Aalto University
School of Science

Some notes about MIFARE DESFire EV1 / Ultralight C

Daniel Andrade, Instituto Superior Técnico, daniel.andrade@alumni.aalto.fi
Sandeep Tamrakar, Aalto University, sandeep.tamrakar@aalto.fi
Tuomas Aura, Aalto University, tuomas.aura@aalto.fi

September 9, 2013

Contents

Chapter 1

MIFARE DESFire EV1

MIFARE DESFire EV1 is the evolution of MIFARE DESFire (MF3ICD40). The security of MF3ICD40 was broken in 2011, when researchers were able to retrieve the secret key by means of a power analysis attack [6].

DESFire EV1 is a contactless smart card that holds up to 28 applications. Each application can have up to fourteen keys associated with it and contain up to 32 files. There are five different file types. Each file has its own file settings, which include communication mode and access control parameters. The communication modes are plain, mac'ed and enciphered and are discussed in Section 1.3.4. Each DESFire EV1 has a single PICC master key and supports DES, 2K3DES, 3K3DES and AES. In comparison, its predecessor, DESFire, only holds up to sixteen files per application and supports DES and 3DES. DESFire EV1 is backward compatible with DESFire [4].

This chapter discusses the operations and data storage of DESFire EV1 and its security mechanisms. It focuses on the cryptographic operations taking place, authentication protocol and data transmission modes between the reader and the card.

The specification of DESFire EV1 is not publicly available and other resources had to be used instead to reach a deep understanding of its functionality. An old specification [7] of its predecessor, DESFire, can be found online and Kasper, von Maurich, Oswald, and Paar [2, sec. 3.2] expose the AES authentication protocol for DESFire EV1; the libfreefare [3] project aims at providing a C library for the manipulation of MIFARE cards, but unfortunately the code is not documented and the card reader used for the thesis is incompatible with this library. Additionally, pieces of information regarding MIFARE smart cards is found in blogs and forums via search engines, albeit often incomplete.

1.1 Commands

Level	Commands
Security-related	`Authenticate`, `ChangeKeySettings`, `SetConfiguration`, `ChangeKey`, `GetKeyVersion`
PICC-level	`CreateApplication`, `DeleteApplication`, `GetApplicationsIDs`, `FreeMemory`, `GetDFNames`, `GetKeySettings`, `SelectApplication`, `FormatMF3ICD81`, `GetVersion`, `GetCardUID`
Application-level	`GetFileIDs`, `GetFileSettings`, `ChangeFileSettings`, `CreateStdDataFile`, `CreateBackupDataFile`, `CreateValueFile`, `CreateLinearRecordFile`, `CreateCyclicRecordFile`, `DeleteFile`
File-level	`ReadData`, `WriteData`, `GetValue`, `Credit`, `Debit`, `LimitedCredit`, `WriteRecord`, `ReadRecords`, `ClearRecordFile`, `CommitTransaction`, `AbortTransaction`

Table 1.1: List of commands of DESFire EV1, grouped by level.

DESFire EV1 commands are divided in four levels: security-related commands, PICC-level commands, application-level commands and data manipulation (file-level) commands. Table 1.1 lists the available commands for the manipulation of the contents of DESFire EV1. The commands were retrieved from the short data sheet of the card, made available by NXP Semiconductors [4].

Security-related commands include authentication and key-related operations. PICC-level commands include application and memory manipulation operations. Application-level commands include operations to manipulate files such as file creation and file deletion. File-level commands include operations that manipulate data such as reading from files and writing to files. See Table 1.1 for a list of commands organized by level.

1.2 File system

DESFire EV1 has a flexible file system capable of containing up to 28 applications and up to 32 files in each application. One can think of the file system as directories containing files, where the directories are the applications. Each application is represented by a 3-byte application identifier (AID), set on application creation. Each file is represented by a 1-byte file number, set on file creation. Access to files may require a preceding authentication, depending on the file access rights. Keys and access rights are discussed in Section 1.3.3 and in Section 1.3.4.2 respectively.

File type	**Description**	**Commands**
Standard data file	Storage of unformatted user data	`ReadData` `WriteData`*

Backup data file	Storage of unformatted user data + integrated backup mechanism	`ReadData`
		`WriteData`*
		`CommitTransaction`
		`AbortTransaction`

Value file	Storage and manipulation of a 32-bit signed integer	`ReadData`
		`WriteData`*
		`GetValue`
		`Credit`*
		`Debit`*
		`LimitedCredit`*
		`CommitTransaction`
		`AbortTransaction`

Linear record file	Storage of structured user data
	(e.g. loyalty programs)
		`WriteRecord`*
		`ReadRecords`
		`ClearRecordFile`*
		`CommitTransaction`
		`AbortTransaction`

Cyclic record file	Storage of structured user data + automatically overwrite oldest record when full
	(e.g. logging transactions)
		`WriteRecord`*
		`ReadRecords`
		`ClearRecordFile`*
		`CommitTransaction`
		`AbortTransaction`

Table 1.2: DESFire EV1 file types and data manipulation operations. The starred commands require validation.

Table 1.2 summarizes the main points of the types of files available on the card. The files contained by an application can be of the same type or of different types. DESFire EV1 supports five different types of files, each one with a different purpose and with a set of operations that can be performed on it. The types of files are standard data files, backup data files, value files, linear record files, and cyclic record files. Standard data files and backup data files are used for the storage of unformatted user data. Value files are used for the storage and manipulation of a 32-bit signed integer value. Linear record files and cyclic record files are used for the storage of structured user data. Structured user data are also called records. A record is a single piece of data organized in a predetermined way and with size, in bytes, set on file creation. A linear record file will become full of records at some point, after which it has to be cleared for further writing. A cyclic record file automatically overwrites the oldest record once it is full. Backup data files, value files, linear record files and cyclic record files include an integrated backup mechanism and require a `CommitTransaction` to validate commands that modify data stored on the card. Alternatively, an `AbortTransaction` can be used to abort the command and proceed with other operations on the card. Multiple commands may be issued, within the same application, with a single commit or abort command at the end. This applies to commands targeting the same file or different files, irrespective of the file type. If commands requiring validation are sent to the card and a `SelectApplication` takes place before a `CommitTransaction`, the data modifications requiring validation are discarded. The backup mechanism takes an additional portion of memory from the card. For backup data files, it consumes double the NV memory when compared with standard data files with the same size. For cyclic record files, it consumes one extra record, which must be taken into account when defining the maximum number of records upon file creation.

1.3 Security

The security of DESFire EV1 relies on multiple mechanisms. It uses error-detecting codes to detect unintentional changes to the data, message authentication codes to ensure the data authenticity and integrity, and encryption algorithms to ensure data confidentiality. The error-detecting codes used are CRC16 and CRC32, implemented according with ISO/IEC 14443A and ITU-T Recommendation V.42 respectively. The message authentication codes are CBC-MAC and CMAC and the encryption algorithms are DES, 2K3DES, 3K3DES and AES. The cryptographic primitives of DESFire EV1 are further explained in Section 1.3.1.

Each DESFire EV1 can be identified by its unique UID, which is programmed into the device during production and cannot be altered. This is done by writing into a manufacturer reserved part of the NV memory and write-protecting those bytes. MF3ICD81 conforms to the Smartcard IC Platform Protection Profile and is certified by the German Federal Office for

Information Security according with the Common Criteria for Information Technology Security Evaluation (CC) at level EAL 4 augmented [1, 5].

1.3.1 Cryptographic primitives

DESFire EV1 supports the following cryptographic primitives: CBC-MAC, CMAC and encryption and decryption using the ciphers Triple DES with 56/112/168-bit keys and AES. Each message authentication code is associated with a cipher. CBC-MAC is used in conjunction with DES and 2K3DES and CMAC is used in conjunction with 3K3DES and AES.

There is a notion of a *global IV*, where the input or output of an encryption, decryption or CMAC operation is used as initialization vector for the next cryptographic operation. This only applies to 3K3DES and AES. The global IV is defined as the last block of the ciphertext resulting from an encryption, as the last block of the ciphertext about to be decrypted and as the result of a CMAC operation. These cryptographic operations—encryption, decryption and CMAC calculation—use the session key created after a successful authentication as secret key. The global IV is set to zeros after the successful authentication and is only applied while the authenticated state is valid.

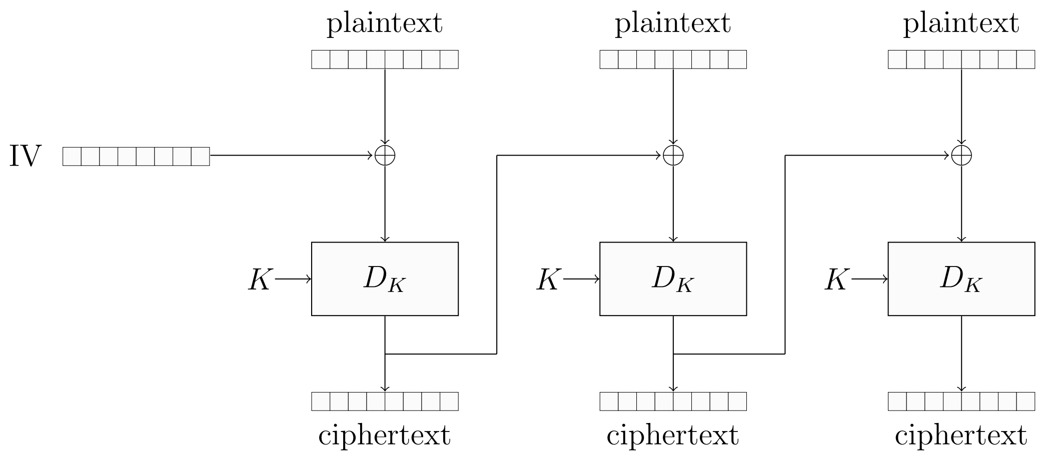
The CBC-MAC is calculated by applying Triple DES encryption in CBC mode to the data. It is used with DES and 2K3DES, which has a block size of 8 bytes. Since the data length must be multiple of 8 bytes it is padded with zeros if required. The MAC is defined as the first half of the last block, that is, the first 4 bytes of the last 8-byte block.

The CMAC is calculated according with the NIST Special Publication 800-38B, with the modification that the encryption function, during MAC generation, receives the global IV as its initialization vector instead of an IV composed by zeros. For 3K3DES, the resulting 8-byte CMAC is used as is. For AES, the first half of the resulting block is used as CMAC, that is, the first 8 bytes of the resulting 16-byte block.

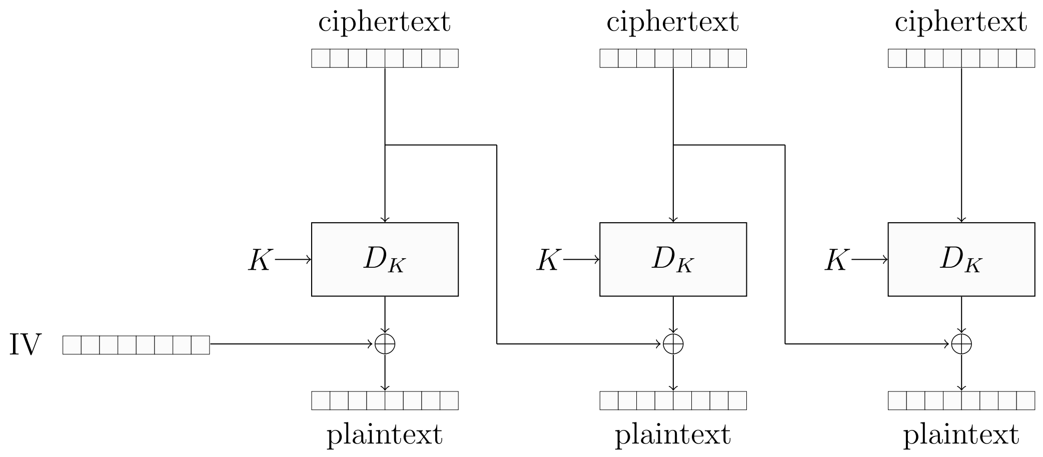
The data length is included in the commands that read data and in the commands that write data. The padding of the CBC-MAC and the padding of the CMAC is only used for the computation and is not exchanged between PICC and PCD.

When using a DES or 2K3DES cipher, the PCD always decrypts the data and the PICC always encrypts the data.[1] The cryptographic operation

[1]The DESFire MF3ICD40 is only able to encrypt data in order to save on hardware implementation costs. The DES and 2K3DES related operations on DESFire EV1 are backward compatible with DESFire MF3ICD40.



(a) CBC send mode for PCD.



(b) CBC receive mode for PCD.

Figure 1.1: CBC send mode and CBC receive mode for PCD.

is done in either CBC send mode or CBC receive mode. In CBC send mode, the PCD performs the logical operation XOR before decrypting a block of data. The data block being decrypted is xor'ed before the decryption, with the previously decrypted data block, as seen in Figure 1.1(a). In CBC receive mode, the PCD performs the logical operation XOR after decrypting a block of data. The data block being decrypted is first decrypted and only then it is xor'ed, with the previous 8-byte block of ciphertext, as seen in Figure 1.1(b). In both cases and since there is no previous block, the first block being decrypted is xor'ed with an all-zero initialization vector.[2]

[2]For all practical purposes, this returns the first block without any modification, making the XOR operation redundant.

When using 3K3DES or AES, the PCD encrypts when sending data and decrypts when receiving data. CBC mode is used in both cases. In comparison to DES and 2K3DES, there is an IV maintained between cryptographic operations: the global IV. The first block of data being encrypted or being decrypted is xor'ed with this IV and not with the all-zero IV. Like previously mentioned, this is the same IV used in the encryption operation that takes place during the MAC generation phase of CMAC.

Operation	**DES/2K3DES**	**3K3DES/AES**
CBC-MAC	DES/2K3DES encryption + CBC	
CMAC		CMAC + reuse of IV in MAC generation phase
Encryption	DES/2K3DES decryption + CBC send mode	3K3DES/AES encryption + CBC + reuse of IV
Decryption	DES/2K3DES decryption + CBC receive mode	3K3DES/AES decryption + CBC + reuse of IV

Table 1.3: Cryptographic operations in DESFire EV1.

The cryptographic primitives detailed in the previous paragraphs are outlined in Table 1.3. The operations for DES are essentially the same for 2K3DES and the same relation applies to 3K3DES and AES.

1.3.2 Authentication protocol

A mutual three-pass authentication may take place before transmitting data, which ensures that both the PCD and the PICC are in possession of a common secret. This secret, the secret key, is stored on the PICC and the PCD is required to have the correct key for the authentication to succeed. A successful mutual authentication results in a session key that can be used to protect subsequent data transmissions. The data transmission modes supported by DESFire EV1 are described in Section 1.3.4.

The authentication protocol is similar for all ciphers. The differences are on the length of the random numbers generated, the algorithms used for the encryption of data and for the decryption of data, and the session key generation algorithm. The length of the random numbers is 8 bytes for DES and 2K3DES, and 16 bytes for 3K3DES and AES. The encryption and decryption algorithms are presented in Section 1.3.1, and the session key generation algorithm is presented in Table 1.4. A diagram of the DES and 2K3DES authentication protocol is depicted in Figure 1.2, and a diagram of

Cipher	**Session key**
DES	$RndA_{0-3} \| RndB_{0-3}$
2K3DES	$RndA_{0-3} \| RndB_{0-3} \| RndA_{4-7} \| RndB_{4-7}$
3K3DES	$RndA_{0-3} \| RndB_{0-3} \| RndA_{6-9} \| RndB_{6-9} \| RndA_{12-15} \| RndB_{12-15}$
AES	$RndA_{0-3} \| RndB_{0-3} \| RndA_{12-15} \| RndB_{12-15}$

Table 1.4: Session key generation for DESFire EV1 using *RndA* and *RndB*.

the 3K3DES and AES authentication protocol is depicted in Figure 1.3. The authentication protocol is always initiated by the PCD and it is composed by the following steps:

1. The PCD sends an authentication request to the PICC, along with a 1-byte key number. This key number references the secret key to be used during the authentication protocol. The secret key can be part of an application or it can be the PICC master key, depending on the selected AID. After powering up the PICC, the 3-byte AID `00 00 00`$_\mathtt{H}$ is implicitly selected.

2. The PICC receives the authentication command and generates and encrypts a random number *RndB*. The length of the random number generated and the algorithm used for encryption are related to the cipher associated with the key number received from the PCD. The resulting ciphertext is sent to the PCD as a response.

3. The PCD receives and decrypts the response obtaining the random number *RndB* generated by the PICC. It then rotates *RndB* one byte to the left yielding $x_2 = RndB'$. The PCD generates its own random number *RndA* and concatenates $RndB'$ to it. $RndA \| RndB'$ is encrypted and sent to the PICC.

4. The PICC decrypts the received message, rotates its *RndB* to the left and compares it with the decrypted $RndB'$ sent by the PCD. If the match fails, the PICC returns an error code to the PCD. Otherwise, it rotates *RndA* one byte to the left, yielding $RndA'$ (x_8). $RndA'$ is encrypted and sent to the PCD.

5. The PCD decrypts the received message, rotates its *RndA* one byte to the left and compares it with the $RndA'$ received from the PICC. If the match fails ($x_{10} \neq x_{11}$), the PCD may abort the authentication protocol at this point. If the verification is successful ($x_{10} = x_{11}$), then the PCD is ready to generate a session key.

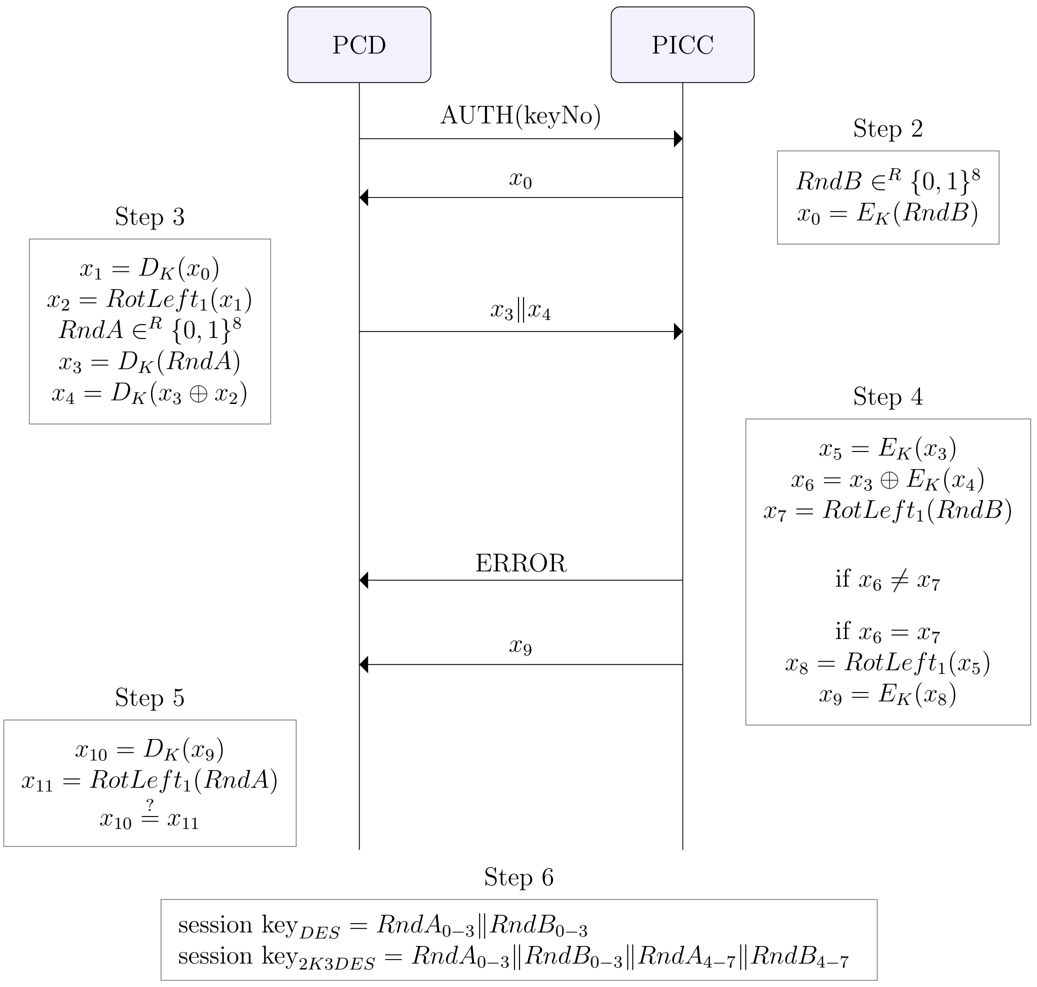


Figure 1.2: DES and 2K3DES authentication protocol for DESFire EV1.

6. The session key is generated by the PCD and by the PICC using both *RndA* and *RndB* according with Table 1.4. For 3K3DES and for AES, the global IV is reset to zeros at this point and is reused between all subsequent cryptographic operations. For DES and for 2K3DES, cryptographic operations are independent from each other and always start with an IV set to zeros.

1.3.3 Keys

There is a single PICC master key and up to fourteen keys per application. Each application always has at least one key: the application master key. The PICC master key and the application master key have the key number

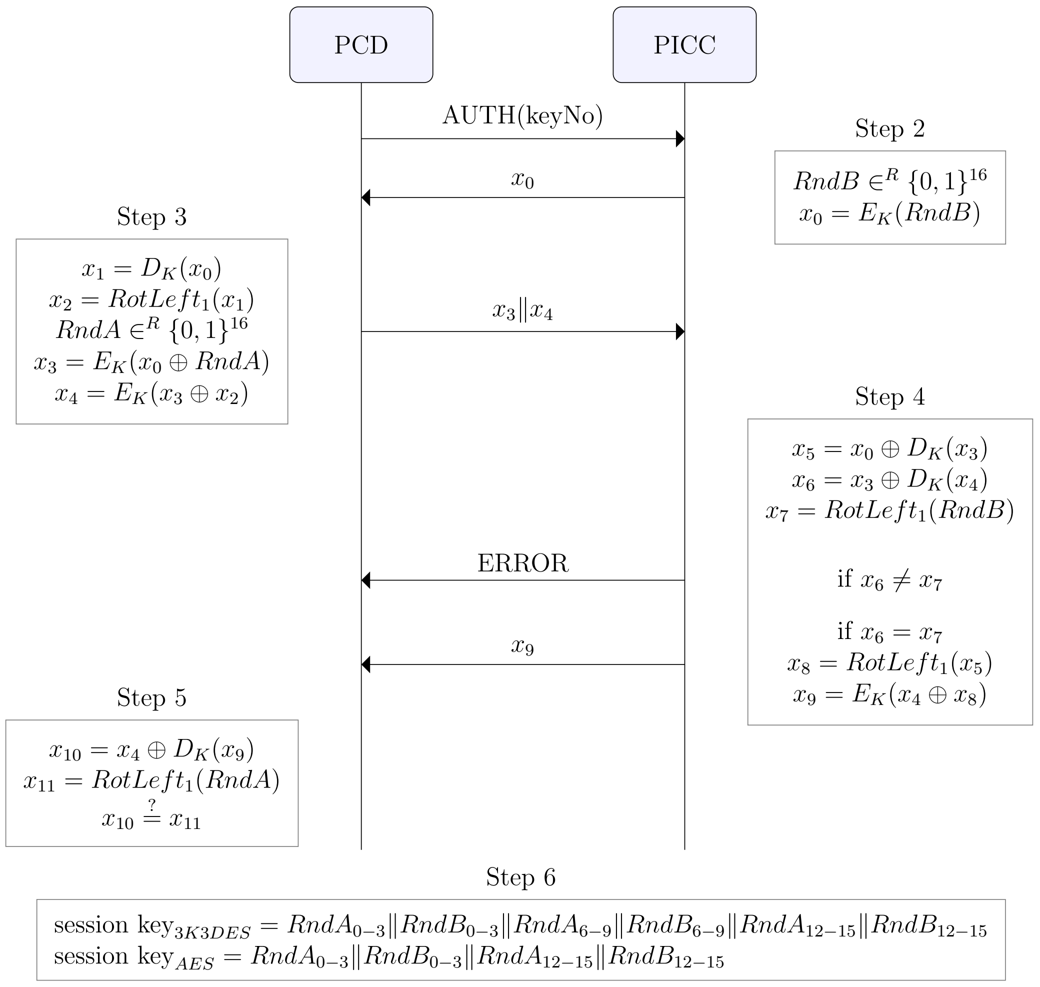

PCD
PICC
AUTH(keyNo)
Step 2
$RndB \in^R \{0,1\}^{16}$
$x_0 = E_K(RndB)$
x_0
Step 3
$x_1 = D_K(x_0)$
$x_2 = RotLeft_1(x_1)$
$RndA \in^R \{0,1\}^{16}$
$x_3 = E_K(x_0 \oplus RndA)$
$x_4 = E_K(x_3 \oplus x_2)$
$x_3 \| x_4$
Step 4
$x_5 = x_0 \oplus D_K(x_3)$
$x_6 = x_3 \oplus D_K(x_4)$
$x_7 = RotLeft_1(RndB)$
if $x_6 \neq x_7$
ERROR
if $x_6 = x_7$
$x_8 = RotLeft_1(x_5)$
$x_9 = E_K(x_4 \oplus x_8)$
x_9
Step 5
$x_{10} = x_4 \oplus D_K(x_9)$
$x_{11} = RotLeft_1(RndA)$
$x_{10} \overset{?}{=} x_{11}$
Step 6
session key$_{3K3DES} = RndA_{0-3} \| RndB_{0-3} \| RndA_{6-9} \| RndB_{6-9} \| RndA_{12-15} \| RndB_{12-15}$
session key$_{AES} = RndA_{0-3} \| RndB_{0-3} \| RndA_{12-15} \| RndB_{12-15}$

Figure 1.3: 3K3DES and AES authentication protocol for DESFire EV1.

00_H. For certain commands, a preceding authentication with a specific key is a requisite for their successful completion. For instance, a preceding authentication is compulsory when changing a key. For other commands, however, whether a preceding authentication is required for a successful completion or not depends on the access rights of files, explained in Section 1.3.4, and on the PICC master key settings or on the application master key settings.

The PICC master key settings apply to PICC level and each application has its own application master key settings. In both cases, the size of the master key settings is 1 byte. These settings define if the master key is changeable and if the master key settings are changeable. In addition, the master key settings indicate if an authentication is required to perform certain operations such as getting the master key settings and creating applications and files. This is achieved by toggling bits on the master key settings byte. The MF3ICD40 specification [7, sec. 4.3.2] provides more information about the master key settings and about which commands will require a preceding authentication when bits are toggled.

The length of the keys, when interacting with the card, is 8 bytes for DES, 16 bytes for 2K3DES, 24 bytes for 3K3DES, and 16 bytes for AES. Triple DES keys use the least significant bit of each byte as a parity bit, which means that the actual size of the keys is respectively 56, 112 and 168 bits for DES, 2K3DES and 3K3DES. DESFire EV1 ignores the parity bits when handling Triple DES ciphers and these can be used for key versioning. It is advisable to set the parity bits to a default value, if not used for key versioning, because the PICC will reveal them with a `GetVersion` command. This would reduce the strength of the key if the bits were really used as parity bits. The key version is only one byte and for Triple DES it is taken from the first key. If using AES, there is an extra byte specifically for the purpose of key versioning.

Internally, DESFire EV1 handles both DES and 2K3DES keys as 16-byte keys. When the first half of the key is equal to the second half of the key, the key type is considered to be DES. When the first half of the key is different from the second half of the key, the key type is considered to be 2K3DES. For example, $\texttt{20 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00}_{\texttt{H}}$ is a DES key and $\texttt{20 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00}_{\texttt{H}}$ is a 2K3DES key. Note that a key such as $\texttt{00 00 00 00 00 00 00 02 00 00 00 00 00 00 01 03}_{\texttt{H}}$ is a DES key because the key values only differ in the parity bits, i.e. key version. In the example, the version of the key would be $\texttt{00}_{\texttt{H}}$ and clearing the parity bits results in the key $\texttt{00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 02}_{\texttt{H}}$. In the authentication, differences in the parity bits have no effect in the access as failure.

The cipher used by an application is decided during its creation together with the number of keys. Three possibilities exist for this option: DES+2K3DES, 3K3DES and AES. The type of cipher cannot be changed for the application later on. The default secret keys, upon application creation, are 16 bytes set to zero for DES+2K3DES and AES and 24 bytes set to zero for 3K3DES. Since DES and 2K3DES are grouped, it is possible to have DES and 2K3DES keys mixed within the same application and it is possible to change a key from DES to 2K3DES and vice versa. When using DES and 2K3DES as the application cipher, the default key is 16 bytes set to zero, which means it is a DES key. The cipher of the PICC master key can be altered during a key change by tweaking the key number.

To change a key, the PCD sends the message $C4 \| keyNo \| ciphertext$ to the PICC. $C4$ is the command code. $keyNo$ is the number of the key to be

keyNo	number of the key to be changed
new key	the new key
old key	the old key (only required if changing different keys)

```
if changing the PICC master key then
    if type of new key is 3K3DES then
        keyNo ← $40_h$
    else if type of new key is AES then
        keyNo ← $80_h$
    end
else if authentication key number $\neq$ number of key to change then
    temp ← new key $\oplus$ old key (concatenate multiple copies of old key if
    necessary)
end
if new key type is AES then
    plaintext ← temp + key version
else
    plaintext ← temp
end
switch type of key used for authentication do
    case DES or 2K3DES
        compute the CRC16 of plaintext
        append the resulting CRC16 to plaintext
        if authentication key number $\neq$ number of key to change then
            compute the CRC16 of new key
            append the resulting CRC16 to plaintext
        end
        ciphertext ← decryption of plaintext in CBC send mode
    endsw
    case 3K3DES or AES
        compute the CRC32 of cmd||keyNo||plaintext
        append the resulting CRC32 to plaintext
        if authentication key number $\neq$ number of key to change then
            compute the CRC32 of new key
            append the resulting CRC32 to plaintext
        end
        ciphertext ← encryption of plaintext in CBC mode using global IV
    endsw
endsw
```

DES or 2K3DES: $D_{CBC_{send}}(\mathsf{new} \oplus \mathsf{old} \| CRC16(\mathsf{new} \oplus \mathsf{old}) \| CRC16(\mathsf{new}))$
3K3DES or AES: $E_{CBC}(\mathsf{new} \oplus \mathsf{old} \| CRC32(\mathsf{cmd} \| \mathsf{keyNo} \| \mathsf{new} \oplus \mathsf{old}) \| CRC32(\mathsf{new}))$

Algorithm 1.1: Computation of the ciphertext for the change key command.

changed and contains a value between `00` and `0D`. *ciphertext* contains information about the new key and its generation is presented in Algorithm 1.1. When changing the PICC master key, the *keyNo* is tweaked according with the cipher used by the new key. When the new key type is 3K3DES, the key number is set to $\texttt{40}_\texttt{H}$, and when the new key type is AES, the key number is set to $\texttt{80}_\texttt{H}$. For both DES and 2K3DES keys, the key number is set to $\texttt{00}_\texttt{H}$. For applications, the tweak is not required because the cipher chosen when the application is created is fixed.

1.3.4 Data transmission

The three communication modes supported by DESFire EV1 are plain, mac'ed and enciphered. Which communication mode is used during an operation depends on the file settings, if dealing with files, and on the command itself. A rule of thumb is that the plain communication mode is used for all commands, unless stated otherwise.

Each file is associated with its own file settings. The file settings indicate the file size for data files, the record size, boundaries and current number of records for record files, and the boundary values and state of the limited credit option for value files. Additionally, the file settings include the communication settings and the access rights for the file. The communication settings point out the communication mode to use when operating on this particular file. The access rights state whether a preceding authentication is required to interact with the file or not and, if so, which key number the PCD is to authenticate against.

1.3.4.1 Communication settings

The different communication settings are applied to the file-level commands `ReadData`, `WriteData`, `GetValue`, `Credit`, `Debit`, `WriteRecord`, `ReadRecords` and `LimitedCredit`. The application-level command `ChangeFileSettings` uses the enciphered communication mode and requires the preceding authentication to be done with the CAR key of the related file. If the CAR key of that file is set to free access, then the data is exchanged in cleartext. The commands `ChangeKeySettings` and `GetCardUID` use the enciphered communication mode. The commands `Authenticate` and `ChangeKey` use their own security mechanisms, which are detail in Section 1.3.2 and in Section 1.3.3 respectively. The remaining commands use the plain communication mode but some of these commands may require a preceding authentication to succeed depending on the PICC master key settings or on the application master key settings, and on the command itself.

A native DESFire EV1 command can be divided in a command code, headers and the data to be secured. The command code is one byte stating which operation is under action. The headers are pieces of data which are part of the payload of the command but which are not to be encrypted. These include the 1-byte key number referencing secret keys, and the 3-byte offset and 3-byte length fields when writing data to files. The data to be secured is the piece of data which is to be stored on the card. A native DESFire EV1 response can be divided in a status code and the data to be secured. The status code indicates whether the command is successful and the error code if it failed, and the data to be secured is the piece of data received from the card.

When calculating a cyclic redundancy check or message authentication code, or when ciphering data, it is necessary to know on which piece of data to perform the operation. The CRC16 and the CBC-MAC are calculated only over the data to be secured for both commands and responses. The CRC32 and the CMAC are calculated over the command code, headers and data for commands, and over the status code and data, if any, for responses. While a native response is presented as the status code followed by the data, the CRC32 and CMAC calculation is done over the data followed by the status code, i.e. the status code and the data swap for the computation. The encryption and decryption takes place only over the data to be secured and the CRC, leaving out the command code and headers and the response status code.

The objective of the communication modes is to protect the data exchanged between the two parties. In other words, while both the command and respective response are interlinked, the security is applied to the direction carrying the data. If writing data into the card, then the command is protected. If reading data from the card, then the response is protected. This is how DES and 2K3DES operate. For 3K3DES and for AES the system is different because all the operations are chained in order to improve the overall security of the system.

The chaining occurring for 3K3DES and for AES relies on a global initialization vector, an IV which is shared between the commands and responses. This IV is read and subsequently updated when computing CMACs and when encrypting and decrypting data. If authenticated, the IV is still kept up to date even when the plain communication mode is used, which means this chaining is applied to nearly all commands. The exception are commands that cause the authentication state to be changed, such as a new authentication, or that cause the authentication state to be lost, such as an application selection operation.

In the plain communication mode, the data is transmitted between the

PCD and PICC in cleartext for DES and for 2K3DES. This is also the behavior for 3K3DES and for AES if not authenticated. If authenticated then the CMAC is calculated over the commands and over the responses enabling all the operations to be chained. For commands, the CMAC is not appended but it is still calculated by the PCD in order to update the global IV. For responses, the CMAC is calculated and appended to the data by the PICC.

In the mac'ed communication mode, a CBC-MAC or CMAC is appended to the data. The message authentication code is appended to the command if modifying data on the card and is appended to the response if reading data from the card. When using 3K3DES or AES, the CMAC is always appended to the response.

In the enciphered communication mode, a CRC is appended to the data for integrity, and the payload—data to be secured and the CRC—is encrypted for confidentiality. For DES and for 2K3DES, the CRC16 is calculated over the data to be secured. For 3K3DES and for AES, the CRC32 is calculated over the command code, headers and data to be secured in the case of commands, or over the data to be secured and the status code in the case of responses. The encryption is done according with Table 1.3.

1.3.4.2 Access rights

Access rights are used to control the access to files. Each file has four access rights associated with it: read (R), write (W), read&write (RW) and change access rights (CAR). Each of these access rights is coded in 4 bits and references one of the keys associated with the file. A file can have between 1 and 14 keys associated with it and this value is set when the file is created. To reference a key, the access right will hold a value between `0` and `D`. The key referenced is required to exist. There are two special values, `E` and `F`, to respectively indicate free access and deny access. Free access means that access is always granted to the linked access right, with or without a preceding authentication. Deny access means that access is always denied to the linked access right, irrespective of the authentication state. Table 1.5 lists the access rights and the commands to which they apply. These are all file-level commands, with the exception of the application-level command `ChangeFileSettings`.

For a command to succeed, one of the associated access rights must be satisfied. This means that a single positive acknowledgment suffices, independently of the amount of access rights associated with a given command. An access right is satisfied when a preceding successful authentication takes place using the referenced key. Alternatively, access is always granted when one of the access rights associated with the command is set to $\mathtt{E}_\mathtt{H}$.

Command	**Access rights**			
	R	**W**	**RW**	**CAR**
`ChangeFileSettings`				✗
`GetValue`	✗	✗	✗	
`Debit`	✗	✗	✗	
`LimitedCredit`		✗	✗	
`Credit`			✗	
`ReadData`	✗		✗	
`WriteData`		✗	✗	
`ReadRecords`	✗		✗	
`WriteRecords`		✗	✗	
`ClearRecordFile`			✗	

Table 1.5: DESFire EV1 access rights associated with commands.

If one of the access rights associated with a command references the key number used to reach an authenticated state, then the communication takes place according with the communication settings for that file. Otherwise, one of those access rights may contain the free access value `E`, in which case communication is done in plain mode, ignoring the communication settings of the file. In this case, the authentication state is irrelevant. If neither of the access rights associated with the command references the authenticated key number or contains the free access value `E`, the command fails with an authentication error status code.

1.4 Trace

The goal of the following traces is to illustrate the differences between the communication modes and between DES/2K3DES and 3K3DES/AES ciphers in DESFire EV1. The traces are generated by a sample application created for this purpose that uses the library developed during this thesis and presented in Appendix 3. The communication between the reader and card is shown in command–response pairs with the native DESFire EV1 commands and responses wrapped in ISO/IEC 7816-4 APDUs. Only DES and AES are used in the example, but the same line of thought used for DES applies to 2K3DES, and the same line of thought used for AES applies to 3K3DES. The flowchart of the sample application showing the commands sent to the card is depicted in Figure 1.4.

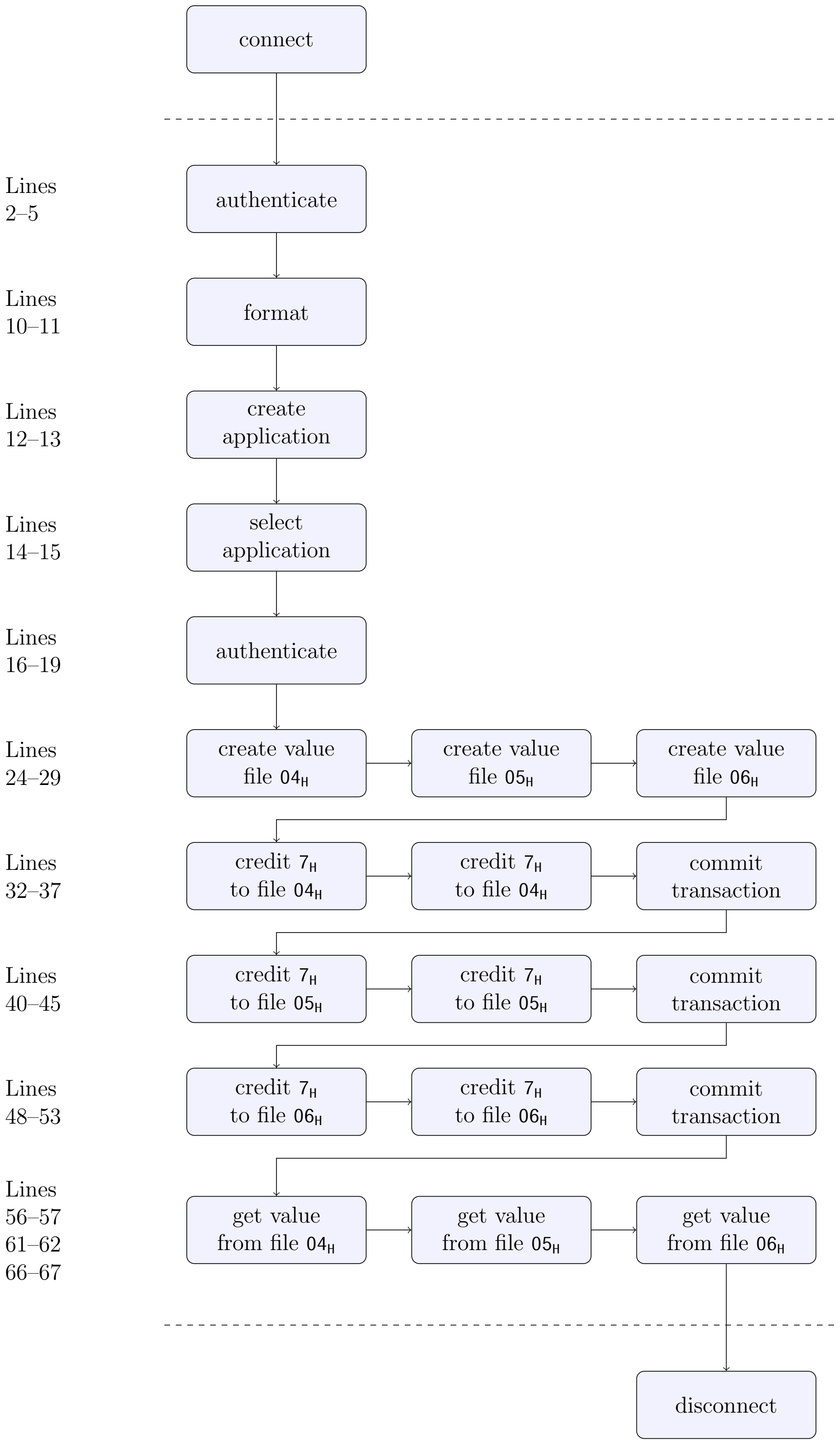


Figure 1.4: Flowchart of the sample application that generates the traces showing the commands sent to the card.

The sample application authenticates at PICC-level and formats the card, which deletes all the existing applications and frees the corresponding memory, and creates a new card application setting its cipher to either DES or AES. This new application is selected and another authentication takes place but at the application-level instead of PICC-level. Three value files are then created, one for each of the three possible communication settings—plain, mac'ed and enciphered. A value file consists of a four-digit counter, which is in this example set to the initial value of $\mathtt{32}_{\mathtt{H}}$ (50_{10}). A value of $\mathtt{7}_{\mathtt{H}}$ is credited twice in each file and the new value $\mathtt{40}_{\mathtt{H}}$ (64_{10}) is retrieved from the card.

```
PC/SC card in SCL011G Contactless Reader [SCL01x Contactless Reader]
    (21161044200765) 00 00, protocol T=1, state OK
>> 90 aa 00 00 01 00 00 (AUTHENTICATE_AES)
<< c9 c1 a7 12 57 e5 2c b6 d7 c8 9c ea 9a 73 c1
    17 91 af (ADDITIONAL_FRAME)
>> 90 af 00 00 20 02 f4 cc 61 fb 8a b1 33 2c 48 87 cf 44 c5 bf c2 57 41 a7
    b1 8a 24 8b 14 0f 39 08 a4 96 35 c3 d6 00 (MORE)
<< 18 e9 ba c5 7f 99 c1 3a d8 ba 5f d6 de 96 90 ce 91 00 (OPERATION_OK)
The random A is 76 69 06 3b d7 51 01 a8 0a 5a b8 35 2b 23 4d 5a
The random B is ad 2c a4 85 6d 7d f5 73 ae 87 0e 7f 07 6a 3c cc
The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
The session key is 76 69 06 3b ad 2c a4 85 2b 23 4d 5a 07 6a 3c cc
>> 90 fc 00 00 00 (FORMAT_PICC)
<< fd 21 05 eb 70 fa e9 62 91 00 (OPERATION_OK)
>> 90 ca 00 00 05 01 02 03 0f 05 00 (CREATE_APPLICATION)
<< 2a fd d4 17 a4 21 34 73 91 00 (OPERATION_OK)
>> 90 5a 00 00 03 01 02 03 00 (SELECT_APPLICATION)
<< 91 00 (OPERATION_OK)
>> 90 0a 00 00 01 03 00 (AUTHENTICATE_DES_2K3DES)
<< da eb 40 1d c9 49 56 6a 91 af (ADDITIONAL_FRAME)
>> 90 af 00 00 10 ac d7 f0 80 f6 18 97 70 17 f3 74 76 75 8c e7
    54 00 (MORE)
<< ca 92 61 78 15 31 b2 1e 91 00 (OPERATION_OK)
The random A is c5 a0 5c 2c 39 4c 91 42
The random B is d0 04 8c 5e 1a 2f 4b f0
The secret key is 00 00 00 00 00 00 00 00
The session key is c5 a0 5c 2c d0 04 8c 5e
>> 90 cc 00 00 11 04 00 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
    CREATE_VALUE_FILE)
<< 91 00 (OPERATION_OK)
>> 90 cc 00 00 11 05 01 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
    CREATE_VALUE_FILE)
<< 91 00 (OPERATION_OK)
>> 90 cc 00 00 11 06 03 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
    CREATE_VALUE_FILE)
<< 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
```

```
<< 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
>> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
<< 91 00 (OPERATION_OK)
>> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
<< 91 00 (OPERATION_OK)
>> 90 c7 00 00 00 (COMMIT_TRANSACTION)
<< 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
<< 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
>> 90 0c 00 00 09 05 07 00 00 00 e1 f6 48 e4 00 (CREDIT)
<< 91 00 (OPERATION_OK)
>> 90 0c 00 00 09 05 07 00 00 00 e1 f6 48 e4 00 (CREDIT)
<< 91 00 (OPERATION_OK)
>> 90 c7 00 00 00 (COMMIT_TRANSACTION)
<< 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
<< 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
>> 90 0c 00 00 09 06 5c ba af d0 96 5c d3 fc 00 (CREDIT)
<< 91 00 (OPERATION_OK)
>> 90 0c 00 00 09 06 5c ba af d0 96 5c d3 fc 00 (CREDIT)
<< 91 00 (OPERATION_OK)
>> 90 c7 00 00 00 (COMMIT_TRANSACTION)
<< 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
<< 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
>> 90 6c 00 00 01 04 00 (GET_VALUE)
<< 40 00 00 00 91 00 (OPERATION_OK)
The stored value (fileNo=4, cs=0) is 64
>> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
<< 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
>> 90 6c 00 00 01 05 00 (GET_VALUE)
<< 40 00 00 00 24 3a fa 5d 91 00 (OPERATION_OK)
The stored value (fileNo=5, cs=1) is 64
>> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
<< 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
>> 90 6c 00 00 01 06 00 (GET_VALUE)
<< 93 a9 4b 99 61 fd 21 68 91 00 (OPERATION_OK)
The stored value (fileNo=6, cs=3) is 64
success.
```

Listing 1.1: Trace with DES.

```
PC/SC card in SCL011G Contactless Reader [SCL01x Contactless Reader]
    (21161044200765) 00 00, protocol T=1, state OK
>> 90 aa 00 00 01 00 00 (AUTHENTICATE_AES)
<< 48 2f 40 ad eb f2 47 a6 e6 e3 fe fe 83 06 0c
    07 91 af (ADDITIONAL_FRAME)
>> 90 af 00 00 20 91 89 ac dc 04 37 67 fa 7d 25 ef 5f b3 ce 68 9d a7 cc 9e
    a8 a7 5b 2a 69 73 9c f0 ab 64 f0 8d 92 00 (MORE)
```

```
<< 88 30 a2 33 db b8 d1 16 1d 28 fa 08 af f6 3e e4 91 00 (OPERATION_OK)
The random A is 95 6b 22 dc 89 f3 ae 21 ab 3c 5b d1 97 11 a3 e1
The random B is 14 43 ba 75 6c 21 84 5b 4c 30 a7 83 d0 d2 1b 8c
The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
The session key is 95 6b 22 dc 14 43 ba 75 97 11 a3 e1 d0 d2 1b 8c
>> 90 fc 00 00 00 (FORMAT_PICC)
<< 66 75 82 d7 7b 34 fc 64 91 00 (OPERATION_OK)
>> 90 ca 00 00 05 01 02 03 0f 85 00 (CREATE_APPLICATION)
<< d9 6a c1 e7 7b 7b 43 76 91 00 (OPERATION_OK)
>> 90 5a 00 00 03 01 02 03 00 (SELECT_APPLICATION)
<< 91 00 (OPERATION_OK)
>> 90 aa 00 00 01 03 00 (AUTHENTICATE_AES)
<< 1f 56 4c 74 42 d0 d6 81 76 5a 29 92 7c d6 f6
    a4 91 af (ADDITIONAL_FRAME)
>> 90 af 00 00 20 76 1b fe 66 9c 55 d9 04 7a 5f ec c2 a8 68 02 8b 07 0b ec
    22 d2 ab d2 3b b6 87 63 bc e7 6f 06 f9 00 (MORE)
<< 4f 62 62 2d f0 e8 a5 aa 97 46 22 5c 7e d2 ec 1f 91 00 (OPERATION_OK)
The random A is ab df 1b 16 60 7d 5c cd fe 74 97 35 c2 5e bf a4
The random B is 0f a9 a1 2c 31 4f 93 e4 85 8a 0c e7 b2 80 f9 a7
The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
The session key is ab df 1b 16 0f a9 a1 2c c2 5e bf a4 b2 80 f9 a7
>> 90 cc 00 00 11 04 00 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
    CREATE_VALUE_FILE)
<< 38 71 1c 80 dd b4 c9 99 91 00 (OPERATION_OK)
>> 90 cc 00 00 11 05 01 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
    CREATE_VALUE_FILE)
<< 93 4f 97 85 4d 56 5c 6d 91 00 (OPERATION_OK)
>> 90 cc 00 00 11 06 03 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
    CREATE_VALUE_FILE)
<< 6e d7 80 b7 b7 58 9f fe 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
<< 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 63 87 c3 00 59 1d 6d
    00 91 00 (OPERATION_OK)
>> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
<< 20 bf 20 a0 6a 48 1b eb 91 00 (OPERATION_OK)
>> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
<< a4 d1 20 40 48 42 cd 4b 91 00 (OPERATION_OK)
>> 90 c7 00 00 00 (COMMIT_TRANSACTION)
<< be bf a5 86 0d 5a f3 2c 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
<< 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 5b 1f 61 37 d7 37 f2
    f9 91 00 (OPERATION_OK)
>> 90 0c 00 00 0d 05 07 00 00 00 1b b5 e6 91 77 50 d2 ca 00 (CREDIT)
<< 0c b6 7f a8 12 69 62 4f 91 00 (OPERATION_OK)
>> 90 0c 00 00 0d 05 07 00 00 00 7b 36 d6 fe f0 66 15 7c 00 (CREDIT)
<< 62 50 7a cc f4 15 54 0d 91 00 (OPERATION_OK)
>> 90 c7 00 00 00 (COMMIT_TRANSACTION)
<< 13 7a af 32 5d e5 a3 38 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
```

```
<< 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 f4 f2 44 99 12 58 78
    e3 91 00 (OPERATION_OK)
>> 90 0c 00 00 11 06 c7 12 75 ba 6b 57 7f ec 92 91 3d 7c ef 4c 1a
    27 00 (CREDIT)
<< c7 dd 3f 55 94 e3 b8 25 91 00 (OPERATION_OK)
>> 90 0c 00 00 11 06 3c 42 75 b5 c9 7e 08 94 c0 88 17 a1 c0 c2 f0
    29 00 (CREDIT)
<< d8 1c 31 e4 72 7c 57 fb 91 00 (OPERATION_OK)
>> 90 c7 00 00 00 (COMMIT_TRANSACTION)
<< e2 ba 9a e7 7d 63 aa 77 91 00 (OPERATION_OK)
>> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
<< 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 f1 0d 4e 2d 33 31 2e
    3d 91 00 (OPERATION_OK)
>> 90 6c 00 00 01 04 00 (GET_VALUE)
<< 40 00 00 00 c6 ed 8f 47 49 4b 83 0d 91 00 (OPERATION_OK)
The stored value (fileNo=4, cs=0) is 64
>> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
<< 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 ac 3e bc 34 09 c5 de
    89 91 00 (OPERATION_OK)
>> 90 6c 00 00 01 05 00 (GET_VALUE)
<< 40 00 00 00 81 b2 95 31 ac bf d9 bb 91 00 (OPERATION_OK)
The stored value (fileNo=5, cs=1) is 64
>> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
<< 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 67 9e b6 25 d4 3f c9
    4c 91 00 (OPERATION_OK)
>> 90 6c 00 00 01 06 00 (GET_VALUE)
<< 99 ff 1c 08 9f 2b 33 8a d4 67 d0 94 74 3d 08 2e 91 00 (OPERATION_OK)
The stored value (fileNo=6, cs=3) is 64
success.
```

Listing 1.2: Trace with AES.

```
PC/SC card in SCL011G Contactless Reader [SCL01x Contactless Reader]
    (21161044200765) 00 00, protocol T=1, state OK
>> 90 aa 00 00 01 00 00 (AUTHENTICATE_AES)
<< ec 42 de 2d c8 0a 67 4b 94 9e 52 e9 ad 6c 69
    ba 91 af (ADDITIONAL_FRAME)
>> 90 af 00 00 20 b3 b0 a5 26 49 5c 97 72 a6 9b 88 cb c9 f7 b3 aa 2c f9 ba
    08 fd c6 86 99 05 84 64 73 8a 55 77 26 00 (MORE)
<< 72 8c 5c 09 6f f8 ba f8 de 8a 00 99 b2 27 2b ec 91 00 (OPERATION_OK)
The random A is 4c a1 76 1b c4 c9 b5 5d ad ee 29 09 17 d7 a6 4f
The random B is df a3 28 c7 3e 68 e5 88 99 a5 3a 65 03 1a 80 b4
The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
The session key is 4c a1 76 1b df a3 28 c7 17 d7 a6 4f 03 1a 80 b4
>> 90 fc 00 00 00 (FORMAT_PICC)
<< c8 18 0c 83 ae f2 8c e6 91 00 (OPERATION_OK)
>> 90 ca 00 00 05 01 02 03 0f 85 00 (CREATE_APPLICATION)
<< 43 c3 2e 35 64 5d d5 bc 91 00 (OPERATION_OK)
>> 90 5a 00 00 03 01 02 03 00 (SELECT_APPLICATION)
```

```
15  << 91 00 (OPERATION_OK)
16
17
18
19                  // No authentication
20                  // inside the application.
21
22
23
24  >> 90 cc 00 00 11 04 00 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
        CREATE_VALUE_FILE)
25  << 91 00 (OPERATION_OK)
26  >> 90 cc 00 00 11 05 01 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
        CREATE_VALUE_FILE)
27  << 91 00 (OPERATION_OK)
28  >> 90 cc 00 00 11 06 03 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
        CREATE_VALUE_FILE)
29  << 91 00 (OPERATION_OK)
30  >> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
31  << 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
```

Listing 1.3: Trace with AES and no authentication at the application-level.

The same piece of code is used for the three traces with minor differences. The security of the operations, after selecting the application, is based on CBC-MAC and DES for the first trace, and it is based on CMAC and AES for the second and third traces. For the third trace, however, there is no authentication inside the application, which results in the last steps, `Credit` and `GetValue`, not being executed. In this case, the application terminates before the first `Credit` because it verifies the settings of the target file, in lines 30–31, and realizes that it does not have the necessary permissions. The traces are presented in Listing 1.1, Listing 1.2 and Listing 1.3.

AES is always used for the PICC-level authentication in lines 2–5. The application-level authentication uses DES or AES, according with the parameters used on application creation. The `CreateApplication` command in line 12 contains either $\texttt{05}_{\texttt{H}}$ or $\texttt{85}_{\texttt{H}}$ as the penultimate byte. The first digit is associated with the cipher used by the application, which is DES for the first case and AES for the second case. The second digit is the number of keys, which is five in this example.

The three value files are created with the same access rights but with different communication settings and the initial value $\texttt{32}_{\texttt{H}}$. The access rights, explained in Section 1.3.4.2, control the access to the file. The communication settings, explained in Section 1.3.4.1, define the level of security for the communication between the reader and the card. The communication settings for the first, second and third value files are respectively plain, mac'ed

File	**FileNo**	**CommSett**	**ARs**
Value file 1	04_H	00_H–plain	$30\ 00_H$
Value file 2	05_H	01_H–mac'ed	$30\ 00_H$
Value file 3	06_H	03_H–enciphered	$30\ 00_H$

Table 1.6: Settings of the value files.

and enciphered. The communication settings are represented by the seventh byte in lines 24, 26 and 28, and the access rights are represented by the eigth and ninth bytes of the same lines. The settings for each value file are summed up in Table 1.6.

The access rights of the value files are $30\ 00_H$. This indicates that an authentication with key number 3_H grants RW access and that an authentication with key number 0_H grants CAR, R and W access. RW access allows the use of `Credit` and `GetValue`. R and W also grant access to `GetValue`. The first two traces are complete because the authentication done inside the card application is against the third key, granting the program access to the required commands. However, the third trace does not authenticate inside the card application; hence the `Credit` and `GetValue` commands are not executed and the program terminates before reaching the end. This is the reason why Listing 1.3 ends in line 31, just before the `Credit` command.

Command	**Plain**	**Mac'ed**	**Enciphered**
`Credit`	32–35	40–43	48–51
`GetValue`	56–57	61–62	66–67

Table 1.7: Lines of the `Credit` and `GetValue` commands per communication mode.

The `Credit` and the `GetValue` commands are affected by the communication settings of the target file. The `Credit` command writes data into the card and the `GetValue` command reads data from the card. When a secure communication mode is used, it applies to the command APDU for `Credit` and it applies to the response APDU for `GetValue`. The objective is to add the security mechanism to the messages carrying data. Table 1.7 shows the lines where these two commands occur and the respective communication modes.

In Listing 1.1, since the application relies on DES for security, each command-response pair is an independent operation when calculating the CBC-MAC or enciphering data. In Listing 1.2, the application relies on AES instead of DES for security, which reuses the IV between operations making the end operation dependent from the previous one. As long as the preceding authentication is done successfully, a CMAC is appended to all the responses, unless the response data is enciphered. When creating the value files in lines 24–29 of Listing 1.2, the CMAC is appended to the response received from the PICC but in Listing 1.3 the CMAC is not appended. This happens because the authenticated state at PICC-level is lost in lines 14–15, when the application is selected, and in Listing 1.3 there is not a successful authentication inside the application. Without the authenticated state it is not possible to calculate the CMAC, because the cryptographic functions require the session key generated after a successful authentication to operate.

For the mac'ed communication mode, a MAC is appended to the command when crediting and a MAC is appended to the response when retrieving data. For the enciphered communication mode, the body of the command is enciphered when crediting and the body of the response is enciphered when retrieving data. Additionally, if AES is used and a preceding successful authentication took place inside the application, a CMAC is also appended to the responses when crediting. Since with DES the pairs of operations are independent from each other, it is likely that operations such as `Credit` can be successfully replayed by a third party. For example, in Listing 1.1, repeating lines 42–43 after line 43 or lines 50–51 after line 51 appears to be feasible and would result in additional credit being added to the target files. However, in the latter case, with an enciphered command-response pair, the amount being credited is not visible. The application purposely contains `Credit` operations in pairs to illustrate their similarities when using DES and their differences when using AES. For Listing 1.1, the command-response pair in lines 32–33 is exactly the same as in lines 34–35, and likewise when comparing lines 40–41 with lines 42–43 and lines 48–49 with lines 50–51. This does not happen in Listing 1.2, where all the `Credit` command-response pairs are similar, yet with a different CMAC. Equally important, the security mechanisms when relying on the 2K3DES cipher operate the same way as in DES, hence suffering from the same weakness. In conclusion, an attacker relaying messages between the PCD and the PICC and with the ability to inject new ones can take advantage of this protocol flaw, effectively producing a man-in-the-middle attack.

Chapter 2

MIFARE Ultralight C

This appendix provides a study of MIFARE Ultralight C, that was carried after studying MIFARE Ultralight and before studying the more complex MIFARE DESFire EV1. MIFARE Ultralight is quite similar to MIFARE Ultralight C and is not included in the thesis for that reason.

MIFARE Ultralight C (MF0ICU2) is a low-cost memory-based smart card for limited-use applications. For instance, event ticketing, loyalty schemes and public transportation. It uses a page-based memory structure like that of MIFARE Ultralight (MF0ICU1), but it has a larger amount of memory and it comprises additional features such as the 16-bit counter and the 2K3DES authentication mechanism. Ultralight C can be considered an intelligent memory card because it offers an access control system, imposing restrictions on which memory pages can be read and written.

Manipulation of the content of an Ultralight C smart card is done using the `Read` and the `Write` commands. `Read` allows to read user data and memory-mapped security configurations. `Write` allows to update user data and modify the security configurations. In both cases, the access is done one page at a time. Exceptions to this guideline include modification of the locking mechanism and of the OTP bits, where only the bits to be modified are set. Table 2.1 lists the commands available for the manipulation of Ultralight C.

Commands
`Authenticate`, `Read`, `Write`

Table 2.1: List of commands of Ultralight C.

2.1 Memory organization

Ultralight C uses a page-based memory structure. Each of the 48 pages is 4 bytes in size, for a total of 192 bytes of EEPROM memory.

00_H	UID	UID	UID	BCC0	Page 0
01_H	UID	UID	UID	UID	Page 1
02_H	BCC1	internal	LOCK0	LOCK1	Page 2
03_H	OTP	OTP	OTP	OTP	Page 3
04_H	user data	user data	user data	user data	Page 4
...	...	...	...	...	...
27_H	user data	user data	user data	user data	Page 39
28_H	LOCK2	LOCK3			Page 40
29_H	counter	counter			Page 41
$2A_H$	AUTH0				Page 42
$2B_H$	AUTH1				Page 43
$2C_H$	K1/0	K1/1	K1/2	K1/3	Page 44
$2D_H$	K1/4	K1/5	K1/6	K1/7	Page 45
$2E_H$	K2/0	K2/1	K2/2	K2/3	Page 46
$2F_H$	K2/4	K2/5	K2/6	K2/7	Page 47

Figure 2.1: Memory layout of MIFARE Ultralight C.

The memory layout of Ultralight C is presented in Figure 2.1. The first nine bytes of memory contain the unique read-only 7-byte serial number (SN/UID) and two block check character bytes (BCC). The last two bytes of page 2 contain the lock mechanism for pages 3–15. Page 3 contains the one time programmable (OTP) bits. Pages 4–39 are the user pages, that is, the pages available for the application to store data. This means that 144-bytes of application data can be stored on the card. The first two bytes of page 40 contain the lock mechanism for pages 16–47. The first two bytes of page 41 contain the 16-bit one-way counter. The first byte of page 42 and the first byte of page 43 store the authentication configuration, used to restrict access to pages. Pages 44–47 contain the 2K3DES secret key.

2.2 Security

The security features provided by Ultralight C include the unique UID, the page locking mechanism, the OTP bits, the one-way counter and the authentication configuration. The authentication configuration, allied with the 2K3DES authentication mechanism, enables access restrictions to pages.

The unique 7-byte UID of Ultralight C is programmed after production by the IC manufacturer. These seven bytes, along with the two BCC bytes, are write-protected to prevent later modification. The BCC calculation follows ISO/IEC 14443-3 and is defined as $CT \oplus SN0 \oplus SN1 \oplus SN2$ for BCC0 and as $SN3 \oplus SN4 \oplus SN5 \oplus SN6$ for BCC1. CT stands for cascade tag byte and is defined as 88_H.

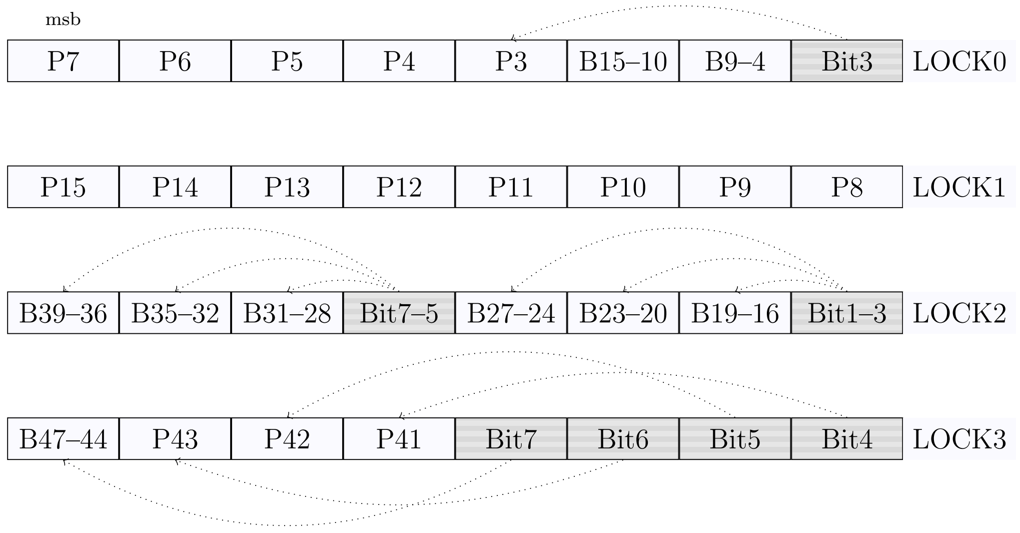


Figure 2.2: Layout of the lock bytes of MIFARE Ultralight C.

The page locking mechanism enables the write-protection of pages. However, it does not prevent pages from being read. The locking mechanism is 4 bytes in size. LOCK0–1 are the last two bytes of page 2 and LOCK2–3 are the first two bytes of page 40. The layout of the lock bytes of Ultralight C is presented in Figure 2.2. It is possible to lock individual pages, blocks of pages and individual lock bits. These are represented respectively as PX, BX−Y and Bit$Z$. Pages 3–15 and pages 41–43 can be individually locked. Pages 4–39, that is, the user data pages, can be locked in blocks. Pages 44–47, which contain the 2K3DES secret key and are not readable, can also be locked as a single block. Some individual lock bits can be set to prevent other lock bits from being accidentally or intentionally set. For example, if the most significant bit of LOCK3 is set, the secret key stored in pages 44–47 is frozen. To prevent this bit from being set, the fourth bit of LOCK3 can be set. If this is done while the eighth bit is cleared, then the secret key will always be changeable.[1]

Page 3 contains 32 OTP bits. These bits are cleared by default and can be individually set. Each of these bits can only be set once.

The 16-bit one-way counter, in page 41, is used to keep track of an always incrementing value. The default value is $\texttt{0000}_{\texttt{H}}$.

The authentication configuration bytes, AUTH0 and AUTH1, can be used to restrict either write or read and write access to pages. AUTH0 contains a page number, in hexadecimal, marking the page from which the settings in AUTH1 are applied. The effect extends from that page to the end of the memory. If AUTH0 is set to $\texttt{30}_{\texttt{H}}$, then there are no restricted pages because there are only $\texttt{2F}_{\texttt{H}}$ pages in total. The configuration in AUTH1 is either $\texttt{01}_{\texttt{H}}$ or $\texttt{00}_{\texttt{H}}$, for respectively restricted write access and restricted read and write access. A successful authentication enables full access to restricted pages.

The 2K3DES authentication protocol of Ultralight C ensures that both parties share a common 16-byte secret key. The authentication protocol, depicted in Figure 2.3, is composed by the following steps:

1. The PCD sends an authentication request to the PICC. The APDU wrapped authentication request command is $\texttt{FF EF 00 00 02 1A 00}_{\texttt{H}}$.

2. The PICC receives the authentication command and generates and encrypts an 8-byte random number *RndB*. The resulting ciphertext is sent to the PCD as a response.

3. The PCD receives and decrypts the response obtaining the random number *RndB* generated by the PICC. It then rotates *RndB* one byte to the left yielding $x_2 = RndB'$. The PCD generates its own 8-byte random number *RndA* and concatenates $RndB'$ to it. $RndA \| RndB'$ is encrypted in CBC mode and sent to the PICC.

4. The PICC decrypts the received message, rotates its *RndB* to the left and compares it with the decrypted $RndB'$ sent by the PCD. If the match fails, the PICC returns an error code to the PCD. Otherwise, it rotates *RndA* one byte to the left, yielding $RndA'$ (x_8). $RndA'$ is encrypted and sent to the PCD.

[1]Authentication with the current secret key may be required, depending on the authentication configuration in pages 42–43.

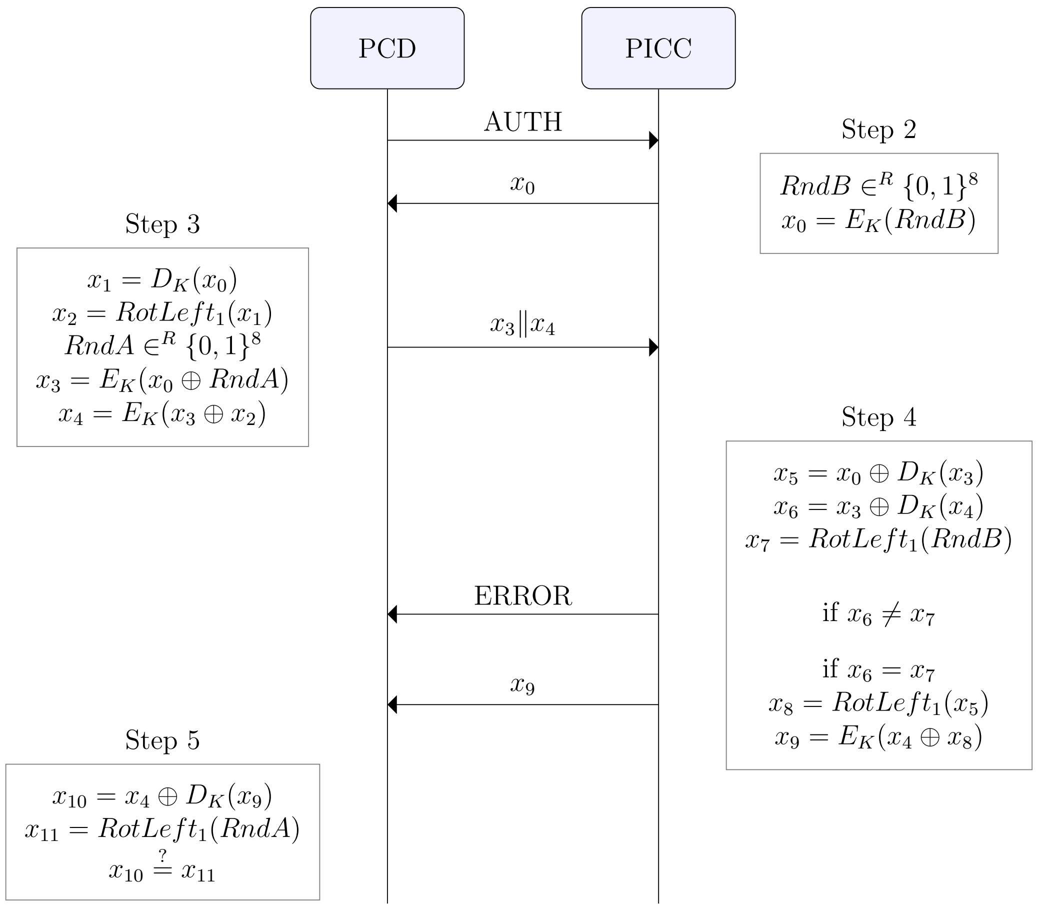


Figure 2.3: 2K3DES authentication protocol for Ultralight C.

5. The PCD decrypts the received message, rotates its *RndA* one byte to the left and compares it with the *RndA′* received from the PICC. If the rotated *RndA* and *RndA′* match, the authentication is successful.

The parity bits of the 2K3DES secret key are ignored by the card. This means that a key with the parity bits set is equal to the same key with the parity bits cleared.

To change the secret key, it is enough to write the new key to pages 44–47. During the authentication protocol, the encryption and the decryption of data is done with the 2K3DES secret key in big-endian form. However, the secret key is stored in pages 44–47 in little-endian form, which is also the adopted form when changing the secret key. The first part of the key is written, in little-endian, to pages 44–45. The second part of the key is written, in little-endian, to pages 46–47. Consider the secret key `A7 A6 A5`

`A4 A3 A2 A1 A0 B7 B6 B5 B4 B3 B2 B1 B0`$_H$. To authenticate, the key is used as is. On the other hand, to change the secret key to this key, it is written as `A0 A1 A2 A3 A4 A5 A6 A7 B0 B1 B2 B3 B4 B5 B6 B7`$_H$. Page 44 stores `A0 A1 A2 A3`$_H$, page 45 stores `A4 A5 A6 A7`$_H$, page 46 store `B0 B1 B2 B3`$_H$ and page 47 stores `B4 B5 B6 B7`$_H$. After changing the secret key, the new key becomes active after reconnecting to the card. This implies that if the secret key is changed and an authentication is attempted before halting the card, the old secret key must be used.

Chapter 3

New NFC Java library

The NFC Java library (nfcjlib) produced in this project eases the development of applications for MIFARE smart cards. The objective is to accelerate the development of applications and to increase their reliability by hiding the complexity of the native commands and cryptographic operations and to offer a less error-prone alternative to manually handle these repetitive tasks. The library is used to manipulate both MIFARE DESFire EV1 and MIFARE Ultralight C.

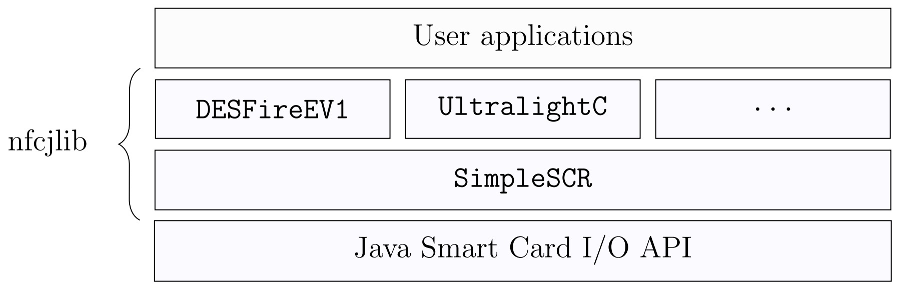


Figure 3.1: The NFC Java library architecture.

The library was designed to be simple and expandable. The ecosystem is composed by three layers: the standard Java Smart Card I/O API, the nfcjlib library itself, and the applications of the end user. The Java Smart Card I/O API communicates with the smart card via a smart card reader and provides functions to nfcjlib to interact with the smart card. Nfcjlib consumes those functions and provides services to the end user. The end user manipulates the smart card through the services provided by nfcjlib. The architecture of the library is presented in Figure 3.1 and is formed of two layers: the reader module and the smart card modules. The reader module interacts with the Java Smart Card I/O API and provides a service to the upper layer to connect, disconnect and transmit APDUs. This upper layer is divided into modules, one for each smart card type. These modules, take `DESFireEV1` for instance, interact with the reader to establish and tear down connections and exchange information with the smart card, and to provide services to the end users on the upper layer.

The evaluation of the implementation relies on JUnit 4 for automated testing.[1] It aims to answer two questions:

Validation Are all the commands implemented for both smart cards?

Verification Are those commands implemented correctly?

To validate the library, the total number of commands is counted for both smart cards and compared with the number of commands implemented. To verify the correct implementation of those commands, a test plan is produced and JUnit 4 is used for the creation of test cases. The evaluation of DESFire EV1 is presented in Section 3.1 and the evaluation of Ultralight C is presented in Section 3.2.

In JUnit 4, both `Assume` and `Assert` methods are heavily used. Assumptions are conditions expected to be met for the test to be considered valid. Assertions are predicates related to the feature under test. For instance, suppose feature X requires a preceding authentication to be successfully tested. In such case, an assumption that a preceding successful authentication took place, before testing the feature, is made. Only then, can assertions regarding feature X take place. When an assertion fails, the test case fails. However, when an assumption fails, the test case succeeds. This happens because the conditions imposed for testing a feature are not met. This subtlety is important for a proper understanding of the test cases. It could be argued that some features may never be tested because the assumptions for those features are never met, and that it is impossible to tell whether a test is successful because the assertions were correct or because the assumptions failed. However, the assumptions are themselves tested. This ensures that if a test suite completes without errors, then all the test cases and respective assertions completed successfully.

3.1 Evaluation of the MDF implementation

The number of commands for the evaluation of the implementation of DESFire EV1 is taken from Table 1.1. There are five security-related commands, ten PICC-level commands, nine application-level commands and eleven data manipulation commands, for a total of 35 commands. Of the 35 commands,

[1]JUnit 4 is a framework to write repeatable tests for Java. See `http://junit.org/`.

only two commands, `SetConfiguration` and `GetDFNames`, are not implemented. This means that 33 out of 35 commands are implemented for DESFire EV1.

The DESFire EV1 smart card used during the evaluation is required to meet certain conditions. The PICC master key is set to 0^{16} and the type of key is AES. The PICC master key settings are set to $\texttt{0F}_\texttt{H}$. The following test plan details the tests taking place. The results of the test cases implemented in JUnit 4 are presented in Table 3.1. All the test cases are done using all the different ciphers available—DES, 2K3DES, 3K3DES and AES.

100/Security Authenticate at PICC-level (key with version bits cleared).

101/Security Authenticate at PICC-level (key with version bits set).

102/Security Authenticate at application-level with key number $\texttt{00}_\texttt{H}$ (key with version bits cleared).

103/Security Authenticate at application-level with key number $\texttt{00}_\texttt{H}$ (key with version bits set).

104/Security Authenticate at application-level with key number different from $\texttt{00}_\texttt{H}$ (key with version bits cleared).

105/Security Authenticate at application-level with key number different from $\texttt{00}_\texttt{H}$ (key with version bits set).

106/Security Change PICC master key: from key with all version bits cleared to key with all version bits set and vice versa.

107/Security Change application master key: from key with all version bits cleared to key with all version bits set and vice versa.

108/Security Change application key different from $\texttt{00}_\texttt{H}$: from key with all version bits cleared to key with all version bits set and vice versa.

109/Security Change the PICC master key using X as version. Verify if the key version returned by the PICC is X.

110/Security Change the PICC master key settings to $\texttt{0F}_\texttt{H}$. Verify if the key settings are correct.

111/Security Change the key settings of an application to $\texttt{0D}_\texttt{H}$. Verify if the key settings are correct.

112/Security Successfully retrieve the manufacturing related data of the card.

113/Security Successfully retrieve the card UID.

114/Security Authenticate and format the card.

115/Security Attempt to format the card without a preceding authentication. The command fails.

116/Security Retrieve the free memory on the card.

200/Application Successfully select the AID `00 00 00`$_{\mathtt{H}}$.

201/Application Attempt to select the AID of a non-existing application. The command fails.

202/Application Create an application.

203/Application Attempt to create an application with an existing AID. The command fails.

204/Application Create an application and then delete it successfully.

205/Application Attempt to delete a non-existing application. The command fails.

206/Application Create file X. Delete file X. File X does not exist after deletion.

207/Application Attempt to delete a non-existent file. The operation fails.

208/Application Create multiple files from 0 to 31 and verify if the amount of file identifiers is correct.

209/Application Create file X. Get and verify the properties of file X.

210/Application Attempt to get the properties of a non-existent file. The operation fails.

211/Application Create a file and successfully modify its properties.

300/SDF Create a standard data file.

301/SDF Create a standard data file and write X with offset Y. Read the entire file and verify if X is written at offset Y.

302/SDF Create a standard data file with size X and write data with size $> X$ to it. The writing fails.

400/BDF Create a backup data file.

401/BDF Create a backup data file with size > 512. To test the correct message separation in frames, write the full file length and read the entire file. Verify if the length and contents read are correct.

500/VF Create a value file with initial value X. A `GetValue` returns X.

501/VF Create a value file with initial value X. Increase the stored value by Y. The new value stored in the file is $X + Y$.

502/VF Create a value file with initial value X. Decrease the stored value by Y and by Z with $Y \neq Z$. The new value stored in the file is $X - Y - Z$.

503/VF Create a value file with initial value X and with `LimitedCredit` enabled. Decrease the stored value by Y. Successfully increase the value by Y using `LimitedCredit`.

504/VF Create a value file with initial value X and maximum value M. Increase the initial value by Y such that $X + Y > M$. The `Credit` command fails.

505/VF Create a value file with initial value X and minimum value M. Decrease the initial value by Y such that $X - Y < M$. The `Debit` command fails.

506/VF Create a value file with initial value X and with `LimitedCredit` enabled. Decrease the stored value by 5. Now increase the value by 1 using `LimitedCredit`. Repeat the previous command after committing. `LimitedCredit` fails. It can only be done once after a transaction involving a `Debit`.

507/VF Create a value file with initial value X and with `LimitedCredit` enabled. Decrease the stored value by Y. Now increase the value by Z using `LimitedCredit`, such that $Z > Y$. `LimitedCredit` fails.

508/VF Create a value file with initial value X and with `LimitedCredit` disabled. Decrease the stored value by Y. Now increase the value by Y using `LimitedCredit`. `LimitedCredit` fails.

600/LRF Create a linear record file.

601/LRF Create a linear record file with record size X and number of records Y. Create Y records. A full read returns $X \times Y$ bytes.

602/LRF Create a linear record file with two records. A read with offset 1 and number of records 0 returns 1 record.

603/LRF Create a linear record file with three records. A read with offset 0 and number of records 2 returns 2 records.

604/LRF Create a linear record file with record size 2. Create a record containing `41 42`$_\mathsf{H}$. A read of the created record returns `41 42`$_\mathsf{H}$.

605/LRF Create a linear record file and write a record. Clear the record file and attempt to read it. Reading fails because there are no records.

700/CRF Create a cyclic record file.

701/CRF Create a cyclic record file with record size 2 and number of records 3. Create $3 - 1$ records with contents `41 42`$_\mathsf{H}$ and `51 52`$_\mathsf{H}$ respectively for the first and for the second records. A full read returns `41 42 51 52`$_\mathsf{H}$.

702/CRF Create a cyclic record file with record size 1 and number of records 3. Create 3 records with contents `1A`$_\mathsf{H}$, `1B`$_\mathsf{H}$ and `1C`$_\mathsf{H}$. A full read returns `1B 1C`$_\mathsf{H}$.

703/CRF Create a cyclic record file with record size 1. An attempt to write a record with size > 1 fails.

800/File Create a value file with initial value X, credit Y and commit the transaction. The value stored on the file is $X + Y$.

801/File Create a value file with initial value X, credit Y and abort the transaction. The value stored on the file is X.

Id/context	**Key operations performed**	**Test for**	**Result**
100/Security	`Authenticate`	success	✔
101/Security	`Authenticate`, `ChangeKey`	success	✔
102/Security	`Authenticate`	success	✔
103/Security	`Authenticate`, `ChangeKey`	success	✔
104/Security	`Authenticate`	success	✔
105/Security	`Authenticate`, `ChangeKey`	success	✔

106/Security	`Authenticate`, `ChangeKey`	success	✔
107/Security	`Authenticate`, `ChangeKey`	success	✔
108/Security	`Authenticate`, `ChangeKey`	success	✔
109/Security	`GetKeyVersion`, `ChangeKey`	success	✔
110/Security	`ChangeKeySettings`, `GetKeySettings`	success	✔
111/Security	`ChangeKeySettings`, `GetKeySettings`	success	✔
112/Security	`GetVersion`	success	✔
113/Security	`GetCardUID`	success	✔
114/Security	`Authenticate`, `FormatPICC`	success	✔
115/Security	`FormatPICC`	failure	✔
116/Security	`FreeMemory`	success	✔
200/Application	`SelectApplication`	success	✔
201/Application	`SelectApplication`	failure	✔
202/Application	`CreateApplication`	success	✔
203/Application	`CreateApplication`	failure	✔
204/Application	`DeleteApplication`	success	✔
205/Application	`DeleteApplication`	failure	✔
206/Application	`DeleteFile`, `GetFileIds`	success	✔
207/Application	`DeleteFile`	failure	✔
208/Application	`GetFileIds`	success	✔
209/Application	`GetFileSettings`	success	✔
210/Application	`GetFileSettings`	failure	✔
211/Application	`ChangeFileSettings`	success	✔
300/SDF	`CreateStdDataFile`	success	✔
301/SDF	`WriteData`, `ReadData`	success	✔
302/SDF	`WriteData`	failure	✔
400/BDF	`CreateBackupDataFile`	success	✔
401/BDF	`WriteData`, `ReadData`	success	✔
500/VF	`CreateValueFile`, `GetValue`	success	✔
501/VF	`Credit`, `GetValue`	success	✔
502/VF	`Debit`, `GetValue`	success	✔
503/VF	`LimitedCredit`, `GetValue`	success	✔
504/VF	`Credit`	failure	✔

505/VF	`Debit`	failure	✔
506/VF	`LimitedCredit`	failure	✔
507/VF	`LimitedCredit`	failure	✔
508/VF	`LimitedCredit`	failure	✔
600/LRF	`CreateLinearRecordFile`	success	✔
601/LRF	`WriteRecord`, `ReadRecords`	success	✔
602/LRF	`WriteRecord`, `ReadRecords`	success	✔
603/LRF	`WriteRecord`, `ReadRecords`	success	✔
604/LRF	`WriteRecord`, `ReadRecords`	success	✔
605/LRF	`ClearRecordFile`, `ReadRecords`	failure	✔
700/CRF	`CreateCyclicRecordFile`	success	✔
701/CRF	`WriteRecord`, `ReadRecords`	success	✔
702/CRF	`WriteRecord`, `ReadRecords`	success	✔
703/CRF	`WriteRecord`	failure	✔
800/File	`CommitTransaction`	success	✔
801/File	`AbortTransaction`	success	✔

Table 3.1: Test case results of nfcjlib for DESFire EV1.

3.2 Evaluation of the MUC implementation

The number of commands, for the evaluation of the implementation of Ultralight C, is taken from Table 2.1. There are three commands, all of them implemented in nfcjlib.

The Ultralight C smart card used during the evaluation is required to meet certain conditions. The secret key is set to 0^{16}, the lock bytes are all set to zero and AUTH0 is greater than or equal to $\texttt{30}_\texttt{H}$. The following test plan details the tests taking place for the evaluation of Ultralight C. The results of the test cases, implemented in JUnit 4, are presented in Table 3.2.

900 Authenticate using the default secret key (0^{16}) with all parity bits cleared. The authentication succeeds.

901 Authenticate using the default secret key with some parity bits set. The authentication succeeds.

902 Change the secret key to X, where X is different from the default secret key. Authenticate using X with all parity bits cleared. The authentication succeeds.

903 Change the secret key to X, where X is different from the default secret key. Authenticate using X with some parity bits set. The authentication succeeds.

904 Authenticate with an incorrect secret key. The authentication fails.

905 Change the secret key. The command succeeds.

906 Update a user page. The commands succeeds.

907 Read a user page. The commands succeeds.

908 Write X into a user page Y. Reading page Y returns X.

909 Attempt to update a non-existent user page. The command fails.

910 Attempt to read a non-existent user page. The command fails.

Id	**Key operations performed**	**Test for**	**Result**
900	`Authenticate`	success	✔
901	`Authenticate`	success	✔
902	`Authenticate`, `ChangeSecretKey`	success	✔
903	`Authenticate`, `ChangeSecretKey`	success	✔
904	`Authenticate`	failure	✔
905	`ChangeSecretKey`	success	✔
906	`Update`	success	✔
907	`Read`	success	✔
908	`Update`, `Read`	success	✔
909	`Update`	failure	✔
910	`Read`	failure	✔

Table 3.2: Test case results of nfcjlib for Ultralight C.

Bibliography

[1] German Federal Office for Information Security. *MIFARE DESFire EV1 MF3ICD81–Certification Report*, July 2011. Certification Report BSI-DSZ-CC-0712-2011.

[2] Kasper, T., von Maurich, I., Oswald, D., and Paar, C. Chameleon: a versatile emulator for contactless smartcards. In *Proceedings of the 13th international conference on Information security and cryptology* (Berlin, Heidelberg, 2011), ICISC'10, Springer-Verlag, pp. 189–206.

[3] n.a. libfreefare. [Website]. `https://code.google.com/p/libfreefare/`. Accessed 23.8.2013.

[4] NXP Semiconductors. *MF3ICDx21_41_81–MIFARE DESFire EV1 contactless multi-application IC–Product short data sheet–Rev. 3.1*, December 2010. `http://www.nxp.com/documents/short_data_sheet/MF3ICDX21_41_81_SDS.pdf`. Accessed 25.8.2013.

[5] NXP Semiconductors. *Security Target Lite–MIFARE DESFire EV1 MF3ICD81–Rev. 1.5*, May 2011. Security Target BSI-DSZ-CC-0712-2011.

[6] Oswald, D., and Paar, C. Breaking mifare desfire mf3icd40: Power analysis and templates in the real world. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, B. Preneel and T. Takagi, Eds., vol. 6917 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 207–222.

[7] Philips Semiconductors. *mifare DESFire: Contactless Multi-Application IC with DES and 3DES Security–MF3 IC D40–Product specification–Rev. 3.1*, April 2004.