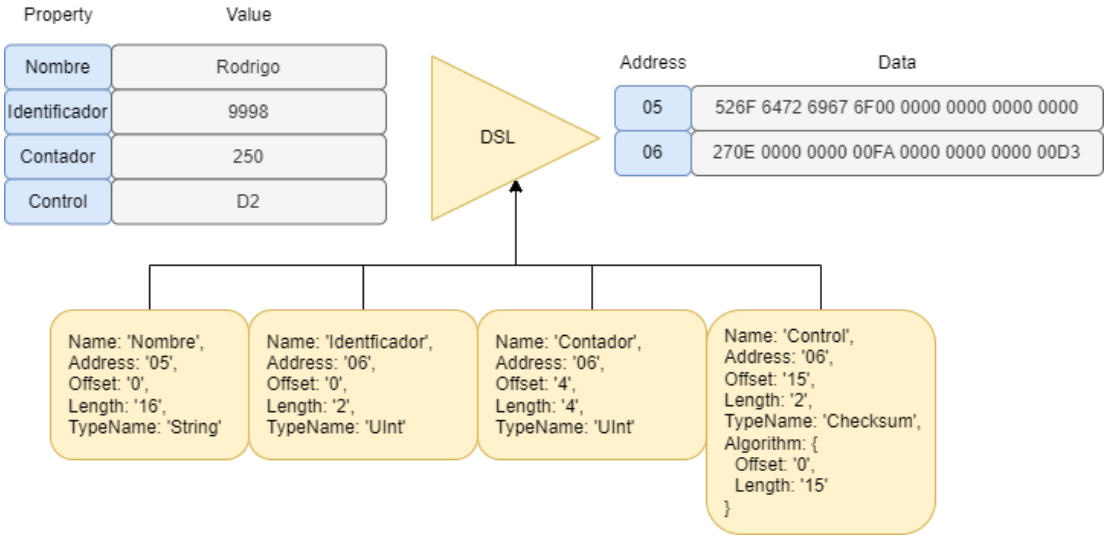

Figura 10 Detalle de transformación inversa mediante el DSL.

Sección 3.3. Interpretación del DSL

Para interpretar el DSL se ha creado un prototipo que consta de un formulario en el cual el usuario podrá visualizar el contenido del mapa de memoria de una tarjeta sin contacto, siempre que se conozca las claves de acceso para la tarjeta en concreto. El esquema general se muestra en la Figura 11.

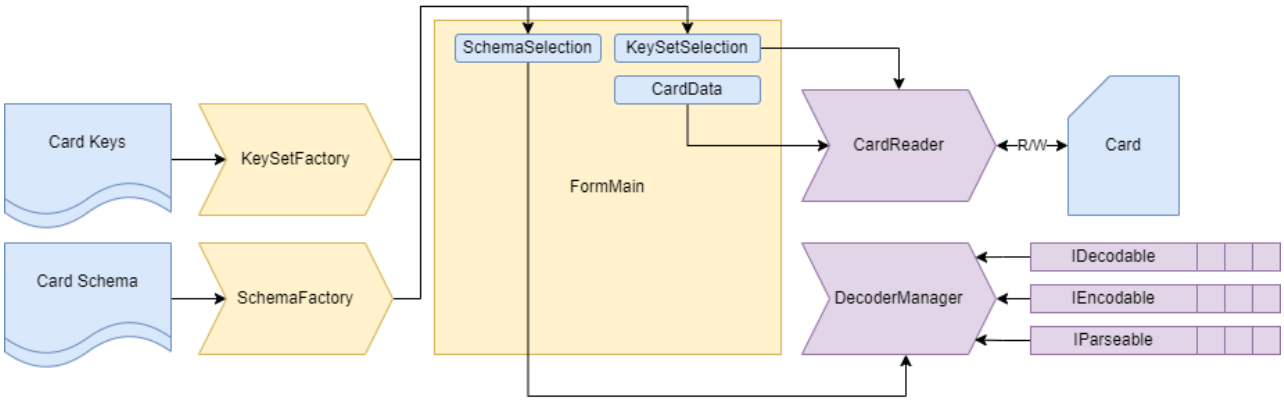

Figura 11 Detalle implementación intérprete DSL

La clase **SchemaFactory** es una implementación de una fábrica de esquemas. Esta clase se encarga de leer los archivos esquema, deserializarlos y validar su estructura para asegurarse de que cumplen con los requisitos esperados.

El método principal de la clase es **GetSchema**, el cual recibe la ruta del archivo de esquema como parámetro. En este método, se lee el contenido del archivo y se deserializa el JSON en un objeto **Schema**.

La clase tiene una dependencia de un objeto **IDecoderManager** que se utiliza para obtener los decodificadores necesarios durante el proceso de validación del esquema.

A continuación, se realiza una serie de comprobaciones en la estructura del esquema para asegurarse de que cumple con los requisitos. Se verifica que el esquema tenga pestañas, columnas y propiedades correctamente definidas. Además, se verifica que cada propiedad tenga una implementación de decodificación (con la interfaz **IDecodable**, cuyas características se detallan a continuación). Si alguna de estas comprobaciones falla, se lanza una excepción indicando el problema específico encontrado en el archivo de esquema. Finalmente, si el esquema pasa todas las validaciones, se devuelve el objeto **Schema** deserializado.

La clase **KeySetFactory** es una implementación de una fábrica de conjuntos de claves (**KeySet**). Esta clase se encarga de inicializar y proporcionar la lista de juegos de claves para acceder al contenido de las tarjetas.

El método principal de la clase es **Initialize** que se utiliza para inicializar los conjuntos de claves a partir de archivos en un directorio específico. Recibe como parámetros la ruta del directorio donde están los ficheros de claves y la extensión de los mismos. En este método, se itera sobre cada archivo en el directorio especificado con la extensión proporcionada. Para cada archivo, se llama al método privado **GetKeySet** que se encarga de leer los datos del archivo, descifrarlos, convertirlos a una cadena JSON y luego deserializarlos en un objeto **KeySet**. Si ocurre algún problema con este proceso, se lanza una excepción indicando que el archivo no es un conjunto de claves válido.

Por su parte el conjunto de clases del adaptador de lector de tarjetas (**CardReader**) proporcionan la lógica necesaria para el acceso al contenido de tarjetas físicas como se ha comentado en el subapartado anterior.

Asimismo, la clase **DecoderManager** se apoya en una serie de clases registradas (decodificadores) que se encargan de la transformación de los datos binarios al formato correspondiente que se ha mostrado en las figuras anteriores. Estas clases han de implementar las siguientes interfaces:

- **IDecodable**: La interfaz proporciona una forma de decodificar datos binarios en un formato correspondiente utilizando un delegado llamado **FieldDecoder**. Este delegado es responsable de la transformación de los datos binarios a un objeto de datos. Esto permite una decodificación personalizada y flexible de los datos binarios, utilizando los parámetros de entrada como el contenido binario, el desplazamiento, la longitud y la posición dentro de los datos binarios. El código fuente de la interfaz se presenta en la Figura 12.

```
    /// <summary>
    /// This delegate is responsible for the transformation of binary data to the corresponding
format.
    /// </summary>
    /// <param name="content">Binary data input.</param>
    /// <param name="pointer">Offset within the binary data in bytes.</param>
    /// <param name="positionBit">Offset within the binary data in bits.</param>
    /// <param name="lengthBytes">Length within the binary data in bytes.</param>
    /// <param name="lengthBits">Length within the binary data in bits.</param>
    /// <returns>Data object</returns>
    public delegate object FieldDecoder(byte[] content, uint pointer, uint positionBit, uint
lengthBytes, uint lengthBits);

    public interface IDecodable
    {
        /// <summary>
        /// Type declaration of the class.
        /// </summary>
        Type Type { get; }

        /// <summary>
        /// Get 'FieldDecoder' delegate.
        /// </summary>
        /// <returns></returns>
        FieldDecoder GetDecoder();

        /// <summary>
        /// Text length fixed
        /// </summary>
        bool TextLengthFixed { get; }
    }
```

Figura 12 Interfaz IDecodable

IEncodable: La interfaz proporciona una forma de codificar datos en un formato correspondiente en datos binarios utilizando un delegado llamado **FieldEncoder**. Este delegado es responsable de la transformación de los datos en un objeto de datos en un array de bytes. Esto permite una codificación flexible y personalizada de los datos en formato binario, utilizando los parámetros de entrada correspondientes, tales como los datos del objeto, el tamaño del arreglo de salida y la cantidad de bits utilizados en el tamaño del arreglo. El código fuente de la interfaz se presenta en la Figura 13.

```
    /// <summary>
    /// This delegate is responsible for the transformation of data in corresponding format to
binary data.
    /// </summary>
    /// <param name="data">Object data input.</param>
    /// <param name="dataSize">Output data array size in bytes.</param>
    /// <param name="dataSizeBits">Output data array size in bits.</param>
    /// <returns>Binary data array</returns>
    public delegate byte[] FieldEncoder(object data, int dataSize, uint dataSizeBits);

    public interface IEncodable
    {
        /// <summary>
        /// Type declaration of the class.
        /// </summary>
        Type Type { get; }

        /// <summary>
        /// Get 'FieldEncoder' delegate.
        /// </summary>
        /// <returns></returns>
        FieldEncoder GetEncoder();
    }
```

Figura 13 Interfaz IEncodable

- **IParseable**: La interfaz proporciona una forma de analizar y transformar texto de cadena en un formato correspondiente utilizando un delegado llamado **FieldParser**. Este delegado es responsable de la transformación del texto de cadena en un objeto de datos. Esto permite un análisis personalizado y flexible del texto que el usuario pueda modificar para convertirlo de nuevo en datos estructurados, utilizando el parámetro de entrada como el contenido de la cadena. El código fuente de la interfaz se presenta en la Figura 14.

```
    /// <summary>
    /// This delegate is responsible for the transformation of string text to the corresponding
format.
    /// </summary>
    /// <param name="content">String data input.</param>
    /// <returns>Data object</returns>
    public delegate object FieldParser(string content);

    public interface IParseable
    {
        /// <summary>
        /// Type declaration of the class.
        /// </summary>
        Type Type { get; }

        /// <summary>
        /// Get 'FieldParser' delegate.
        /// </summary>
        /// <returns></returns>
        FieldParser GetParser();
    }
```

Figura 14 Interfaz IParseable

Las tres interfaces tienen en común el campo **Type** el cual sirve de índice a la clase **DecoderManager** para localizar la clase correspondiente definida en las propiedades del DSL utilizando reflexión.

Sección 3.4. Decodificadores

Las clases que se han implementado cumpliendo las interfaces anteriormente detalladas son los decodificadores, estos mismos serán los tipos que soportará el DSL. Los decodificadores reciben una secuencia de bytes o bits y los convierten al tipo de dato objetivo y viceversa. Se han implementado los siguientes para los tipos de datos nativos:

- **BooleanDecodable** implementa las interfaces para decodificar, codificar y analizar objetos booleanos. Proporciona los delegados para decodificar bytes en booleanos, codificar booleanos en bytes y analizar cadenas de texto para valores de tipo **bool**.

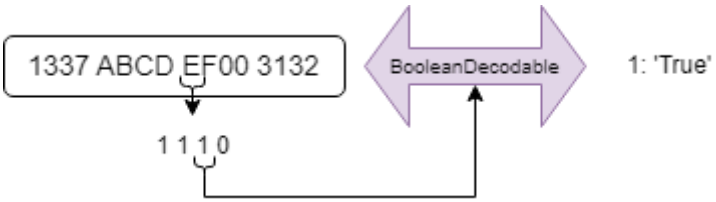

Figura 15 Decodificador de bool

- **CharDecodable** implementa las interfaces para decodificar, codificar y analizar objetos de tipo **char**. Proporciona los delegados para decodificar bytes en booleanos, codificar booleanos en bytes y analizar cadenas de texto para obtener caracteres.

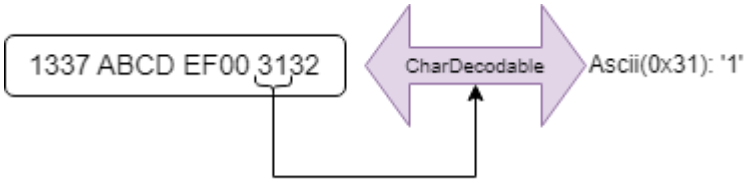

Figura 16 Decodificador de char

- **ByteDecodable** implementa interfaces para decodificar, codificar y analizar objetos de tipo byte. Proporciona los delegados pertinentes para el tipo **byte**.

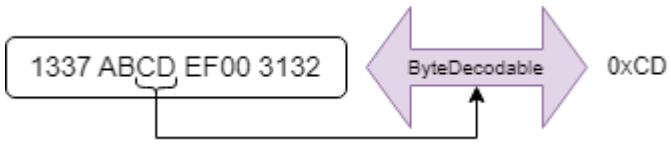

Figura 17 Decodificador de byte

- **UnsignedShortDecodable** implementa interfaces para decodificar, codificar y analizar enteros de 16 bits sin signo. Permite la conversión de bytes en formatos específicos, codificación de objetos byte en bytes y análisis de cadenas de texto para obtener valores de tipo **UInt16**.

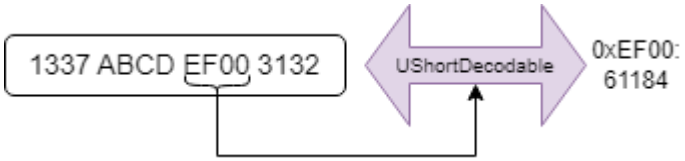

Figura 18 Decodificador de UInt16

- **ShortDecodable** implementa interfaces para decodificar, codificar y analizar enteros de 16 bits con signo. Proporciona los delegados pertinentes para el tipo de dato **Int16**.

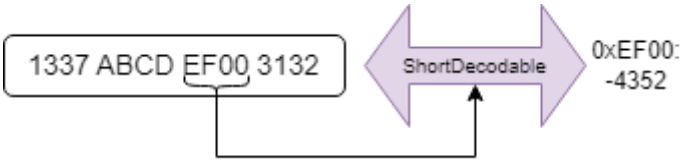

Figura 19 Decodificador de Int16

- **UnsignedIntegerDecodable** implementa interfaces para decodificar, codificar y analizar enteros de 32 bits sin signo. Proporciona los delegados pertinentes para el tipo de dato **UInt32**.

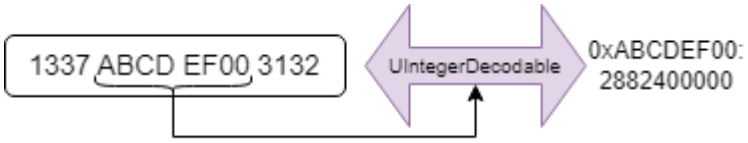

Figura 20 Decodificador de UInt32

- **IntegerDecodable** implementa interfaces para decodificar, codificar y analizar enteros de 32 bits con signo. Proporciona los delegados pertinentes para el tipo de dato **Int32**.

Figura 21 Decodificador de Int32

- **LongDecodable** implementa las interfaces para decodificar, codificar y analizar enteros de 64 bits con signo. Proporciona los delegados pertinentes para el tipo de dato **long**.

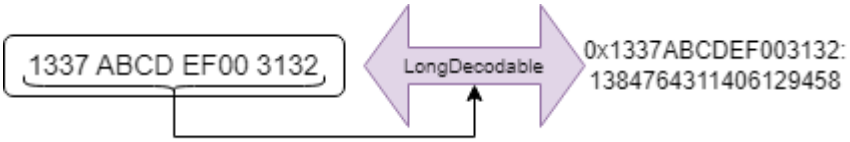

Figura 22 Decodificador de Int64

- **StringDecodable** implementa interfaces para decodificar, codificar y analizar cadenas de texto en la codificación ASCII. Proporciona los delegados pertinentes.

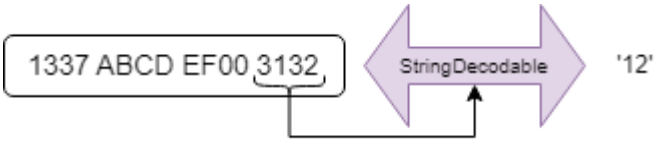

Figura 23 Decodificador de string

También se han implementado decodificadores para tipos creados:

- **HexStringDecodable** implementa interfaces para decodificar, codificar y analizar objetos de tipo **HexString**, que representa una cadena de texto en formato hexadecimal. Permite convertir bytes en una cadena de texto hexadecimal, codificar objetos **HexString** en bytes y analizar cadenas de texto en formato hexadecimal para obtener objetos **HexString**.

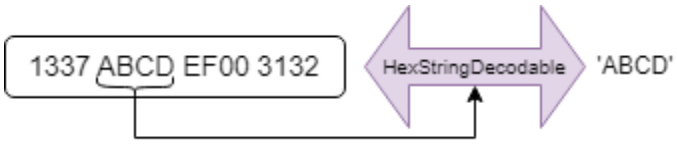

Figura 24 Decodificador de HexString

- **BinaryStringDecodable** implementa interfaces para decodificar, codificar y analizar objetos de tipo **BinaryString**, que representa una secuencia de bits en formato binario. Permite convertir bytes en una cadena de bits, codificar objetos **BinaryString** en bytes y analizar cadenas de texto en formato binario para obtener objetos **BinaryString**.

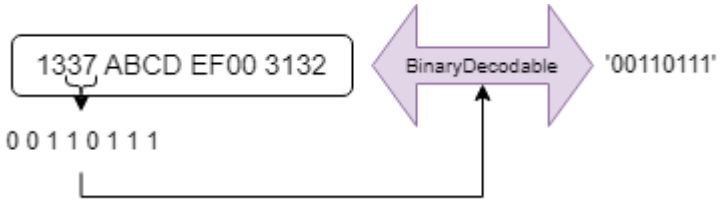

Figura 25 Decodificador de BinaryString

- **DateDecodable** es una implementación sellada que permite decodificar, codificar y analizar objetos de tipo "**Date**", que representa una fecha en dos bytes (7 bits para el año, 4 para el mes y 5 para el día). La clase se utiliza para convertir bytes en una fecha, codificar objetos **Date** en bytes y analizar cadenas de texto en formato de fecha para obtener objetos **Date**. Además, la clase maneja la referencia del año y aplica la lógica correspondiente para realizar las conversiones adecuadas.

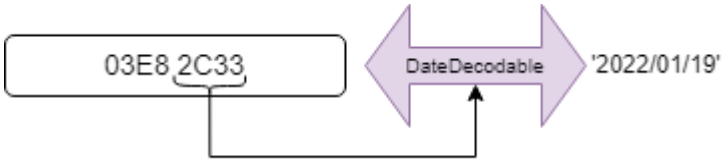

Figura 26 Decodificador de Date

- **TimeDecodable** es una implementación sellada que permite decodificar, codificar y analizar objetos de tipo **Time**, que representa una hora en dos bytes (6 bits para las horas, 6 para los minutos y 5 para los pares de segundos). La clase se utiliza para convertir bytes en una fecha, codificar objetos **Time** en bytes y analizar cadenas de texto en formato de hora para obtener objetos **Time**.

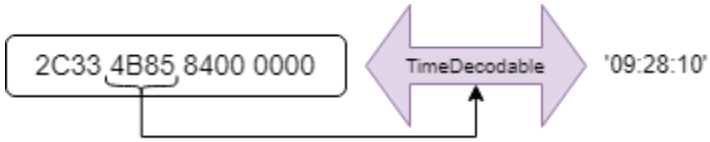

Figura 27 Decodificador de Time

- **DateTimeDecodable** es una implementación sellada que permite decodificar, codificar y analizar objetos de tipo **DateTime**, que representa una fecha con hora en cuatro bytes (6 bits para el año, 4 para el mes, 5 para el día, 5 para las horas, 6 para los minutos y 6 para los segundos). La clase se utiliza para calcular la fecha desde bytes, transformar objetos **DateTime** en bytes y analizar cadenas de texto en formato de fecha para obtener objetos **DateTime**. Además, la clase maneja la referencia del año y aplica la lógica correspondiente para realizar las conversiones adecuadas.

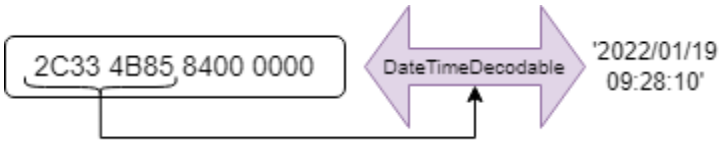

Figura 28 Decodificador de DateTime

- **BalanceDecodable** es una implementación que permite decodificar, codificar y analizar objetos de tipo Balance, que representa un saldo, los saldos en la mayoría de las tarjetas del tipo Mifare Classic se guarda con un formato especifico que consiste en: 4 bytes para el valor en Little Endian, 4 bytes para el mismo valor negado, 4 bytes con el valor de nuevo y 4 bytes finales para configurar permisos especiales.

Figura 29 Decodificador de Balance

- **ChecksumDecodable** implementa las interfaces que permiten decodificar y codificar objetos de tipo **Checksum**, que representa un valor de suma de comprobación. La clase se utiliza para convertir bytes en un objeto **Checksum**, y codificar objetos **Checksum** en bytes.

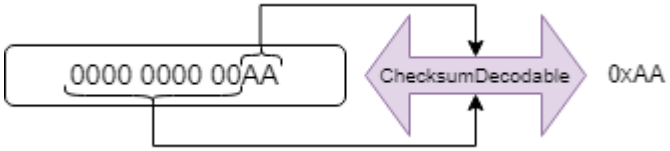

Figura 30 Decodificador de CheckSum

Al haber utilizado el patrón de inyección de dependencias añadir nuevos decodificadores al intérprete no sería muy costoso.

Capítulo 4. Validación empírica

En este capítulo se detallarán cinco escenarios donde se ha utilizado el DSL para mapear el contenido de la memoria de tarjetas de transporte a la interfaz gráfica del interprete. Para empezar, se mostrará el contenido en crudo sin tener conocimiento del esquema de memoria y luego el contenido una vez mapeado con el esquema codificado mediante el DSL.

Algunos de los esquemas mostrados a continuación pertenecen a tarjetas de transporte empleadas en sistemas en uso; de esta manera los mapas mostrados no son completos y tampoco se puede detallar explícitamente el contenido de las tarjetas.

Sección 4.1. Tarjeta MiFare Ultralight

En la siguiente Figura 31 se muestran las direcciones de memoria de una tarjeta MiFare Ultralight en formato hexadecimal.

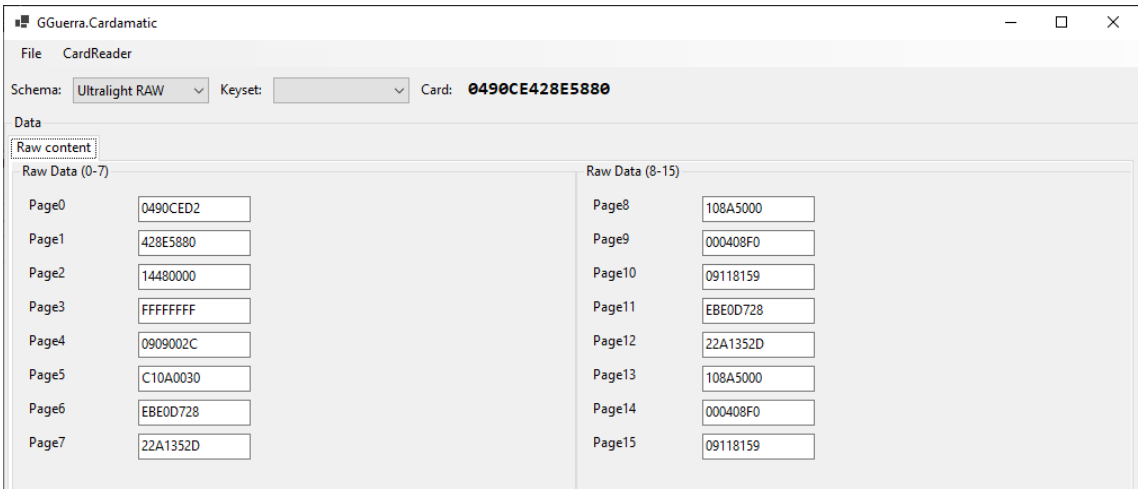

Figura 31 Contenido tarjeta MiFare Ultralight en crudo.

Parte del esquema de la tarjeta codificado en el DSL

```
{
  "Description": "Ultralight Px",
  "ContactlessTechnology": "Ultralight",
  "Tabs": [
    {
      "Description": "General Data",
      "Columns": [
        {
          "Description": "Card Data",
          "Properties": [
            {
              "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
              "name": "SerialNumber_H",
              "address": 0,
              "offset": 0,
              "length": 3
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
              "name": "SerialNumber_L",
              "address": 1,
```

```
            "offset": 0,
            "length": 4
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
            "name": "B0",
            "address": 0,
            "offset": 3,
            "length": 1
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
            "name": "B1",
            "address": 2,
            "offset": 0,
            "length": 1
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
            "name": "PRI",
            "address": 2,
            "offset": 1,
            "length": 1
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
            "name": "PRL",
            "address": 2,
            "offset": 2,
            "length": 2
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "OTP",
            "address": 3,
            "offset": 0,
            "length": 4
          }
        ]
      },
      {
        "Description": "Company Data",
        "Properties": [
          {
            "typeName": "System.UInt16",
            "name": "Version",
            "address": 4,
            "offset": 0,
            "length": 0,
            "offsetBits": 5,
            "lengthBits": 3
          },
          {
            "typeName": "System.UInt16",
            "name": "Company",
            "address": 4,
            "offset": 0,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 5
          },
          {
            "typeName": "System.UInt16",
            "name": "CardType",
            "address": 4,
            "offset": 0,
            "length": 0,
            "offsetBits": 5,
            "lengthBits": 3
          },
          {
            "typeName": "System.Boolean",
            "name": "Active",
            "address": 4,
```

```
            "offset": 0,
            "length": 0,
            "offsetBits": 4,
            "lengthBits": 1
          }
        ]
      }
    ]
  }
]
}
```

La salida del intérprete del DSL con el mapa de memoria formateado se presenta en la Figura 34.

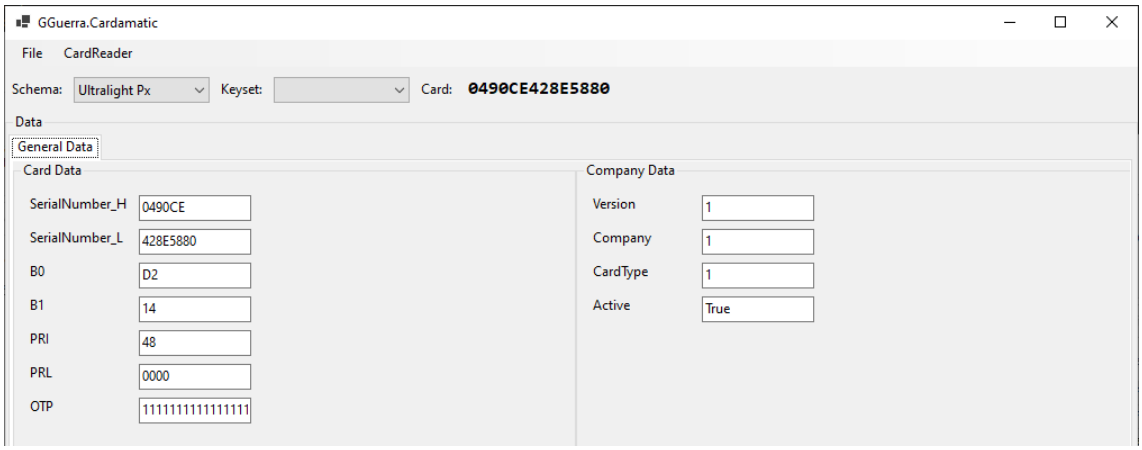

Figura 32 Contenido tarjeta MiFare Ultralight formateado.

Sección 4.2. Tarjeta MiFare Classic (A)

En las dos siguientes figuras (Figura 33 y Figura 34) se muestran las direcciones de memoria de una tarjeta MiFare Classic en formato hexadecimal.

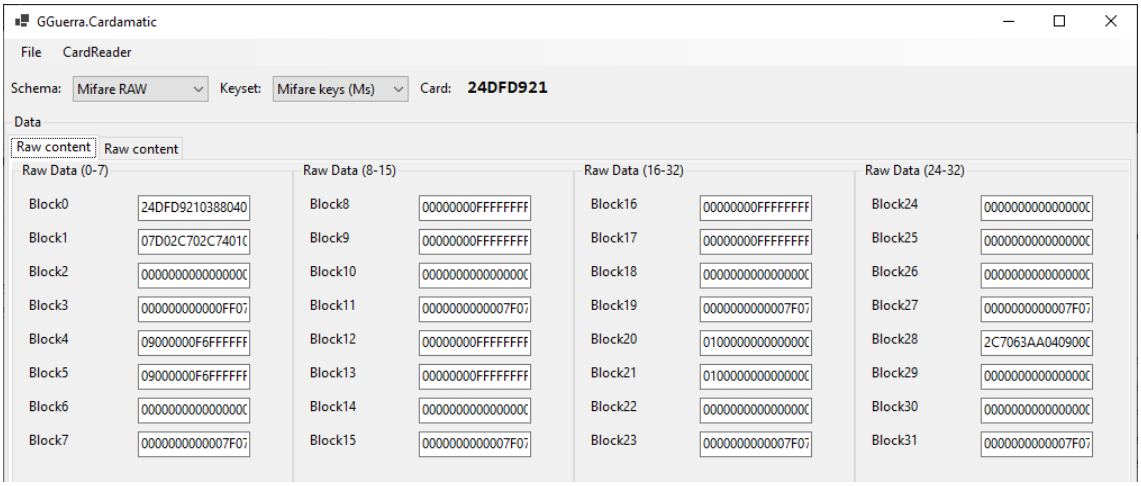

Figura 33 Contenido tarjeta MiFare Classic (A) en crudo (I)

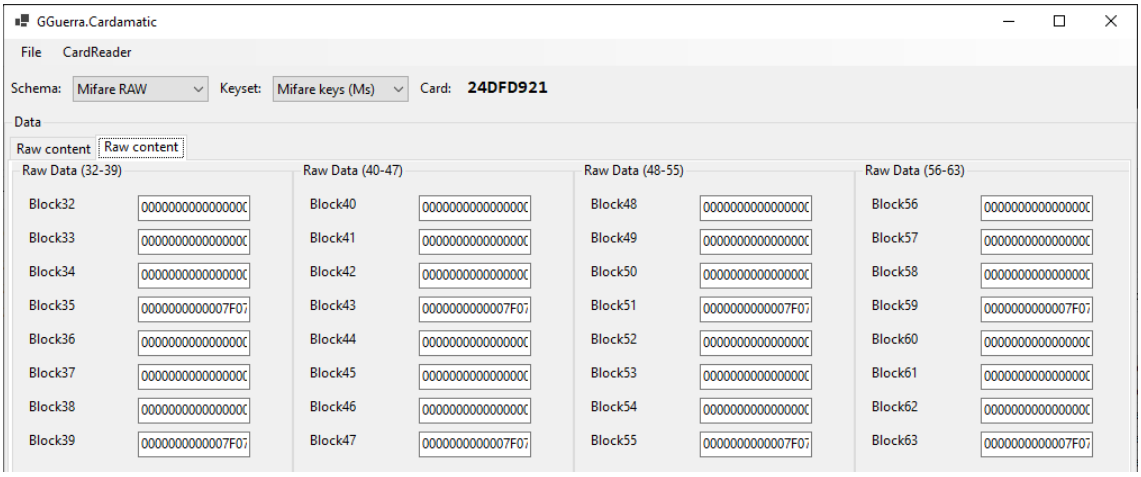

Figura 34 Contenido tarjeta MiFare Classic (A) en crudo (II)

Extracto del esquema del mapa de memoria codificado en el DSL.


```
{
  "description": "Mifare Ms",
  "ContactlessTechnology": "MifareClassic",
  "tabs": [
    {
      "description": "General Data",
      "columns": [
        {
          "description": "Company Data",
          "properties": [
            {
              "typeName": "GGuerra.Cardamatic.Encoding.Date.Date",
              "name": "CompanyDate",
              "address": 3,
              "offset": 14,
              "length": 2
            },
            {
              "typeName": "System.UInt16",
              "name": "CompanyCode",
              "address": 3,
              "offset": 12,
              "length": 2
            },
            {
              "typeName": "System.Byte",
              "name": "CompanyType",
              "address": 3,
              "offset": 10,
              "length": 1
            }
          ]
        }
      ]
    },
    {
      "description": "Title Data",
      "columns": [
        {
          "description": "Title 1",
          "properties": [
            {
              "typeName": "System.UInt16",
              "name": "TitleCode",
              "address": 1,
              "offset": 0,
              "length": 2
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.Date.Date",
              "name": "StartDate",
              "address": 1,
              "offset": 2,
              "length": 2
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.Date.Date",
              "name": "EndDate",
              "address": 1,
              "offset": 4,
              "length": 2
            },
            {
              "typeName": "System.Byte",
              "name": "TitleType",
              "address": 1,
              "offset": 6,
              "length": 1
            },
            {
```

```
            "typeName": "GGuerra.Cardamatic.Encoding.Balance.Balance",
            "name": "Balance",
            "address": 4,
            "offset": 0,
            "length": 16
          }
        ]
      },
      {
         // Intentionally omitted content
      }
    ]
  },
  {
    "description": "Transaction Data",
    "columns": [
      {
        "description": "Transaction[0]",
        "properties": [
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Date.Date",
            "name": "Date",
            "address": 28,
            "offset": 0,
            "length": 2
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Time.Time",
            "name": "Time",
            "address": 28,
            "offset": 2,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "Type",
            "address": 28,
            "offset": 4,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 6
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "Pos",
            "address": 28,
            "offset": 4,
            "length": 0,
            "offsetBits": 6,
            "lengthBits": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "P2",
            "address": 28,
            "offset": 5,
            "length": 1
          },
          {
            "typeName": "System.UInt16",
            "name": "P1",
            "address": 28,
            "offset": 6,
            "length": 1
          },
          {
            "typeName": "System.UInt16",
            "name": "Stop",
            "address": 28,
            "offset": 8,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
```

```
            "name": "Route",
            "address": 28,
            "offset": 10,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "Line",
            "address": 28,
            "offset": 12,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "P3",
            "address": 28,
            "offset": 14,
            "length": 2
          }
        ]
      },
      {
         // Intentionally omitted content
      }
    ]
  }
 ]
}
```

El resultado del esquema anterior mostrando el contenido de la tarjeta debidamente formateado se presenta en las próximas figuras (Figura 35, Figura 36 y Figura 37).

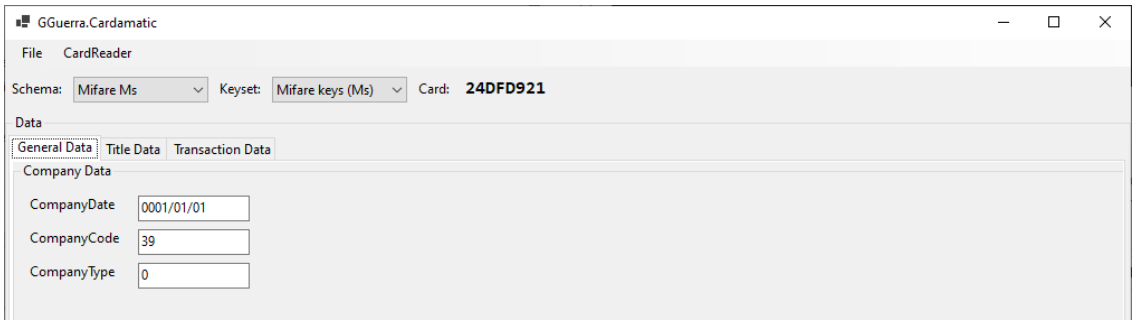

Figura 35 Contenido tarjeta MiFare Classic (A) formateado (I)

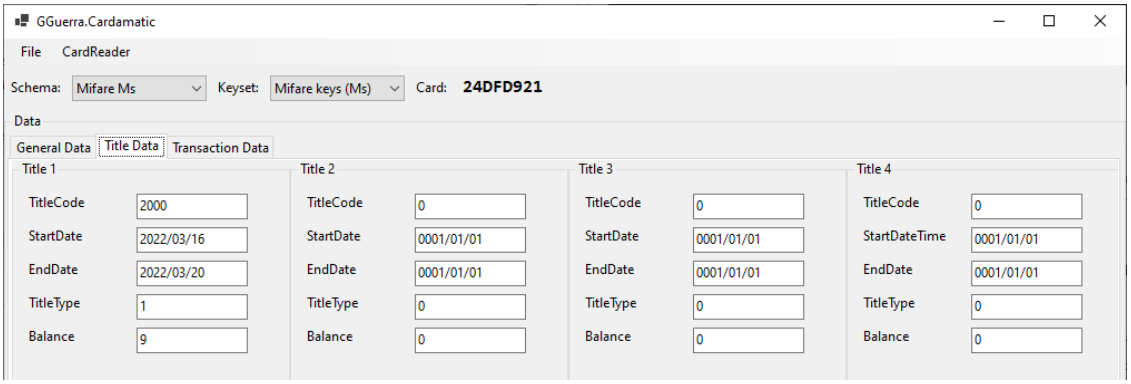

Figura 36 Contenido tarjeta MiFare Classic (A) formateado (II)

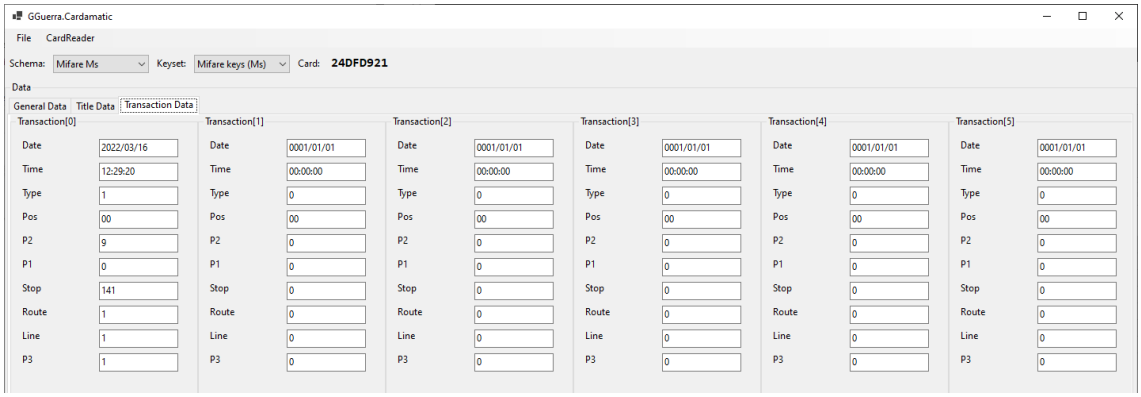

Figura 37 Contenido tarjeta MiFare Classic (A) formateado (III)

Sección 4.3. Tarjeta MiFare Classic (B)

En las dos figuras consecuentes (Figura 38 y Figura 39) se muestran las direcciones de memoria de una tarjeta MiFare Classic en formato hexadecimal.

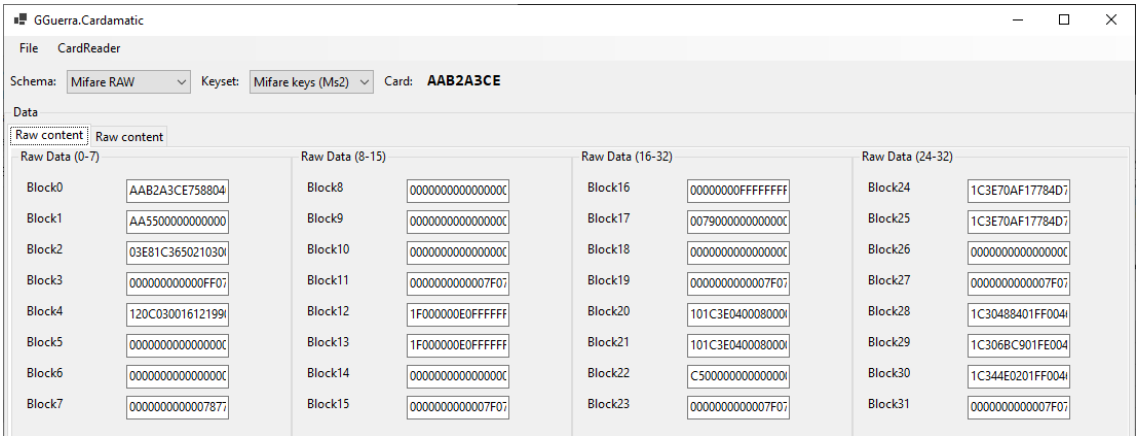

Figura 38 Contenido tarjeta MiFare Classic (B) en crudo (I)

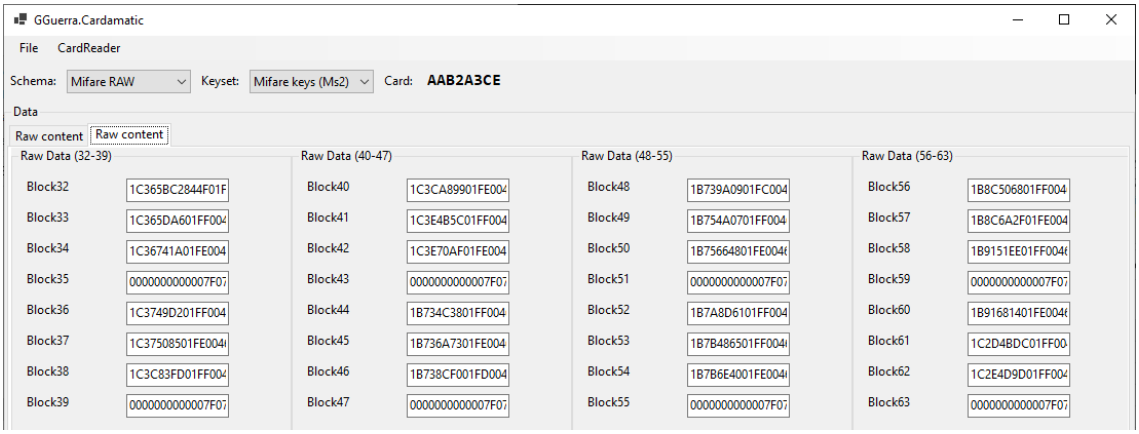

Figura 39 Contenido tarjeta MiFare Classic (B) en crudo (II)

Para este caso se ha utilizado el mismo esquema que el caso anterior, pero las claves de acceso de la tarjeta utilizadas han sido diferentes. El resultado de la decodificación de los datos de la tarjeta se presenta en las siguientes figuras (Figura 40, Figura 41 y Figura 42).

Figura 40 Contenido tarjeta MiFare Classic (B) formateado (I)

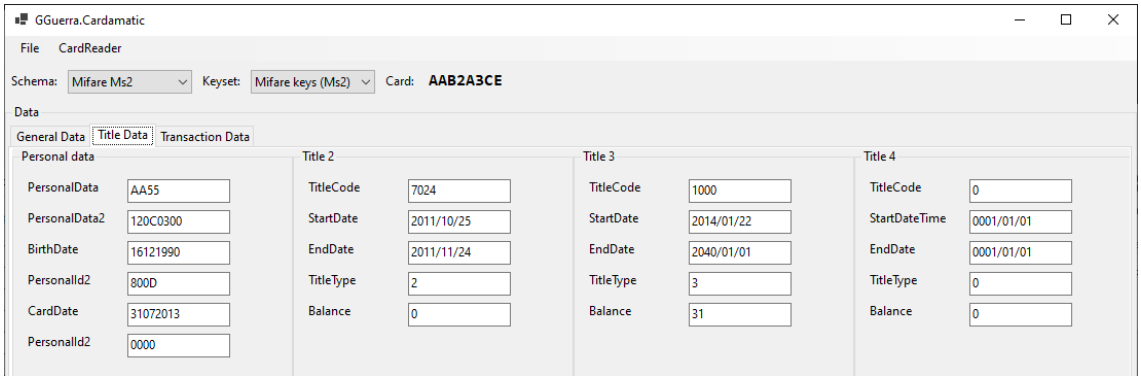

Figura 41 Contenido tarjeta MiFare Classic (B) formateado (II)

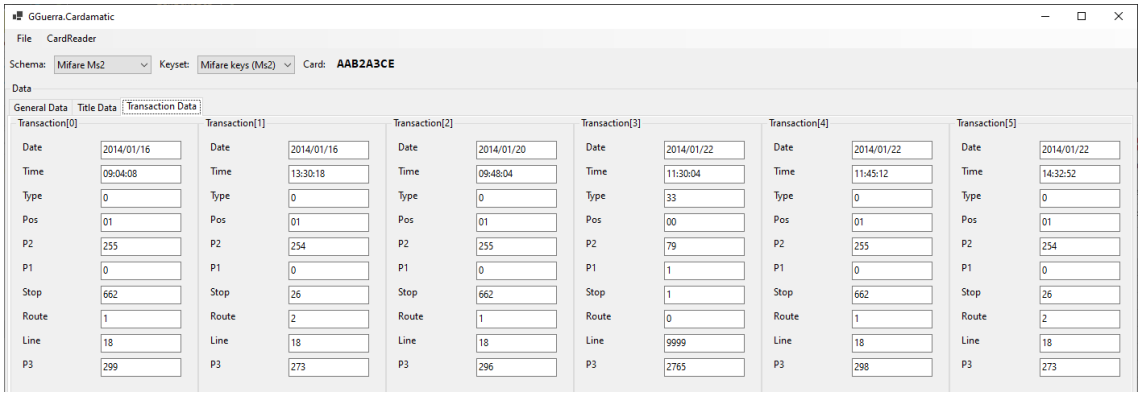

Figura 42 Contenido tarjeta MiFare Classic (B) formateado (III)

Sección 4.4. Tarjeta MiFare Classic (C)

En las Figura 43 y Figura 44 se muestran las direcciones de memoria de una tarjeta MiFare Classic en formato hexadecimal.

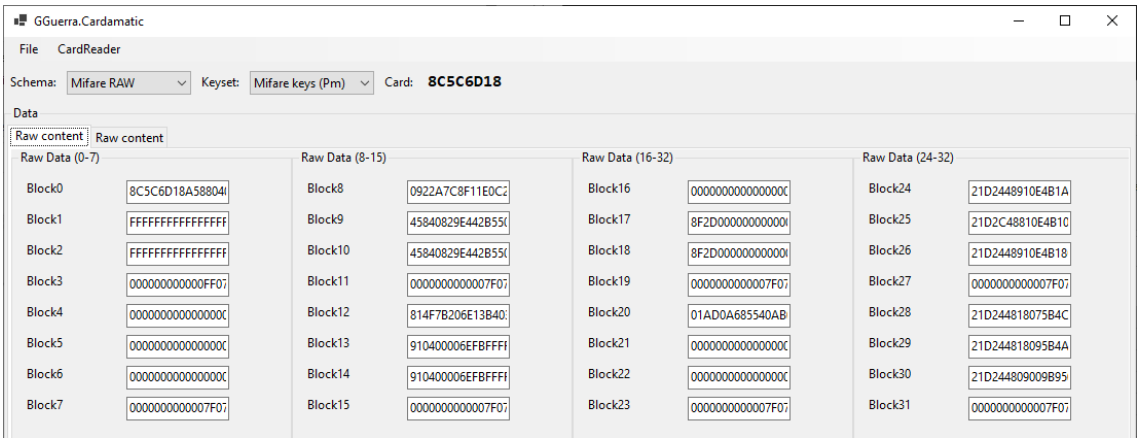

Figura 43 Contenido tarjeta MiFare Classic (C) en crudo (I)

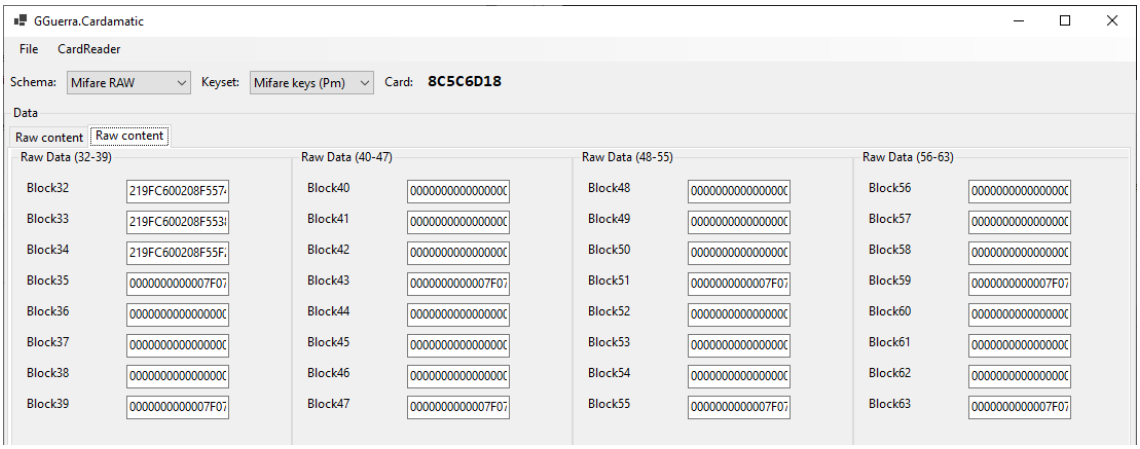

Figura 44 Contenido tarjeta MiFare Classic (C) en crudo (II)

El esquema de este tipo de tarjeta codificado en el DSL es el siguiente:

```
{
  "description": "Mifare Px",
  "ContactlessTechnology": "MifareClassic",
  "tabs": [
    {
      "description": "Card Data",
      "columns": [
        {
          "description": "General Data",
          "properties": [

            {
              "typeName": "System.UInt16",
              "name": "Version",
              "address": 8,
              "offset": 0,
              "length": 0,
              "offsetBits": 5,
              "lengthBits": 3
            },
            {
```

```
            "typeName": "System.UInt16",
            "name": "Company",
            "address": 8,
            "offset": 0,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 5
          },
          {
            "typeName": "System.UInt16",
            "name": "CardType",
            "address": 8,
            "offset": 1,
            "length": 0,
            "offsetBits": 5,
            "lengthBits": 3
          },
          {
            "typeName": "System.UInt16",
            "name": "StartDateDay",
            "address": 8,
            "offset": 1,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 5
          },
          {
            "typeName": "System.UInt16",
            "name": "StartDateMonth",
            "address": 8,
            "offset": 2,
            "length": 0,
            "offsetBits": 4,
            "lengthBits": 4
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "StartDateYear_H",
            "address": 8,
            "offset": 3,
            "length": 0,
            "offsetBits": 7,
            "lengthBits": 1
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "StartDateYear_L",
            "address": 8,
            "offset": 2,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 4
          },
          {
            "typeName": "System.UInt16",
            "name": "EndDateDay",
            "address": 8,
            "offset": 3,
            "length": 0,
            "offsetBits": 2,
            "lengthBits": 5
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "EndDateMonth_H",
            "address": 8,
            "offset": 4,
            "length": 0,
            "offsetBits": 5,
            "lengthBits": 3
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "EndDateMonth_L",
```

```
            "address": 8,
            "offset": 3,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "EndDateYear",
            "address": 8,
            "offset": 4,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 5
          },
          {
            "typeName": "System.Boolean",
            "name": "Active",
            "address": 8,
            "offset": 4,
            "length": 0,
            "offsetBits": 7,
            "lengthBits": 1
          }
        ]
      },
      {
        "description": "User Data",
        "properties": [
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "UserCity_H",
            "address": 8,
            "offset": 6,
            "length": 0,
            "offsetBits": 6,
            "lengthBits": 2
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "UserCity_L",
            "address": 8,
            "offset": 5,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 8
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "ProfileMask",
            "address": 8,
            "offset": 6,
            "length": 0,
            "offsetBits": 2,
            "lengthBits": 4
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "Profile_H",
            "address": 8,
            "offset": 7,
            "length": 0,
            "offsetBits": 4,
            "lengthBits": 4
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
            "name": "Profile_L",
            "address": 8,
            "offset": 6,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 2
          },
```

```
            {
              "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
              "name": "ProfileDateDay_H",
              "address": 8,
              "offset": 8,
              "length": 0,
              "offsetBits": 0,
              "lengthBits": 1
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
              "name": "ProfileDateDay_L",
              "address": 8,
              "offset": 7,
              "length": 0,
              "offsetBits": 0,
              "lengthBits": 4
            },
            {
              "typeName": "System.UInt16",
              "name": "ProfileDateMonth",
              "address": 8,
              "offset": 8,
              "length": 0,
              "offsetBits": 3,
              "lengthBits": 4
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
              "name": "ProfileDateYear_H",
              "address": 8,
              "offset": 9,
              "length": 0,
              "offsetBits": 6,
              "lengthBits": 2
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
              "name": "ProfileDateYear_L",
              "address": 8,
              "offset": 8,
              "length": 0,
              "offsetBits": 0,
              "lengthBits": 3
            }
          ]
        }
      ]
    }
  ]
}
```

Resultado del interprete con el esquema designado en la Figura 45.

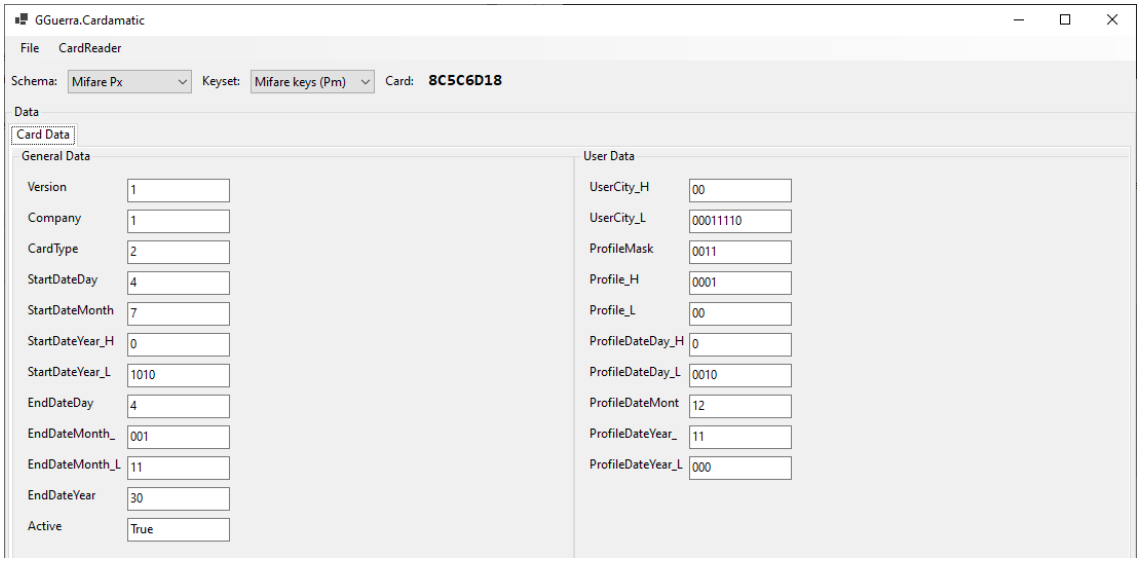

Figura 45 Contenido tarjeta MiFare Classic (C) formateado

Para un caso como es el de este esquema el DSL actual presenta un defecto, por el cual algunas propiedades de la tarjeta las cuales se encuentran en diferentes direcciones de memoria se deben partir en su parte baja (‘L’) y parte alta (‘H’) para que sean compatibles con el diseño actual.

Sección 4.5. Tarjeta MiFare DESFire

En las siguientes tres figuras (Figura 46, Figura 47 y Figura 48) se muestran direcciones de memoria agrupadas por directorios (o aplicaciones) dentro de una tarjeta MiFare DESFire en formato hexadecimal.

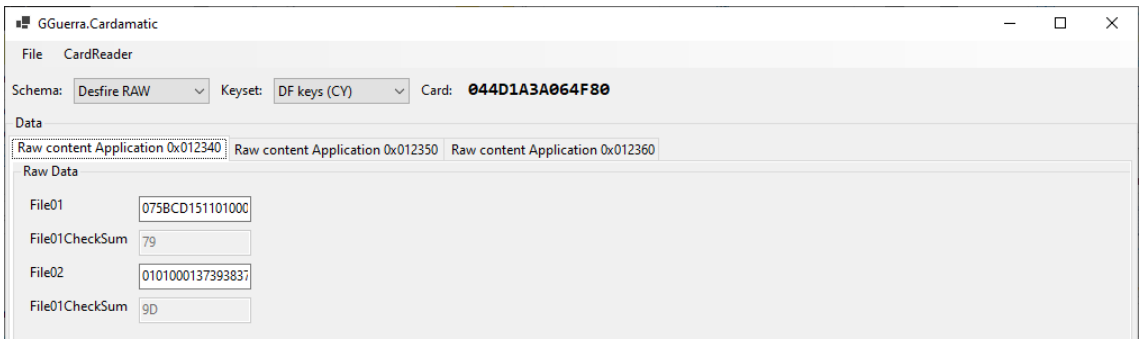

Figura 46 Contenido tarjeta MiFare DESFire en crudo (I)

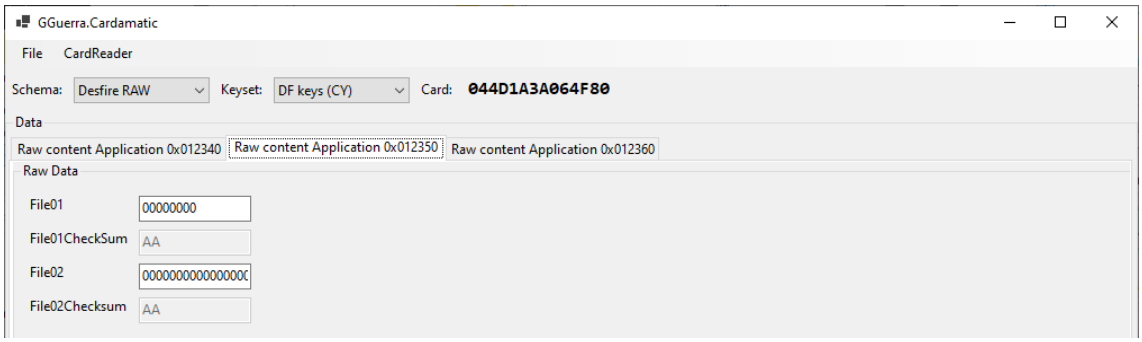

Figura 47 Contenido tarjeta MiFare DESFire en crudo (II)

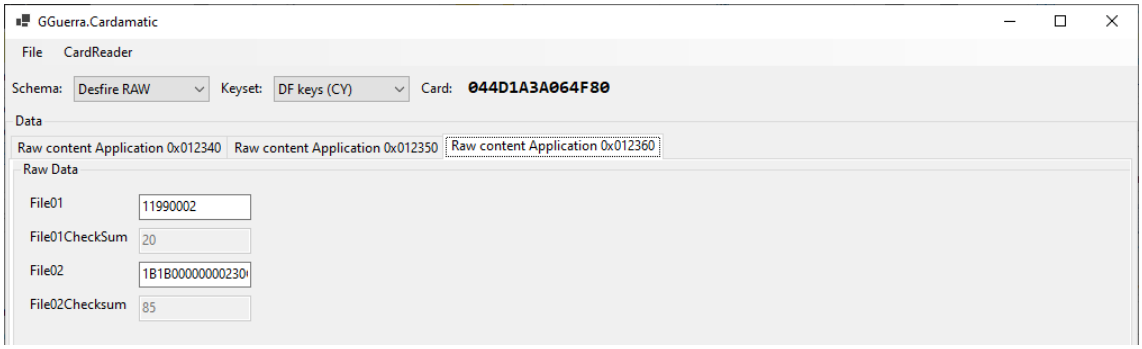

Figura 48 Contenido tarjeta MiFare DESFire en crudo (III)

Extracto del esquema del mapa de memoria codificado en el DSL.


```
{
  "Description": "Desfire Cy",
  "ContactlessTechnology": "Desfire",
  "Tabs": [
    {
      "description": "Card Application",
      "columns": [
        {
          "description": "General Data",
          "properties": [
            {
              "typeName": "System.UInt32",
              "name": "CardNumber",
              "address": "01234001",
              "offset": 0,
              "length": 4
            },
            {
              "typeName": "System.UInt16",
              "name": "Keys",
              "address": "01234001",
              "offset": 4,
              "length": 0,
              "offsetBits": 0,
              "lengthBits": 4
            },
            {
              "typeName": "System.UInt16",
              "name": "Version",
              "address": "01234001",
              "offset": 4,
              "length": 0,
              "offsetBits": 4,
              "lengthBits": 4
            },
            {
              "typeName": "System.Byte",
              "name": "CardType",
              "address": "01234001",
              "offset": 5,
              "length": 1
            },
            {
              "typeName": "System.UInt16",
              "name": "Pers",
              "address": "01234001",
              "offset": 6,
              "length": 2
            },
            {
              "typeName": "System.UInt16",
              "name": "Language",
              "address": "01234001",
              "offset": 8,
              "length": 1
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.Date.Date",
              "name": "StartDate",
              "address": "01234001",
              "offset": 9,
              "length": 2
            },
            {
              "typeName": "GGuerra.Cardamatic.Encoding.Date.Date",
              "name": "ExpirationDate",
              "address": "01234001",
              "offset": 11,
              "length": 2
            },
            {
              "typeName": "System.UInt16",
```

```
            "name": "Company",
            "address": "01234001",
            "offset": 13,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "RefDate",
            "address": "01234001",
            "offset": 15,
            "length": 1
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Checksum.Checksum",
            "name": "CheckSum",
            "address": "01234001",
            "offset": 16,
            "length": 1,
            "algorithm": {
              "offset": 0,
              "length": 16
            }
          }
        ]
      },
      {
        "description": "User Data",
        "properties": [
          {
            "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
            "name": "BirthDate",
            "address": "01234002",
            "offset": 0,
            "length": 4
          },
          {
            "typeName": "System.String",
            "name": "CardId",
            "address": "01234002",
            "offset": 4,
            "length": 12
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Checksum.Checksum",
            "name": "CheckSum",
            "address": "01234002",
            "offset": 16,
            "length": 1,
            "algorithm": {
              "offset": 0,
              "length": 16
            }
          }
        ]
      }
    ]
  },
  {
    "description": "Employee Application",
    "columns": [
      {
        "description": "EmployeIdentifier",
        "properties": [
          {
            "typeName": "System.UInt16",
            "name": "Type",
            "address": "01235001",
            "offset": 0,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "Equipment",
            "address": "01235001",
```

```
            "offset": 2,
            "length": 2
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Checksum.Checksum",
            "name": "File01CheckSum",
            "address": "01235001",
            "offset": 4,
            "length": 1,
            "algorithm": {
              "offset": 0,
              "length": 4
            }
          }
        ]
      },
      {
        "description": "EmployeService",
        "properties": [
          {
            "typeName": "GGuerra.Cardamatic.Encoding.HexString.HexString",
            "name": "DateTime",
            "address": "01235002",
            "offset": 0,
            "length": 3
          },
          {
            "typeName": "System.UInt16",
            "name": "Bus",
            "address": "01235002",
            "offset": 3,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "Line",
            "address": "01235002",
            "offset": 5,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "Route",
            "address": "01235002",
            "offset": 7,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "Stop",
            "address": "01235002",
            "offset": 9,
            "length": 2
          },
          {
            "typeName": "System.UInt16",
            "name": "Company",
            "address": "01235002",
            "offset": 11,
            "length": 2
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Checksum.Checksum",
            "name": "File02Checksum",
            "address": "01235002",
            "offset": 13,
            "length": 1,
            "algorithm": {
              "offset": 0,
              "length": 4
            }
          }
        ]
      }
```

```
    ]
  },
  {
    "description": "Transport Application",
    "columns": [
      {
        "description": "General Data",
        "properties": [
          {
            "typeName": "System.UInt16",
            "name": "Keys",
            "address": "01236001",
            "offset": 0,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 4
          },
          {
            "typeName": "System.UInt16",
            "name": "Version",
            "address": "01236001",
            "offset": 0,
            "length": 0,
            "offsetBits": 4,
            "lengthBits": 4
          },
          {
            "typeName": "System.UInt16",
            "name": "TitlesLength",
            "address": "01236001",
            "offset": 1,
            "length": 0,
            "offsetBits": 0,
            "lengthBits": 3
          },
          {
            "typeName": "System.UInt16",
            "name": "TransactionsLength",
            "address": "01236001",
            "offset": 1,
            "length": 0,
            "offsetBits": 3,
            "lengthBits": 4
          },
          {
            "typeName": "System.Boolean",
            "name": "TestFlag",
            "address": "01236001",
            "offset": 1,
            "length": 0,
            "offsetBits": 7,
            "lengthBits": 1
          },
          {
            "typeName": "System.UInt16",
            "name": "TransactionCounter",
            "address": "01236001",
            "offset": 2,
            "length": 2
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Checksum.Checksum",
            "name": "CheckSum",
            "address": "01236001",
            "offset": 4,
            "length": 1,
            "algorithm": {
              "offset": 0,
              "length": 4
            }
          }
        ]
      },
      {
```

```
    "description": "Transaction Data",
    "properties": [
      {
        "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
        "name": "Priorities",
        "address": "01236002",
        "offset": 0,
        "length": 1
      },
      {
        "typeName": "System.UInt16",
        "name": "Order",
        "address": "01236002",
        "offset": 1,
        "length": 1
      },
      {
        "typeName": "System.UInt32",
        "name": "ListId",
        "address": "01236002",
        "offset": 2,
        "length": 4
      },
      {
        "typeName": "System.UInt16",
        "name": "TitlesUsed",
        "address": "01236002",
        "offset": 6,
        "length": 0,
        "offsetBits": 0,
        "lenthBits": 2
      },
      {
        "typeName": "GGuerra.Cardamatic.Encoding.BinaryString.BinaryString",
        "name": "TitlesActive",
        "address": "01236002",
        "offset": 6,
        "length": 0,
        "offsetBits": 2,
        "lengthBits": 4
      },
      {
        "typeName": "System.Boolean",
        "name": "CardActiveFlag",
        "address": "01236002",
        "offset": 6,
        "length": 0,
        "offsetBits": 6,
        "lenthBits": 1
      },
      {
        "typeName": "System.Boolean",
        "name": "CardBlockedFlag",
        "address": "01236002",
        "offset": 6,
        "length": 0,
        "offsetBits": 7,
        "lenthBits": 1
      },
      {
        "typeName": "System.Boolean",
        "name": "CardRestoredFlag",
        "address": "01236002",
        "offset": 7,
        "length": 0,
        "offsetBits": 0,
        "lenthBits": 1
      },
      {
        "typeName": "System.UInt16",
        "name": "TransactionPointer",
        "address": "01236002",
        "offset": 7,
        "length": 0,
```

```
            "offsetBits": 1,
            "lenthBits": 7
          },
          {
            "typeName": "GGuerra.Cardamatic.Encoding.Checksum.Checksum",
            "name": "Checksum",
            "address": "01236002",
            "offset": 10,
            "length": 1,
            "algorithm": {
              "offset": 0,
              "length": 10
            }
          }
        ]
      }
    ]
  }
]
}
```

El resultado de la estructura previa exhibiendo el contenido de la tarjeta correctamente formateado se muestra en Figura 49, Figura 50 y Figura 51.

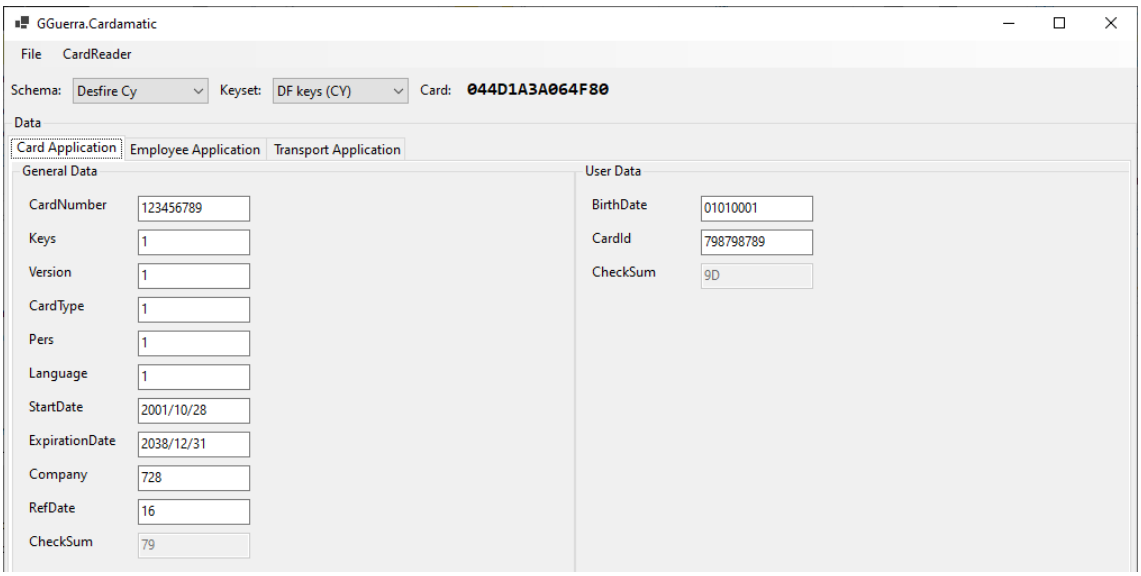

Figura 49 Contenido tarjeta MiFare DESFire formateado (I)

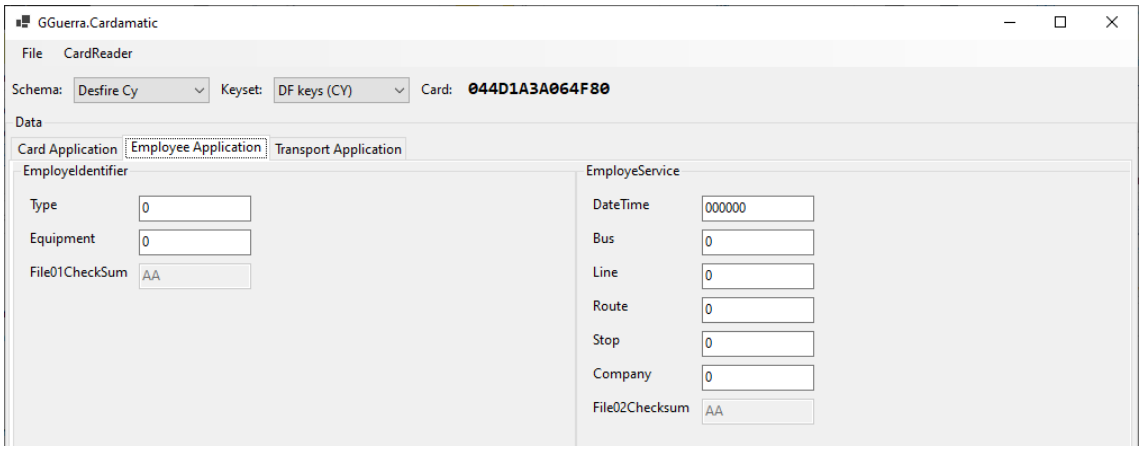

Figura 50 Contenido tarjeta MiFare DESFire formateado (II)

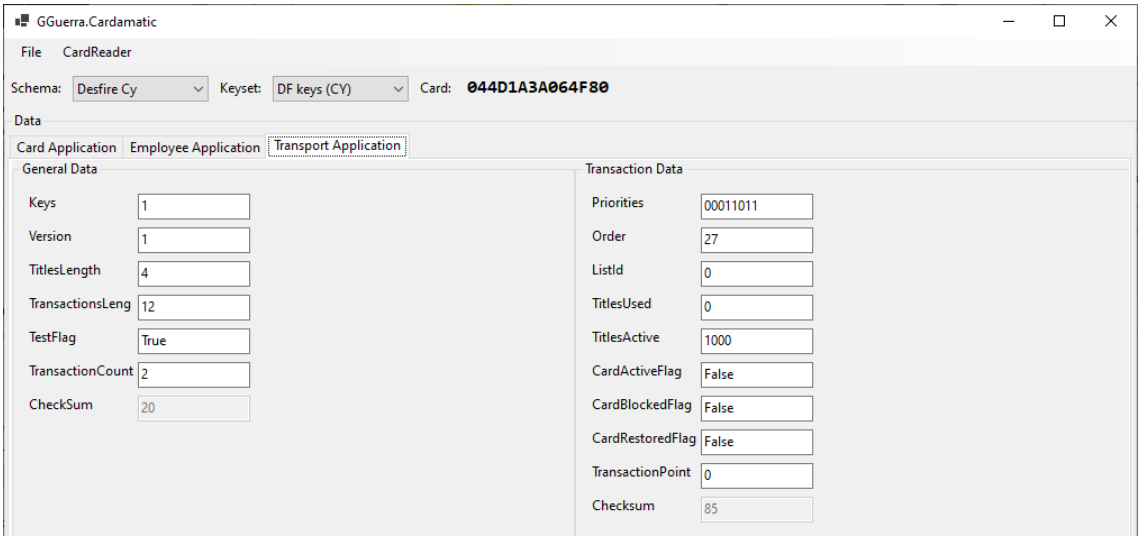

Figura 51 Contenido tarjeta MiFare DESFire formateado (III)

Capítulo 5. Conclusiones

En este capítulo de conclusiones, se resumen los resultados y relevancia del uso de un lenguaje específico de dominio (DSL) para gestionar mapas de memoria en tarjetas sin contacto. Se examinan las ventajas de esta solución y se señalan áreas de mejora. El análisis completo de las contribuciones y limitaciones destaca cómo el DSL mejora la manipulación y seguridad de los datos en tarjetas sin contacto, brindando eficiencia y flexibilidad en este campo.

Sección 5.1. Contribuciones

La implementación de un lenguaje específico de dominio (DSL) para la gestión de mapas de memoria en tarjetas sin contacto presenta una serie de mejoras significativas en diversos aspectos clave. Este enfoque permite una optimización integral en la administración de tarjetas, simplificando la operación y gestión de diferentes tipos de tarjetas y tecnologías. A continuación, se detallan una serie de mejoras que resaltan los beneficios prácticos y la eficacia que aporta el uso de un DSL en este contexto.

1- **Mayor Productividad**: El DSL aumenta la productividad al permitir que los expertos en el dominio adapten, mantengan y evolucionen una única herramienta para gestionar diversas tarjetas sin contacto. Las diferencias entre tecnologías y esquemas están parametrizadas mediante el DSL, simplificando la gestión.
2- **Flexibilidad**: La parametrización a través del DSL permite especificar diferentes claves de autenticación para un mismo esquema de memoria. Esto resulta en una gestión más detallada y controlada de los accesos a áreas de memoria de la tarjeta.
3- **Seguridad Mejorada**: La solución propuesta es compatible con sistemas de mayor seguridad, ya que admite el uso de claves diversificadas. Esto brinda una capa adicional de protección en entornos sensibles.
4- **Enfoque Específico**: Al estar enfocado en un dominio específico, los usuarios pueden aprender a utilizar el lenguaje de manera más eficiente sin lidiar con complejidades innecesarias presentes en lenguajes de propósito general, como librerías o APIs.
5- **Ajuste Natural**: Las expresiones y estructuras del DSL se ajustan de manera natural y cercana a los conceptos y terminología propios del dominio. Esto facilita la comprensión y uso para los usuarios.
6- **Minimización de Errores**: El uso del DSL reduce la probabilidad de errores al trabajar con tarjetas sin contacto, ya que su estructura y enfoque específico disminuyen las posibilidades de malinterpretación.
7- **Consistencia**: La estandarización proporcionada por el DSL asegura la coherencia en la gestión de diferentes tipos de tarjetas, lo que resulta en una mayor confiabilidad y uniformidad en las operaciones.
8- **Agilidad en Evolución**: Gracias a la parametrización, el DSL permite adaptarse y evolucionar con facilidad a los cambios en las tecnologías y esquemas de tarjetas sin contacto, asegurando que la herramienta permanezca actualizada.
9- **Reducción de Barreras de Entrada**: El DSL simplifica el aprendizaje y uso del sistema, lo que disminuye las barreras de entrada para nuevos usuarios y fomenta una adopción más rápida y efectiva.

10- **Eficiencia en la Formación**: La naturaleza específica del DSL facilita la formación de usuarios en su uso, ya que se centra en los aspectos relevantes del dominio y evita complicaciones innecesarias.

Estas mejoras resaltan cómo el uso del DSL beneficia tanto la eficiencia como la efectividad en la gestión de tarjetas sin contacto, al tiempo que mejora la seguridad y la experiencia del usuario.

Sección 5.2. Trabajo futuro

Ya que los DSL se caracterizan por tener una estructura clara y restringida que los hace fáciles de comprender, modificar y ampliar se detallan una serie de características como trabajo futuro de menor a mayor complejidad.

1- Evolución de la sintaxis para **campos segmentados**: Una mejora primordial radica en la evolución de la sintaxis del DSL para permitir la manipulación de campos segmentados en diferentes ubicaciones de memoria no consecutivas en la tarjeta. Esta capacidad añadiría flexibilidad y eficiencia al lenguaje, facilitando operaciones específicas en áreas fragmentadas. Afortunadamente, esto no representaría un desafío significativo debido a la naturaleza altamente mantenible y extensible de los DSL.
2- **Operaciones lógicas** sobre áreas predefinidas: Una mejora adicional sería la posibilidad de definir y ejecutar operaciones lógicas en áreas predefinidas de la tarjeta. Esto permitiría realizar acciones específicas, como el reinicio o el incremento de campos particulares, sin afectar al resto de los datos. Esta característica añadiría un nivel de control adicional en la manipulación de los datos. Ampliar el DSL para admitir operaciones sobre las tarjetas sería un proceso factible, aunque fuera una tarea de mayor envergadura.
3- **Integración de otras tecnologías** de tarjetas sin contacto: Una mejora estratégica sería la integración de tecnologías de tarjetas sin contacto más allá de la familia MiFare. La inclusión de estándares como Calypso, el cual es un estándar desarrollado por un grupo de operadores de diferentes países, comúnmente utilizada en países del centro de Europa; o Felica, tecnología propietaria de Sony que se utiliza ampliamente en Asia. Esto aumentaría la versatilidad y utilidad del DSL, aunque requeriría un profundo conocimiento de estas tecnologías para su implementación efectiva.

Referencias

[1] N. Moebius, K. Stenzel, H. Grandy y W. Reif, «Model-Driven Code Generation for Secure Smart Card Applications,» de *2009 Australian Software Engineering Conference*, Gold Coast, Australia, 2009.

[2] N. Moebius, K. Stenzel, H. Grandy y W. Reif, «SecureMDD: A Model-Driven Development Method for Secure Smart Card,» de *2009 International Conference on Availability, Reliability and Security*, Fukuoka, Japan, 2009.

[3] A. Nikseresht y K. Ziarati, «MDA Based Framework for the Development of Smart Card Based Application,» *Lecture Notes in Engineering and Computer Science,* vol. 2188, pp. 263-268, 2011.

[4] J. Xu y F. Xie, «Developing Smart Card Application with PC/SC,» de *011 International Conference on Internet Computing and Information Services*, Hong Kong, China, 2011.

[5] F. Haim, A. Bergeret, A. González y I. Benavente, «Procedures and lab setup developed to test MIFARE based transportation devices compliance,» de *11th Latin American Test Workshop*, Punta del Este, Uruguay, 2010.

[6] A. Bejo, R. Winata y S. S. Kusumawardani, «Prototyping of Class-Attendance System Using Mifare 1K Smart Card and Raspberry Pi 3,» de *International Symposium on Electronics and Smart Devices (ISESD)*, Bandung, Indonesia, 2018.

[7] A. P. Abellon, C. j. Ariola, E. Blancaflor, A. K. Danao, D. Medel y M. Z. Santos, «Risk Assessments of Unattended Smart Contactless Cards,» de *2021 IEEE 8th International Conference on Industrial Engineering and Applications*, Chengdu, China, 2021.