

RFID Encryption Project

1) Executive Summary

1.1) Project Overview

This project implements a comprehensive security solution for RFID-based door access control systems, focusing on protecting against unauthorised access and card cloning through advanced encryption and key management strategies. The implementation uses MIFARE Classic 1K cards and incorporates AES (Advanced Encryption Standard) encryption for key management and secure authentication.

1.2) Project Goals

1. Design and implement a secure RFID card authentication system
2. Protect access credentials using AES encryption
3. Implement session-based key management to prevent replay attacks
4. Create a reliable key storage and management system
5. Ensure secure communication between all system components

2) Introduction

Radio Frequency Identification Technology (RFID) is a commonly used technology to detect objects. It mainly uses radio waves or electromagnetic fields to transmit data. The system itself consists of a tiny radio transmitter embedded in a tag, and a reader which is a radio receiver. A tag can also be embedded to objects, such as cards, clothing, possessions, etc. the reader usually continuously transmit radio frequency and when a tag is within the range of detection of the reader, the electromagnetic pulse will eventually trigger the tag and cause it to send digital data, such as the Unique Identifier (UID). In most cases, the UID is the identifier which will be stored in the system's database, hence becoming the identifier of the tagged object.

2.1) Background

Due to its demanding usage in day to day life to track and identify possibly high-value items, it raises security concerns. RFID-tag cloning, an attempt to clone the data from a tag on a new tag, is one attacker method to obtain access to what the purpose of the tag is, such as accessing restricted facilities, bypassing the security barriers in which a possession is kept, or even alter stock management. An attacker needs to have a tag reader to perform the cloning process, which later is used to read the original card to obtain the UID or data. On the other hand, RFID skimming is also a common attacker method to steal the transmitted data during the scanning process between an authorised reader and the original card. For instance, an attacker could attempt to install an illegal card reader device in an ATM machine as seen on figure 2.1, to obtain the card number and PIN.

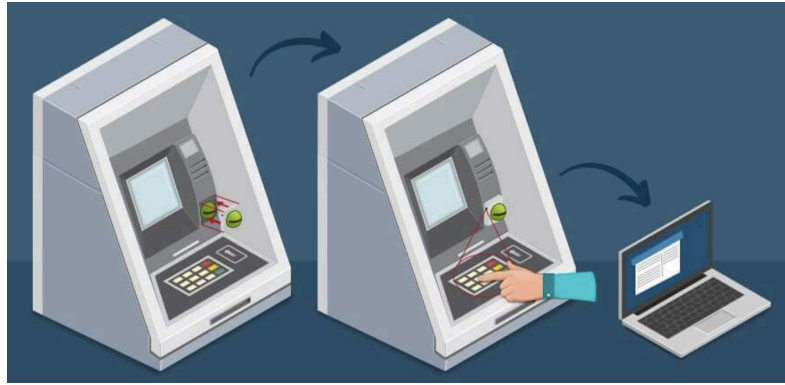


Figure 2.1

2.1.1) Security Approach

Secure key management must address several critical aspects of RFID security. First, it needs to ensure that each card in the system uses unique, cryptographically secure keys that are properly generated and stored. This prevents the systemic vulnerabilities associated with shared or default keys. Second, the management system must implement proper encryption for all stored data, ensuring that even if an attacker gains physical access to a card, the stored information remains secure.

Dynamic session management represents another crucial component of secure key management. By implementing session-based keys that change after each successful authentication, the system can prevent replay attacks and minimise the impact of any compromised keys. This approach ensures that even if an attacker manages to obtain a key, it becomes invalid for future authentication attempts, significantly enhancing the system's security posture. Altogether, Proper key management transforms RFID systems from potentially vulnerable access points into robust security solutions that can reliably protect assets and control access in modern security environments.

3) System Architecture

The system architecture implements a multi-layered approach that integrates hardware components, communication protocols, and security measures. The core system comprises a C++ host application communicating with an Arduino-based RFID reader through serial communication, while maintaining secure data storage in a PostgreSQL database. This architecture ensures secure, reliable interaction between all components while maintaining data integrity and access control.

3.1) C++ and Arduino Integration

The communication between the host C++ application and Arduino is implemented through a robust serial communication protocol. The `SerialPort` class in C++ establishes and manages a serial connection with the Arduino at 9600 baud rate, implementing proper error handling and data validation. The protocol uses a command-response pattern where the C++ application sends formatted commands (e.g., "AUTH,<key>", "WRITE_BLOCK,<blockNumber>,<data>") to the Arduino, which processes these commands and returns structured responses. This bidirectional communication ensures reliable data exchange while maintaining system state consistency.

3.2) MFRC522 Library

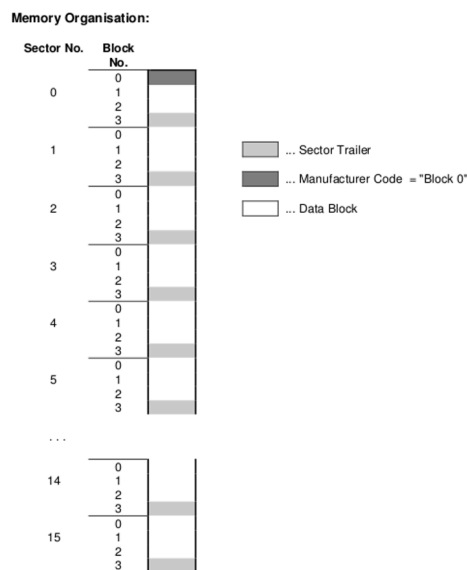
This library manages low-level communication with the RFID reader hardware, handling complex RFID protocols and providing a high-level interface for card operations. Key features used from the library include card presence detection, UID reading, sector authentication, and block read/write operations. The library also enables proper error handling and status monitoring.

3.3) PostgreSQL Database Integration

The PostgreSQL database implements a secure and efficient structure for key storage and management. The database schema is designed with security and performance in mind, utilising appropriate data types and indexing strategies. It comprises several key tables, such as `rfid_keys` and `permitted_access`. The `rfid_keys` table stores essential card data including UIDs and their associated encryption keys. Each card entry maintains a reference to its access key, current custom key and encryption key. The `permitted_access` table manages authorised access credentials which is a list of permitted access keys and **its corresponding UID???**.

3.4) RFID Architecture

In this project, we will use 1K MiFare cards which will have a layout as shown in figure 1.6



Data block is where the data will be written. Sector trailers act as an authentication system, everytime we want to access the data block in a certain sector we need to authenticate through the sector trailer using key A or key B and the sector key. On a new card, the sector key is in default, i.e. 0xFFFFFFFFFFFF. In this project we will modify the sector 1 key, by writing into the sector 1's sector trailer the encryptedCustomKey that we will generate. After the authentication process, we write data (in this case, the generated encryptedAccessKey) into sector 1. In the code implementation the block number that will be written with data is 4 and the sector trailer is 7, since on the implementation, sector 1 consists of Block No. 4 - 7.

4) Security Analysis

4.1) OpenSSL Random Key Generation

OpenSSL's `RAND.h` offers cryptographically secure random number generation, essential for creating robust encryption keys. This library relies on system entropy sources to produce random numbers suitable for cryptographic purposes, making it more secure than traditional random number generators like `rand()`. This library is used to generate keys used on this project, which includes the encryption keys, custom sector keys, and access keys.

4.2) AES (Advanced Encryption Standard)

This project uses the standard encryption algorithm, AES, due to its balance between security strength, resource requirements, and performance which is crucial in RFID applications as RFID cards or tags usually have limited processing power and quick response times are needed (real-time). AES has low memory requirements compared to other encryption algorithms, such as RSA. This project implements AES from scratch to provide deeper understanding on the encryption process and mathematics behind each operation which could emphasise potential vulnerabilities.

AES is a symmetric encryption algorithm since it uses the same key for both encryption and decryption and without the correct key, the content remains secure. It is based on the Rijndael cipher. The process is reversible only with the correct key. AES uses cryptographic keys of 128, 192, or 256 bits. The size will also determine the number of rounds the data undergoes - 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

AES encryption operates on the principle of Substitution-Permutation (SP) networks, which forms the foundation of its security mechanism. At its core, an SP network combines three essential operations: XOR operations, substitutions (where data is replaced using predefined lookup tables), and permutations (where data bits are systematically rearranged). The algorithm processes data in fixed blocks of 128 bits, treating this data as a 4x4 matrix of bytes called the "state". It also used a lookup table also known as Rijndael S-box during its operations. When performing encryption, a forward S-box is used (figure 4.1). Otherwise, during decryption, an inverse S-box is used (figure 4.2).

```
uint8_t sbox[256] = {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};
```

Figure 4.1

```
uint8_t inv_sbox[256] = {
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D
};
```

Figure 4.2

The data is encrypted through several operations and rounds. Different types of operations in AES includes:

4.2.1) Key Expansion

Depending on the size of encryption keys, it will be expanded to the corresponding number of rounds. For instance, this project will use 128-bits keys which will be expanded to 11 round keys. The expansion process transforms the original 16-byte key into a 176-byte key schedule by going through byte substitution using the S-box, rotation, and XOR operations.

4.2.2) XOR Operation

XOR operation takes two arguments, which include the round key and the 16-bytes data. Each byte of the data will XOR-ed with the corresponding byte from the round key. XOR operation has self-inverse property, that is if we XOR a value with the same value twice, we get nacl the original value. When XORing data with a random (in this case the round key), the result appears random to anyone who doesn't know the key.

4.2.3) Subbytes

This operation will use the S-box to substitute each byte of the 16-byte data with the corresponding value in the S-box. In hexadecimal, a byte has two 4 bits value, each ranging from 0 to F. The first 4 bits value corresponds to the row of the S-box, while the next 4-bits value corresponds to the column of the S-box. For instance, a byte value of 0x53 corresponds to the position [5][3] in the S-box.

4.2.4) Shiftrows

In this step, each row in the 4x4 matrix is shifted by a certain position as seen on figure 4.3.

- The first row is not shifted
- The second row is shifted to the left by one position
- The third row is shifted to the left by two position
- The fourth row is shifted to the left by three position

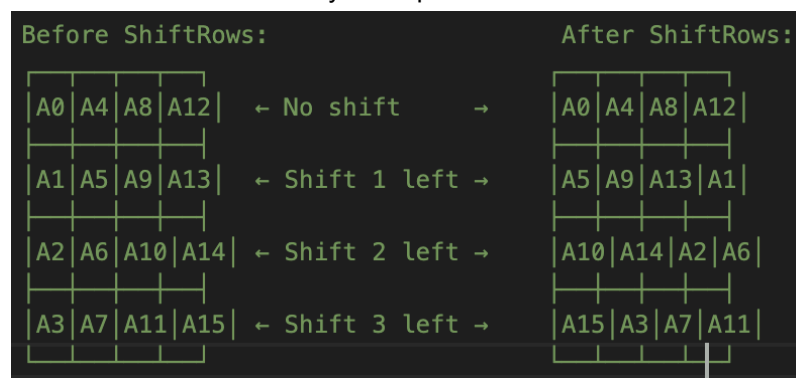


Figure 4.3

The inverse shiftrows will be performed during the decryption process which will undo the shifting by shifting to the right.

4.2.5) Mixcolumns

In this process, each column of the state matrix is viewed as a four-term polynomial and is multiplied by a fixed polynomial in the Galois Field $GF(2^8)$. The transformation involves matrix multiplication, where each column is multiplied by a constant matrix

The inverse of this will be performed during the decryption process. Similar to MixColumns, the operation processes each column independently, but uses different constant multipliers in the transformation matrix. In practice, this means multiplying each column by a fixed matrix [0E, 0B, 0D, 09; 09, 0E, 0B, 0D; 0D, 09, 0E, 0B; 0B, 0D, 09, 0E].

In this step, the algorithm would extract the current round key from the expanded key and perform XOR operation with the state array, which is the array in which the encryption operations are performed.

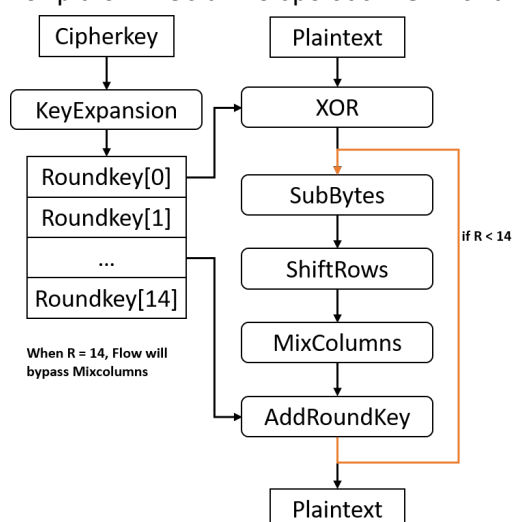


Figure 4.4

The complete process of the encryption algorithm guarantees a cipher text generation that is impossible to be cracked without the encryption key due to its complex operations combination and mathematical calculations.

This project implements a session key in the RFID system which serves as a dynamic security measure by continuously updating the sector keys after each successful authentication. Initially, the system reads the card's UID and retrieves corresponding encryption key in the database and performs a series of authentication. If the authentication succeeds and the session comes to an end, the system generates a new custom key, encrypts it, and rewrite it on the card. This approach significantly enhances security by ensuring that even if an attacker manages to obtain data or key, it becomes invalid after its next legitimate use. This could prevent replay attacks in which the attacker intercept the communication between legitimate reader and tag and attempt to replay it. A session key would require the attacker to constantly interfere the communication after a legitimate session is done.

the attacker is required to brute force attack the encrypted custom key and have a card type that has a changeable UID. For this project, I used a MIFARE card, in which the UID is written during manufacturing and is read-only. Additionally, some newer card types implement additional anti-cloning features. Overall, this approach, combined with encryption, could add an extra security layer on the RFID system.

5) System Implementation

5.1) Initialisation

When a new RFID card is presented to the system, the system identifies the card as new by attempting authentication with the default factory key (0xFFFFFFFF). Upon successful authentication, the system will send the 'DEFAULT_OK' flag to the host which means the card is new since the sector key is still on its default value. Then, the system generates three critical security elements: **Encryption Key**, a unique 128-bit AES encryption key is generated for the card. This key is associated with the card's UID in the database and it never leaves the database and is used for encrypting/decrypting card data. **Custom Key**, a 6-byte custom key is generated for sector authentication. This key is encrypted using the card's encryption key and is written to the sector trailer (block 7) while the decrypted versions are stored in the database. **Access Key**, a 16-byte access key is generated as the authentication credential. This key is encrypted using the card's encryption key. The encrypted access key is written to block 4 while the decrypted access key is stored in the `rfid_keys` and `permitted_access` table.

These custom keys and access keys will be encrypted using the AES algorithm. The encrypted custom key and encrypted access key will be written on the card, however, the system will first authenticate the sector trailer (block no. 7) again and in this case with the default key and if successful ('DEFAULT_OK'), the system will write the sector key to the sector trailer and yields 'OK' flag if the writing is successful (figure 5.1).

```
what is this? 539f731b,DEFAULT_OK

Raw data received: '539f731b,DEFAULT_OK'
Parsed UID: 539f731b
isInitialRead: true
Initial read - UID: 539f731b
Stored RFID data for UID: 539f731b
Keys stored in database successfully
```

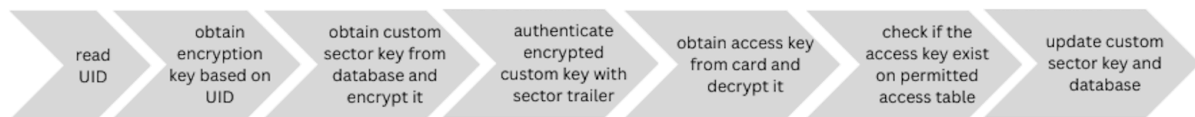
```
Starting write process...
Default key: FFFFFFFFFF
Custom key to write: 7df0827e7f6a
Access key to write: 4220f73f8e326e187681ffb06e0ea718
Authentication attempt 1...
Attempting to authenticate with key: FFFFFFFFFF
Sending command: AUTH,FFFFFFFFFFFF
Received response: '539f731b,DEFAULT_OK
AUTH_OK
539f731b,DEFAULT_OK
539f731b,'
Authentication successful!

Writing sector trailer...
Writing to block 7: 7df0827e7f6a
Sending command: WRITE_TRAILER,FFFFFFFFFFFF,7df0827e7f6a
Received response: 'DEFAULT_OK
539f731b,DEFAULT_OK
539f731b,DEFAULT_OK
539f731b,DEFAULT_OK
OK
'
Sector trailer written successfully
```

Figure 5.1

The system will then attempt to write the encrypted access key on sector 1, block no. 4, in which their sector trailer is just changed, so in order to authenticate and unlock the sector, the system needs to authenticate with the encrypted custom key, if successful the encrypted access key will be written to the block.

5.2) Authentication



The authentication process follows a secure multi-step protocol when a card is presented for access:

1. **Initial Card Detection**
 - System reads the card's UID
 - Queries database to retrieve the card's encryption key
 - Verifies the card's registration in the system
2. **Custom Key Verification**
 - System retrieves stored custom key from database
 - Encrypts it using the card's encryption key
 - Authenticates with the card using this key (check it with the sector trailer of sector 1)
 - Successful authentication grants access to read block 4
3. **Access Verification**
 - System reads encrypted access key from block 4
 - Decrypts it using the card's encryption key
 - Compares decrypted access key against permitted keys database
 - Determines access permission based on this comparison
4. **Session Management**
 - Before a session terminates, a new custom sector key is generated
 - New custom sector key is encrypted and written to the card
 - Database is updated with the new key
 - Previous session key is invalidated

5.3) Database Structure

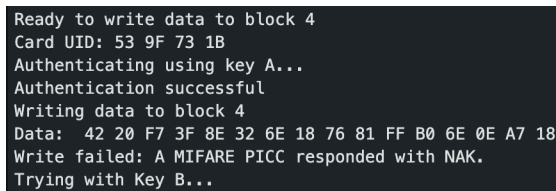
The database used PostgreSQL and consists of two main table:

- a) **rfid_keys** = consist of UID (primary key), Encryption Key, Custom Key, Timestamp
- b) **permitted_access** = consist of Access Key ID (primary key), Access Key (foreign key), UID (foreign key), Access Granted (boolean), Access Time

6) Result and Discussions

6.1) Reflections & Challenges Encountered

6.1.1) Access Key Writing Issues

A terminal window with a dark background and white text. The text shows the steps of writing data to a block: 'Ready to write data to block 4', 'Card UID: 53 9F 73 1B', 'Authenticating using key A...', 'Authentication successful', 'Writing data to block 4', 'Data: 42 20 F7 3F 8E 32 6E 18 76 81 FF B0 6E 0E A7 18', 'Write failed: A MIFARE PICC responded with NAK.', and 'Trying with Key B...'.

```
Ready to write data to block 4
Card UID: 53 9F 73 1B
Authenticating using key A...
Authentication successful
Writing data to block 4
Data: 42 20 F7 3F 8E 32 6E 18 76 81 FF B0 6E 0E A7 18
Write failed: A MIFARE PICC responded with NAK.
Trying with Key B...
```

figure 6.1

The most significant challenge remains in the access key writing implementation. While the writing process is indeed challenging, I only managed to authenticate and successfully write the custom sector key and validate that it is indeed written through status authentication. I tried to address the writing issues by constructing a standalone arduino file to manually write the access key in hard code (testing). The output shows successful authentication, meaning the custom key is indeed successfully writing the custom key into the sector trailer, but the data block writing fails. I have been trying to figure out the potential causes of the access key writing issues:

- Block access permission. There might be a permission issue on the sector 1 data block.
- Irreversible blocking due to format violation. There might be an attempt where I accidentally send the wrong format through the serial communication which leads the PICC to detect format violation when trying to write it, leading to irreversible blocking on the data blocks.

6.1.2) Hardware Setup

During the hardware setup phase, a major issue arose when one of the MFRC522 RFID readers was damaged due to improper soldering methods and incorrect pin connections. This incident underscored the need for careful hardware assembly and handling in RFID system setups. Applying too much heat or using an incorrect soldering technique while attaching the pins to the MFRC522 module caused component damage. Additionally, incorrectly connecting the pins between the Arduino and the RFID reader (potentially mixing up power, ground, or data pins) resulted in electrical damage to the hardware.

6.1.3) Communication Interference

Communication interference presented a complex challenge, impacting both the interaction between the RFID reader and card and the serial link between the Arduino and host system. In practice, we noticed that serial communication would occasionally lose sync, resulting in fragmented or corrupted data. This was especially apparent in the serial output, where repeated UID readings or mixed authentication responses would interrupt card detection messages. To maintain data integrity, the communication protocol between the Arduino and host required precise timing and careful buffer management to prevent overflow. For instance, during key-writing tasks, responses like "539f731b,DEFAULT_OK" would sometimes appear amidst authentication messages, complicating command processing. This issue highlighted the need for clear message framing and command sequencing to ensure consistent communication.

6.2) Comparison with Initial Objectives

- Implemented Secure Card Authentication through custom AES encryption and session key management.
- Created secure key storage which includes the encryption key management.

- Implemented custom sector key writing and verification.
- Implemented PostgreSQL integration

Testing needed: Full key rotation implementation, including writing access key, read initialised card and authenticate the access permission, and update the custom key.

7) Suggestions

7.1) Add custom keys history on the database for security audits

This feature provides crucial audit capabilities for security monitoring, forensic analysis, and compliance purposes. By maintaining a history of all key changes, the system can track suspicious patterns in key changes, investigate security incidents, monitor system usage patterns, ensure compliance with security policies, and aid in system recovery

7.2) Enhance database security by adding salt for each key

The current database setup directly stores encryption keys. Adding key salting would provide an extra layer of security. A salt is a unique, randomly generated value for each key that is combined with the key before storing it. This approach ensures that, even if two cards share the same key, the salted and hashed results will differ due to their unique salts. This technique protects against rainbow table attacks and ensures that even if one key is compromised, others remain secure.

7.3) Enhance communication security by adding message authentication

The current setup relies on plain text communication over the serial link between the Arduino and host system, which could be susceptible to interception or tampering. To enhance security, a more secure solution would involve incorporating message authentication and encryption for all communications. Each command exchanged between the host and Arduino should include a timestamp, a nonce (a unique, one-time-use random number), and a Message Authentication Code (MAC). The timestamp restricts messages to a valid time frame, protecting against replay attacks. The nonce introduces variability, ensuring that identical commands yield different encrypted outputs. The MAC, created using a cryptographic hash function such as HMAC-SHA256, guarantees both the integrity and authenticity of the message.

8) Conclusion

In conclusion, this RFID Encryption Project successfully achieved a secure access control system by implementing robust AES encryption and session-based key management, ensuring both data protection and authentication integrity. The reliable key management system, supported by a PostgreSQL database, allows for secure key storage, dynamic key rotation, and effective access control, which collectively strengthen the system's defence against unauthorised access and cloning attempts. While future improvements, such as enhanced message authentication and key rotation refinement, offer promising advancements, the project already demonstrates resilience against potential attacks, establishing a strong foundation for further enhancements and adaptation to evolving security needs in RFID applications.

9) Reference

<https://www.instructables.com/Reading-RFID-Tags-with-an-Arduino/>

More into physical security. Mentioned ffs

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7029721&tag=1>

Reverse engineering on rfid

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7945583>

SP networks (xor, subs, perm)

<https://www.youtube.com/watch?v=DLjzI5dX8jc&t=0s>

How AES works

<https://www.youtube.com/watch?v=O4xNJstN6E>

Asymmetric vs symmetric encryption

<https://preyproject.com/blog/types-of-encryption-symmetric-or-asymmetric-rsa-or-aes#:~:text=Symmetric%20vs%20asymmetric%20is%20a.to%20encrypt%20and%20decrypt%20data.>

Mixing columns

<https://www.youtube.com/watch?v=WPz4Kzz6vk4>

Aes encryption vs decryption

https://www.researchgate.net/figure/Original-AES-Encryption-Decryption-Steps_fig2_331756471

Read

<https://github.com/miguelbalboa/rfid/blob/master/examples/ReadNUID/ReadNUID.ino>

Rfid tag cloning

<https://getsafeandsound.com/blog/rfid-cloning/>