Imperial College London
Department of Computing

Practical Attacks on the MIFARE Classic

by

Wee Hon Tan (wht08)

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of Imperial College London

September 2009

Abstract

The MIFARE Classic is the most widely used contactless smart card chip in the world. Its communication is based on the open ISO-14443-A standard, but the entire authentication and encryption protocols are proprietary. Several academic researchers have cracked the encryption, and even proposed attacks to recover the secret keys. However, none of their attacks have been released so far. In this project, we analyse their attack descriptions and implement three attacks on the MIFARE Classic chip. The most critical attack recovers ANY secret key requiring wireless access to just the card only in less than five minutes on inexpensive commercial off-the-shelf hardware and without any pre-computation. Using our attacks, we expose the vulnerabilities of the Imperial College's access control system and show our ability to masquerade as any valid Imperial College personnel. Most importantly, we crack the internal structure of the Oyster card and locate the exact data bytes that represent the current credit and past transaction records. The impact of our findings are so severe that London's transport operators stand to lose millions of pounds if our findings were made public.

Acknowledgements

I would like to thank the following people for their help and support throughout this project:

My supervisor, Dr Maria Grazia Vigliotti, for her patience and understanding throughout the project. Even though this is a complete new area to the both of us, she never seems to be affected by the difficulties.

My family, for their support and encouragement, even when I am half the globe away and cannot be there to share their joy and pain.

My company, for their generosity in funding my dream of pursuing an overseas education.

And, last but not the least, my friends at Imperial College. Special thanks goes out to Siqi Zhao, for her words of encouragement and for being there for me when I am feeling down.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Radio-frequency identification (RFID) is the use of an object (typically referred to as an RFID tag) applied to or incorporated into a product, animal, or person for the purpose of identification and tracking using radio waves. Some tags can be read from several meters away and beyond the line of sight of the reader. This technique of identification had been used as early as World War II to determine if a distant fighter plane were a friendly or foe [Jou09]. However it was only in the 1970s when RFID technology developed into something like it is today. Since then, it has been used mainly to replace the legacy bar codes, magnetic stripe cards and paper tickets in a wide range of applications, including ticketing in major events and public transport systems, road pricing, access control systems, electronic passports and logistics. Figure 1.1 shows some of these uses. Almost every one carries a RFID chip these days. Moreover, the size of the RFID chip has also diminished dramatically over the years, with the smallest chip just 0.3 square millimeter in size, which makes it even less detectable. As the use of RFID becomes more pervasive in our daily lives, people are becoming more concerned with the security of the information that the chips possess, and the privacy and economical issues they bring.

(a) Road Pricing (b) Public Transport (c) Event Ticketing

Figure 1.1: Applications of RFID

There are many standards that cover the use of RFID. Most govern the message formats and modulation/demodulation of analog signals, but do not cover any security features such as authentication and encryption. It is thus left to the companies which implement it to develop a secure communication layer on top of the default transmission layer. In reality, to reduce production costs and to ensure compatibility among different products in the market, many companies usually adopt a single company's security design. The MIFARE range of chips manufactured by NXP (formerly Philips) is based on such a proprietary protocol. The one particular chip from this range that we will be looking at is the MIFARE Classic chip. Currently, 200 million cards with embedded MIFARE Classic chips are in circulation [Ver09a]. One important application is the Oyster card, an electronic ticketing system used on London's public transport services. The chip can also be

found in the Imperial College ID card.

There is a huge economical motivation with analysing the MIFARE Classic chip, and in particular, its application in the Oyster card. According to [oL07], in April 2007, more than 10 million Oyster cards have been issued and 38 million journeys are made each week using the Oyster. If attackers modify the chip so as to travel for free, Transport for London (TfL) stands to lose millions of pounds each year. University of Cambridge's Ross Anderson, who is one of the founding members of the discipline "Economics of Information Security" gives his opinion in [AM08] that the people who could protect systems from failure are often not the people who suffer the costs of failure. It is the commuters who will lose out in the long run as the insecurity of the chip cannot guarantee the safety of their money from malicious attackers.

The transaction history stored on the chip also poses privacy and legal issues. The attacker, armed with the right equipment, can read off a unknowing victim's card to deduce where the victim lives or works. With the ability to change the travel history, authorities cannot then rely on it as evidence to prosecute suspects who were thought to be at the scene of a crime. It is even more alarming as any previous convictions using such evidence can be challenged and reopened for examination.

1.2 Aims and Objectives

The broad aims of this project are to exploit the weaknesses of the MIFARE Classic chip via attacks and evaluate the impact of such weaknesses in critical applications. These can be further broken down into the following parts.

1. **Understand structure and evaluate weaknesses of MIFARE Classic** — Even though several aspects of the MIFARE Classic, including its internal structure and communication protocol, are proprietary, it has not prevented researchers from reverse engineering it and discovering its weaknesses. We will study their findings to see how they can be exploited.

2. **Understand and develop attacks based on weaknesses** — While researchers have published papers describing attacks aimed at these weaknesses, none of these attacks have been released to date. Moreover, most of the attack descriptions have been vague and inadequate. We will attempt to build on these attack descriptions and incorporate our own findings in our version of the attacks.

3. **Apply attacks on real-world case studies** — We have chosen to apply our attacks on two real-world case studies. The first is the Imperial College ID card. This platform is ideal for two reasons. Firstly we can test our attacks with the support of the College without fear of prosecution. Secondly it serves as a wake up call to the College on the vulnerabilities of the MIFARE Classic, and the College can make a sound decision whether to continue its use in the access control system. With the knowledge and experience gained from the first case study, we then move on to our second case study, which is the Oyster card.

4. **Uncover data structure of case studies** — We will dissect the data structures of our two case studies completely by looking for patterns in changes to the content and locating important bytes. In the case of the Oyster, these bytes include those that store the current credit remaining in the card and those that store transaction history.

Finally, we wish to emphasise the fact that while we share our attack descriptions and findings, we will not release our attack software or divulge the cryptographic keys for the sake of the parties involved.

1.3 Structure of Report

We begin the report in Chapter 2 with an introduction to RFID terminology, ISO-14443-A communication protocol used in the MIFARE Classic, and cryptography concepts that will help in the understanding of the chip. In this chapter, we also give an overview of the work done by researchers on the MIFARE Classic, and highlight the weaknesses and attacks discovered. We then move on to Chapter 3 where we look in greater detail at this chip, first by understanding its memory structure, and then analysing the messages transmitted during communication. In Chapter 4, we analyse the cipher system, Crypto1, used in the MIFARE Classic, and expose its weaknesses. Next we introduce the hardware and software tools that we use to attack the chip in Chapter 5, and the attacks that we have implemented in Chapter 6. We share our findings from analysing the data structures of the Imperial College ID card and the Oyster card in Chapter 7. And finally we wrap up by evaluating the attacks tools used and comparing our card-only attack against those proposed by other researchers in Chapter 8.

Chapter 2

Background Information & Related Work

In this chapter, we first introduce important RFID terminology and the ISO-14443-A communication protocol used by the MIFARE Classic. We then describe key concepts of cryptography related to our work. We end this chapter with an overview of the work done by researchers on the MIFARE Classic, and describe the weaknesses and attacks discovered, some of which we extend in our work.

2.1 RFID

The ISO-14443-A is a four-part standard developed by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) for describing the parameters for identification cards and the use of such cards for international interchange. This ISO standard sets communication standards and transmission protocols between card and reader to create interoperability for contactless smart card products. In this standard, two compulsory hardware parts are clearly distinguished. They are the Proximity Coupling Device (PCD) and the Proximity Integrated Circuit Card (PICC).

2.1.1 PCD & PICC

The PCD, commonly referred to as the reader or writer, houses an antenna to communicate at the 13.56 MHz (±7 kHz) frequency. It generates an electronic field during communication, which can be used to power the PICC. A reader is usually connected to a PC, and commands issued from the PC are sent on the wire to the PCD to transmit over the antenna. The PICC, commonly referred to as the tag, can be either passive or active, and the difference lies in the power supply. Passive tags, which are more commonly seen and relatively inexpensive, depends on the electronic field of the reader for power, and as such the communication range is small (less than 4 inches). The antenna runs along the perimeter of the card and is connected to the chip (see Figure 2.1). The PICC has an internal capacitor, which stores the energy from the reader's electronic field, and a resistor. To reply to the PCD, the PICC uses the resistor to create a small field resistance in the sub-carrier frequency at 847 kHz, which is a divisor of the reader's 13.56 MHz main carrier frequency, to be picked up by the reader.

2.1.2 ISO-14443-A Signal Interface

The PCD uses the Modified Miller encoding to communicate with the PICC, and the PICC uses the Manchester encoding in its replies. The Modified Miller is best explained by a bit sequence example, as shown in Figure 2.2a. To encode a 0, if the previous but was a 0, the electronic field will be dropped completely (shown as a gap in the figure) for 2.28 µs at the beginning of the bit duration. If the previous bit was not a 0, the field remains on. To encode a 1, the field is dropped

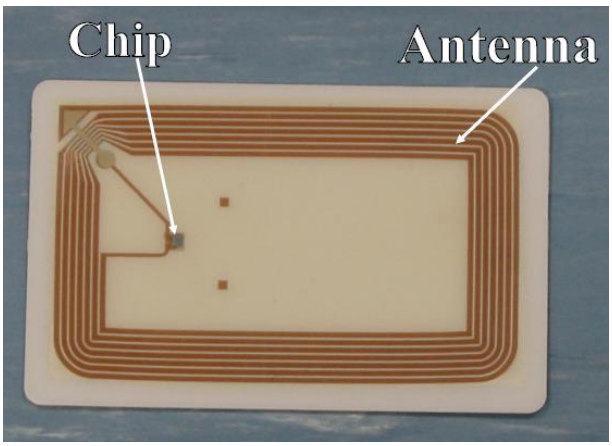

Figure 2.1: PICC

at the middle of the bit duration, regardless of the previous bit. Even though the field is dropped for a very short duration, the PICC gets its power from the internal capacitor.

(a) Modified Miller (b) Manchester

Figure 2.2: Encoding methods used by PCD and PICC

The Manchester encoding is shown in Figure 2.2b. Compared to the Modified Miller, it is simpler to encode. The bit duration is split into two halves. A 0 is encoded by introducing field resistance (shown as a saw-edged transmission in the figure) in the second half, while a 1 is encoded by introducing field resistance in the first half. Knowledge of how Modified Miller and Manchester encodings work is necessary when we program our tools to encode signals to communicate with the PCD or PICC.

2.2 Cryptography Concepts

2.2.1 Symmetric encryption

Cryptography is the science and practice of hiding information [Sta05]. One of the dimensions that characterises cryptosystems is the number of keys used. Symmetric encryption, also known as conventional encryption, is one form of cryptosystem in which encryption and decryption are done with a single key. Together with an encryption algorithm, the original message in clear, also known as plaintext, is transformed into its encrypted form, also known as ciphertext. Using the same key and a decryption algorithm, the plaintext is recovered from the ciphertext with no loss of information. The key is independent of the plaintext and of the algorithm. The algorithm will produce a different output depending on the specific key being used at the time.

In general, there are two ways to break a cryptosystem. One way is to rely on the nature of the algorithm and some knowledge of the general characteristics of the plaintext or sample plaintext-ciphertext pairs. This method is known as cryptanalysis. The other way, known as the brute-force attack, is a form of attack where the attacker tries every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained. On average, half of all possible keys must be

tried to achieve success, and the larger the number of possible combinations of the key, the harder it is to brute-force.

In the context of our project, the MIFARE Classic uses symmetric encryption in the communication between the PCD and the PICC. Both the PCD and the PICC have a copy of the keys. We will look in greater detail at the cryptosystem in Chapter 4.

2.2.2 Stream Ciphers

Another dimension that characterises cryptosystem is the way in which plaintext is processed. A block cipher, as the name suggests, processes one block of plaintext elements at a time and produces a ciphertext element block for each plaintext element block. A stream cipher, on the other hand, processes plaintext elements one by one, producing ciphertext element one at a time, as it goes along. In our following discussions, the unit of one element is a bit.

Figure 2.3 shows a representative diagram of stream cipher structure. In this structure a key is input to a pseudo-random bit generator that produces a stream of bits that are apparently random. At the encryption end, the output of the generator, called a keystream, is combined one bit at a time with the plaintext stream using the bitwise exclusive-OR (XOR) operation to get the ciphertext. At the decryption end, the generator, if given the same key, will produce the same keystream to be combined with the ciphertext using XOR to get the plaintext. Figure 2.4 shows an example of how it works.

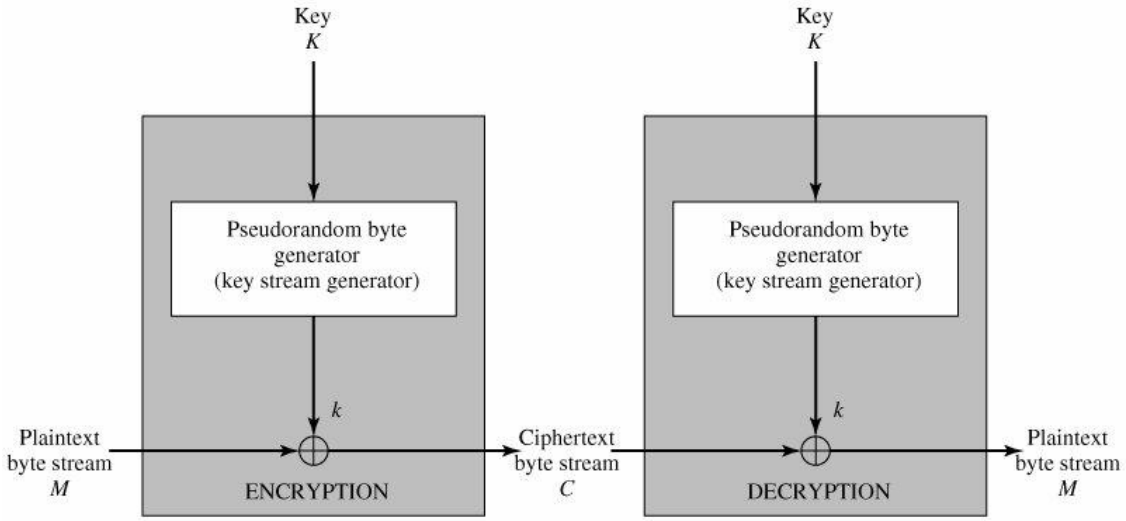

Figure 2.3: Structure of stream ciphers

Plaintext		0 1 1 0 1 1 0 1
Keystream	⊕	1 1 0 1 0 1 0 1
Ciphertext		1 0 1 0 1 0 0 0

(a) Encryption

Ciphertext		1 0 1 0 1 0 0 0
Keystream	⊕	1 1 0 1 0 1 0 1
Plaintext		0 1 1 0 1 1 0 1

(b) Decryption

Figure 2.4: Cipher stream operations

There are many different ways to implement a pseudo-random bit generator. One way is to combine a linear feedback shift register (LFSR) with a nonlinear filter generator. A LFSR is a shift register whose input or feedback bit is a linear function of its previous state. The linear

function in this case is the exclusive-or (XOR), and hence the input bit is the exclusive-or of some pre-determined bits of the overall shift register value. The pre-determined bit locations used to produce the input bit are usually expressed as a generating polynomial. The initial value of the LFSR is called the seed, and because the function of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen linear feedback function can produce a sequence of bits which appears random and which has a very long cycle. LFSRs can be implemented in hardware inexpensively, and this makes them useful in applications that require very fast generation of a pseudo-random sequence, such as in stream ciphers.

The output bits from the LFSR for the keystream are produced by the nonlinear filter generator. The purpose of having the filter generator is to break the inherent linearity of the LFSR. The state bits of the shift register are treated as if they were independent random variables with probability 1/2 of being 0 or 1. The filter generator will combine selected state bits based on its nonlinear function to produce independent random output, as required in the pseudo-random bit generator.

In the context of our project, the MIFARE Classic chip employs two pairs of LFSR and filter function. We will revisit them in greater detail in Chapter 4.

2.2.3 Challenge-Response Authentication

In computer security, challenge-response authentication is an authentication mechanism in which a system presents a question, known as a challenge, and the user must provide a valid answer, known as a response, to be authenticated. The simplest example is password authentication, where the challenge is the request for password and the valid response is the password itself. However, this is also the most insecure example, as the attacker can easily eavesdrop or sniff the authentication request to get the password. Cryptographic solutions have been employed to overcome this problem, and most involve mutual authentication, where both the user and the system must each convince the other that they know the shared secret (the password), without this secret ever being transmitted in the clear over the communication channel.

One way this is done involves using the password as the encryption key to transmit some randomly-generated information as the challenge, whereupon the other end must return as its response a similarly-encrypted value which is some predetermined function of the originally-offered information, thus proving that it was able to decrypt the challenge. The use of information which is randomly generated, usually called a nonce, on each exchange (and where the response is different from the challenge) guards against the possibility of a replay attack, where the attacker simply records the exchanged data and retransmits it at a later time to fool one end into thinking it has authenticated a new connection attempt from the other. If it is impractical to implement a true nonce, a strong cryptographically secure pseudo-random number generator can generate challenges that are highly unlikely to occur more than once. The challenge nonce and the secret may even be combined to generate an unpredictable encryption key for the session.

In the context of our project, the MIFARE Classic chip performs a challenge-response mutual authentication with the PCD, using randomly generated nonces from a LFSR. We will examine this LFSR and the authentication mechanism in greater detail in Chapter 4.

2.3 Related Work

Several researchers have been credited with the discovery of crucial weaknesses in the MIFARE Classic chip in the past two years. Karsten Nohl and Henryk Plötz were the first to disclose the weaknesses in the pseudo-random number generator and in the authentication protocol in [NP07]

and [NESP08], where they had used traditional reverse engineering techniques to uncover the hardware structure of the chip from the silicon implementation. Another team of researchers, led by Professor Bart Jacobs from the Digital Security Group at Radboud University Nijmegen, also began analysing the MIFARE chips that were planned for use on the Dutch public transport. They confirmed the weakness in the pseudo-random number generator in [dHG08], and also recovered via experiments the proprietary command set (which was still a secret at the time) used in the communication between the PCD and PICC. They were the first to propose an attack based on this weakness that allowed them to obtain partial data from the PICC.

Garcia *et.al.* followed up in [GdM+08] with the full disclosure of how the cryptographic cipher is initialised, the nonlinear filter generator's function, and its four weaknesses. With this discovery, the proprietary structure of the MIFARE Classic was no longer a secret. Based on these weaknesses, they proposed a practical attack that allow the adversary to obtain parts of the keystream from intercepted communication between the PCD and PICC, and recover the key used in that communication session. The impact of the discovery was so great that NXP attempted, but failed, to get a court injunction to stop them from publishing their findings. We will look in greater detail at this attack in Section 6.1.

Thus far, proposed attacks require an eavesdropped communication session between the PICC and the genuine PCD. Although this is already disastrous from a cryptographic perspective, system integrators maintain that such attacks cannot be performed undetected. Researchers hence continued to look for attacks that work directly on a card without the help of a genuine reader. Professor Bart's team first discovered weaknesses in the parity bits and proposed two card-only chosen ciphertext attacks in [GvVS09]. However their attacks require pre-computation of a large lookup table of 300GB in size. Nicolas T. Courtois from University Central London then proposed an improved attack in [Cou09] that combines the weaknesses of the parity bits, the pseudo-random number generator, and the nonlinear filter generator. His attack does not require pre-computation and lookup of the large table. We will look at this attack in greater detail in Section 6.3.

In summary, these researchers have discovered critical weaknesses in the following areas:

1. Pseudo-random number generator for nonces
2. Authentication protocol
3. Nonlinear filter generator for keystream
4. Parity bits generation
5. Parity bits encryption and communication

With this background information and work done by other researchers in this area, we will look in greater detail at the MIFARE Classic chip in the following chapter.

Chapter 3

MIFARE Classic

In 1995, NXP introduced the MIFARE family of chips as an advanced technology for RFID identification, and targeted its application in public transportation, access control and event ticketing systems. The MIFARE Classic is a mid-range member of the MIFARE product family, which also includes the cheaper Ultralight and more expensive DESFire. There are two variants of the Classic chip (1K and 4K) but the length of the key (48 bits) is the same. We describe in detail the logical structure, memory access and communication of the 1K Classic chip in the following sections.

3.1 Logical Structure

The MIFARE Classic chip is a memory chip in principle, albeit with a few additional features [NXP08a]. The basic unit of the EEPROM memory is a 16-byte data block. Data blocks are grouped together to form sectors. The difference between the 1K and 4K Classic cards not only lies in the memory size, but also the organization of the memory [NXP08b]. The memory in the 1K card is divided equally into 16 sectors of 4 data blocks each. In the 4K card, the first 32 sectors are 4-block sectors, but the remaining 8 sectors are 16-block ones. The last block of every sector is known as the sector trailer and its contents define keys and access conditions to the four blocks within that sector. Figure 3.1 shows the logical structure of the MIFARE Classic 1K chip.

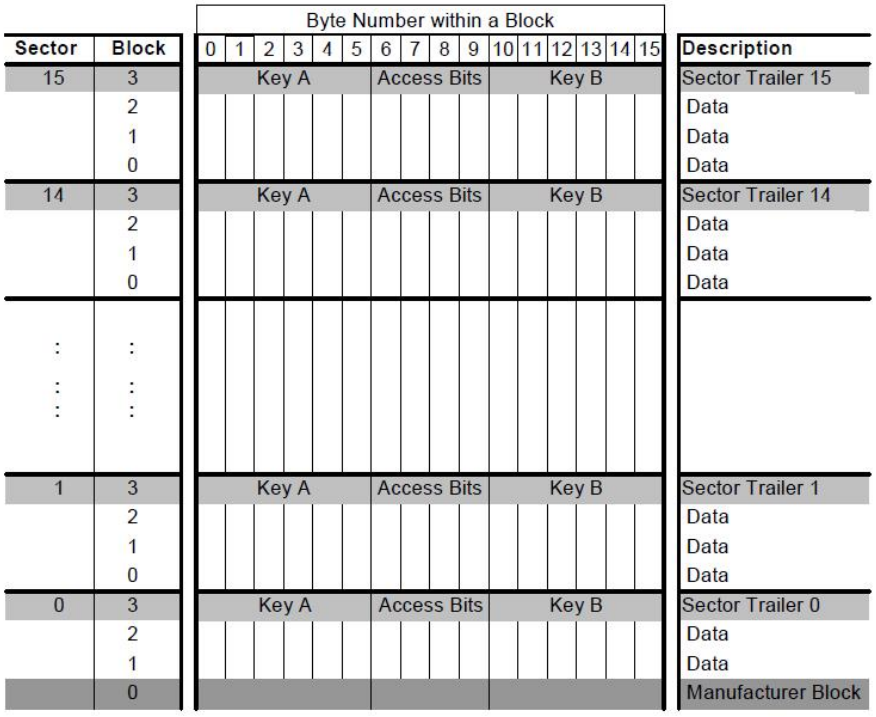

Figure 3.1: MIFARE Classic 1K Memory

Block 0 of sector 0 is called the Manufacturer Block and contains special data (see Figure 3.2). The first four bytes store the unique identifier (UID) of the card. The following 1 byte contains the bit count check (BCC) which is calculated by XOR-ing the separate UID bytes. The remaining bytes store manufacturer data. This block is set and locked at point of manufacture so the contents are not modifiable.

Figure 3.2: Manufacturer Block

The reader needs to be authenticated for a sector before any memory operations on the data blocks of that sector are allowed. The block governing authentication is the last block of each sector known as the sector trailer (see Figure 3.3). It holds

- Secret key A from bytes 0 to 5,
- The access conditions or bits of the four blocks in that sector from bytes 6 to 9, and
- Secret key B (optional) from bytes 10 to 15

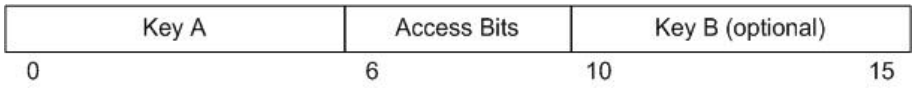

Figure 3.3: Sector Trailer

Key A is never readable, and Key B is readable in some cases when Key A is known. The access bits also specify the type of the data block, i.e. whether the block is a read/write block or a value block. If Key B is not needed, bytes 10 to 15 can be used to store data.

In the 1k card, the second and third blocks of sector 0, and the first three blocks of sectors 1 to 15 are used to store data. Data blocks can be configured by the access bits as read/write blocks used in applications such as contactless access controls systems, or as value blocks used in electronic wallet applications. Any modification to any data block has to be pre-empted by an authentication.

Value blocks store the value of credit unused. The entire value block is generated by a memory write operation. The block format (see Figure 3.4) is designed such that error detection, error correction and backup management can be performed. For reasons of data integrity and security, the first 12 bytes consist of the Value representing a signed 4-byte value stored three times, twice non-inverted (bytes 0 to 3 and bytes 8 to 11) and once inverted (bytes 4 to 7, shown with a overline in the figure). The remaining 4 bytes hold the 1-byte address stored four times, twice inverted and twice non-inverted. This single byte is used to hold the storage address of the block in a backup management system. During memory increment, decrement, restore and transfer operations, the address remains unmodified and can only be altered via a write command.

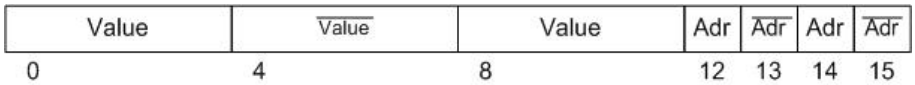

Figure 3.4: Value Block

3.2 Memory Access

The functional specifications for both 1K and 4K MIFARE Classic chip define six operations that can be used to access the data blocks. In addition to these six memory operations, each data block has three bits that define the access conditions (condition defining the access right to execute any of the six memory operations and using which key). These bits are defined in the sector trailer. The sector trailer also has three access condition bits, but since the sector trailer is not used for normal data storage, the access conditions are only defined for read and write operations on the keys and access bits.

3.2.1 Memory Operations

The six memory operations are best described in Table 3.1.

Operation	**Description**	**Valid for Block Type**
Read	Reads a single memory block	read/write, value and sector trailer
Write	Writes a single memory block	read/write, value and sector trailer
Increment	Increments the value and stores the result in the internal data register	value
Decrement	Decrements the value and stores the result in the internal data register	value
Transfer	Transfers contents of internal data register to a block	value
Restore	Reads contents of a block into the internal data register	value

Table 3.1: Memory operations

3.2.2 Access Conditions

Each data block or sector trailer has 3 bits that define the access conditions, and the bits are stored once inverted and once non-inverted in the sector trailer of the specified sector (see Figure 3.5). The bits control the rights of memory access using secret keys A and B. The format of the access bits is verified by internal logic with each memory access and if a violation is detected, the whole sector is irreversibly locked. The access conditions for the sector trailer and data block are shown in Tables 3.2 and 3.3 respectively.

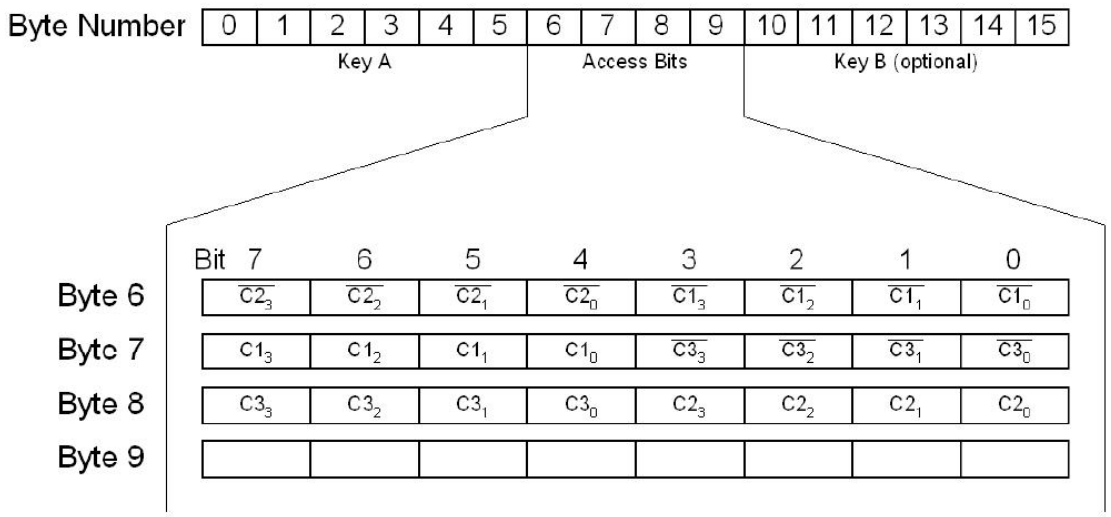

Figure 3.5: Layout of access bits in sector trailer

Access bits			Access condition for					
			KEY A		Access bits		KEY B	
C1	C2	C3	read	write	read	write	read	write
0	0	0	never	key A	key A	never	key A	key A
0	1	0	never	never	key A	never	key A	never
1	0	0	never	key B	key A or B	never	never	key B
1	1	0	never	never	key A or B	never	never	never
0	0	1	never	key A	key A	key A	key A	key A
0	1	1	never	key B	key A or B	key B	never	key B
1	0	1	never	never	key A or B	key B	never	never
1	1	1	never	never	key A or B	never	never	never

Table 3.2: Access conditions for sector trailer

Access bits			Access condition for				Application
C1	C2	C3	Read	Write	Increment	Decrement, Transfer, Restore	
0	0	0	key A or B	key A or B	key A or B	key A or B	transport configuration
0	1	0	key A or B	never	never	never	read/write block
1	0	0	key A or B	key B	never	never	read/write block
1	1	0	key A or B	key B	key B	key A or B	value block
0	0	1	key A or B	never	never	key A or B	value block
0	1	1	key B	key B	never	never	read/write block
1	0	1	key B	never	never	never	read/write block
1	1	1	never	never	never	never	read/write block

Table 3.3: Access conditions for data blocks

3.3 Communication

The communication behind the MIFARE Classic chip follows the ISO-14443-A Part 3 standard. Standardization under ISO or IEC specifications ensures the interoperability of various components produced worldwide by a number of manufacturers. Without such standards, each manufacturer would design products to its own internal specifications only, in essence creating an environment comprised of multiple proprietary systems. No component would be compatible with any competitor's components.

Figure 3.6 shows the top-down communication flow of the MIFARE Classic. It starts from the top, and usually moves downwards stage by stage, completing the stage before it can go to the next (shown as solid arrows). However, it also allows a reset back to the top of the flow at certain stages (shown as dotted arrows). The first three stages are also collectively referred to as the anticollision phase. In the following sections, we will describe the various stages and the commands that are transmitted in that stage. We will also attempt to explain the abbreviations as specified in the standard.

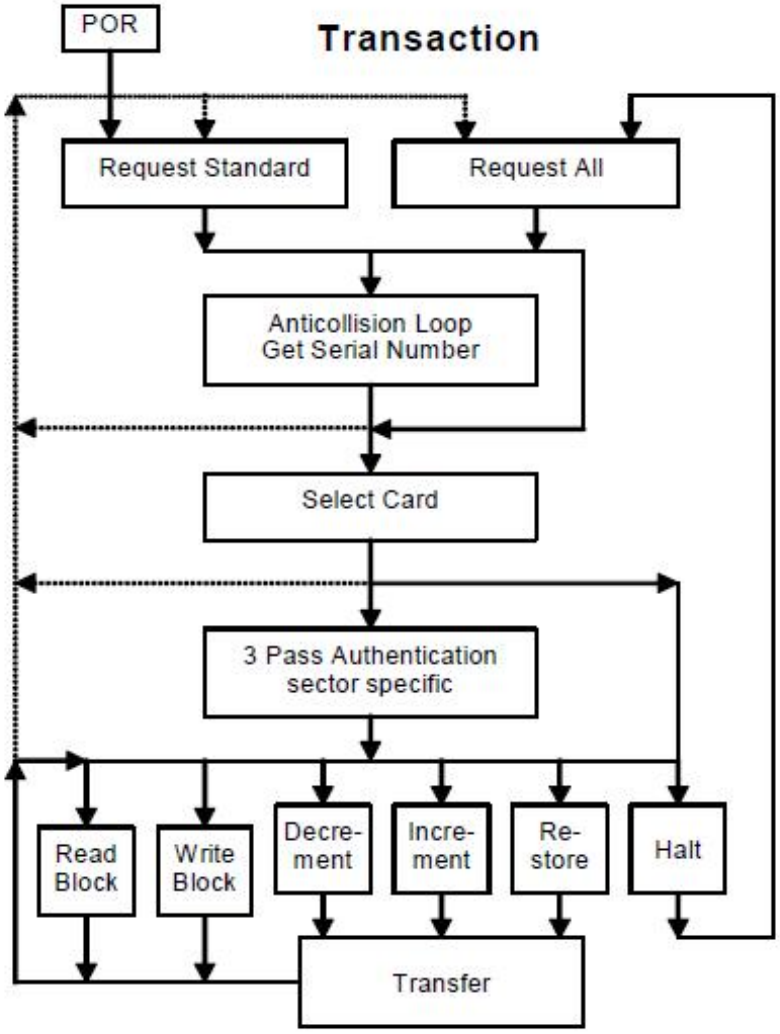

Figure 3.6: Communication Flow of MIFARE Classic

3.3.1 Request standard/Request all

The tag starts in the Power On Reset (POR) state, where it is not in a electronic field of the reader and hence has no power. In order to detect tags in the operating field, a reader shall send

repeated Request standard commands — "REQuest command, Type A" or REQA, or Request all commands — "Wake UP command, Type A" or WUPA. The latter command will wake all tags in the POR state or tags that are in the HALT state (we will explain the Halt command later). When a tag is within range of the reader, it will respond within 5 ms with a "Answer To reQuest, Type A" or ATQA answer. The reader now recognises that at least one card in the read range and begins the anticollision loop sequence.

3.3.2 Anticollision loop

In the anticollision loop stage, if there is only one tag in the vicinity, the UID of the tag is obtained by the reader with the "ANTICOLLISION command" or AC, and the loop terminates. If there are many tags in the vicinity, a collision occurs. A collision occurs when at least two PICCs simultaneously transmit bit patterns with one or more bit positions in which at least two PICCs transmit complementary values. The reader makes note of the bit collision position and therefore reads only those bytes/bits that are valid. The read data is then set as the new search criterion. The reader then sends out another AC updated with the new search criterion and length. The PICC whose UID matches that of the search criterion will respond. This is repeated until there are no collisions. Those not selected by the reader will go into a standby mode and wait for a new request command.

3.3.3 Select card

The reader will send a Select card command or SEL to the chosen card for authentication and subsequent memory access. The card will return a Select AcKnowledge reply or SAK. The SAK and the ATQA messages from earlier also determine the card type and the manufacturer name. Table 3.4 lists some of the well known SAK and ATQA responses. We will revisit these values when we look at our case studies in Chapter 7.

Manufacturer	**Product**	**ATQA**	**1st byte of SAK in hexadecimal**
NXP	MIFARE Mini	04 00	09
	MIFARE Classic 1K	04 00	08
	MIFARE Classic 4K	02 00	18
	MIFARE Ultralight	44 00	00
	MIFARE DESFire	44 03	20
	MIFARE DESFire EV1	44 03	20
	JCOP31	04 03	28
	JCOP31 v2.4.1	48 00	20
	JCOP41 v2.2	48 00	20
	JCOP41 v2.3.1	04 00	28
Infineon	MIFARE Classic 1K	04 00	88
Gemplus	MPCOS	02 00	98

Table 3.4: Well known tag SAK values

3.3.4 Three-Pass Authentication

In Section 2.2.3, we described the challenge-response mutual authentication using nonces as challenge. The response is the output from a transformation function that takes the challenge as the input, and is sent encrypted with the secret key. The secret key is hence not transmitted in any way. The second challenge is sent together with the response so as to reduce communication overheads. Therefore, at the end of this authentication, both parties can decrypt and verify the correctness of the responses, and are certain the other party has the right key. In the MIFARE Classic, the authentication follows exactly that with the three-pass authentication stage. The key

used is the key of the sector of which the following memory operations request access to. The three-pass authentication sequence can be summarised by the sequence diagram in Figure 3.7. After a successful authentication, all memory operations that follow are encrypted.

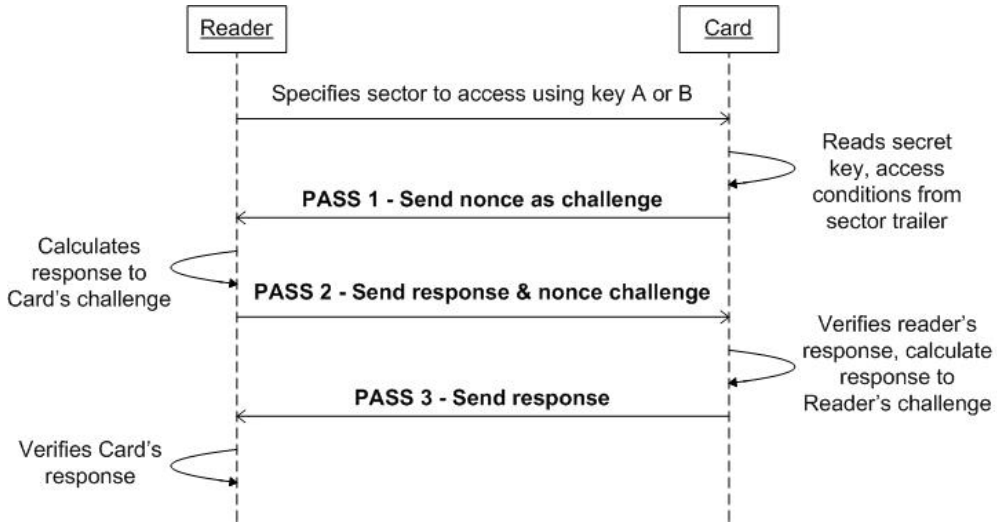

Figure 3.7: Three pass authentication

3.3.5 Halt

At any point after a card has been selected by the reader, the reader can cease all communication with it by issuing a Halt command — "HALT Command, Type A" or HLTA. The tag will go into the HALT state, and can only be awakened by the WUPA command.

3.3.6 An Example Trace

To give a better picture of how the commands work together, we show an example trace in Table 3.5 which illustrates a complete communication flow.

SEQ	Sender	Bytes in hexadecimal	Command
1	PCD	26	REQA (WUPA if it is 52)
2	PICC	04 00	ATQA
3	PCD	93 20	AC
4	PICC	**2A 69 8D 43 8D**	UID & BCC
5	PCD	93 70 **2A 69 8D 43 8D** 52 55	SEL
6	PICC	08 B6 DD	SAK
7	PCD	60 04 D1 3D	Authenticate block 0x04 using key A
8	PICC	3B AE 03 2D	Tag nonce
9	PCD	C4 94 A1 D2 6E 96 86 42	Reader nonce & response
10	PICC	84 66 05 9E	Tag response
11	PCD	50 00 57 CD	HLTA

Table 3.5: An Example Trace

SEQ 1 to 6 are the communication for the anti-collision phase, and SEQ 7 to 10 are the communication for the authentication phase (in this example, the request is for block 0x04 using key A). Once authentication is complete, any command from the command set in Figure 3.8 can be issued by the reader after SEQ 10. However since the issued commands would have been encrypted, we do not show them in the example trace.

Authentication

READER	*CARD*	*READER*	*CARD*
60 YY* Using KeyA	4-byte nonce	8-byte response	4-byte response
61 YY* Using KeyB	4-byte nonce	8-byte response	4-byte response

Data

READER	*CARD*	*READER*
30 YY* Read	16 data bytes*	
A0 YY* Write	ACK / NACK	16 data bytes*

Value blocks

READER	*CARD*	*READER*	*READER*
C0 YY* Decrement	ACK / NACK	4-byte value*	Transfer
C1 YY* Increment	ACK / NACK	4-byte value*	Transfer
C2 YY* Restore	ACK / NACK	4-byte value*	Transfer
B0 YY* Transfer	ACK / NACK		

Other

READER

50 00* Halt

YY = block address
* = Followed by two CRC bytes

Card responses (ACK / NACK)

A (1010) ACK
4 (0100) NACK, not allowed
5 (0101) NACK, transmission error

Figure 3.8: Command set of MIFARE Classic

After looking at the MIFARE Classic structure in greater detail, we shall describe the cryptography behind the communication in the following chapter.

Chapter 4

Crypto1

Crypto1 is a proprietary encryption algorithm developed by NXP Semiconductors specifically for use in the MIFARE Classic chip. The algorithm is implemented in hardware on-chip for rapid cryptographic operations. The security of the card relies in part on the secrecy of the Crypto1 algorithm, which is commonly known as "security by obscurity". Being a proprietary algorithm, it is a trade secret of NXP and even system integrators are not privy to its internal structure. However security researchers Nohl and Plötz have been able to uncover the algorithm by applying reverse engineering techniques directly on the silicon implementation ([NP07] and [NESP08]). Garcia *et.al.* had also in [GdM+08] fully disclosed the entire algorithm. We will consolidate and describe their findings on the algorithm in the following sections. We will end the chapter highlighting the weaknesses that these researchers have discovered.

4.1 Stream Cipher

Crypto1 is essentially a stream cipher (refer to Section 2.2.2 for more information on stream ciphers). The pseudo-random bit generator at the centre of the cipher that produces the keystream bits is made up of a 48-bit linear feedback shift register (LFSR) and a two-layer non-linear filter generator. Figure 4.1 shows the LFSR with the labelled bits 0 to 47, and the two-layer filter generator comprising of the functions f_a, f_b and f_c.

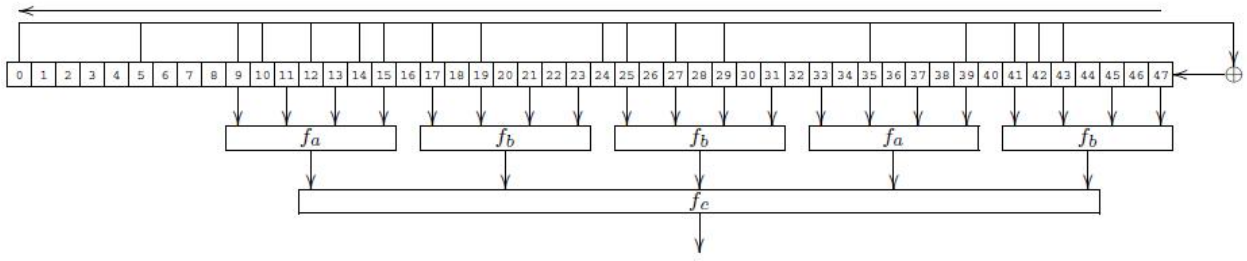

Figure 4.1: Diagram showing LFSR and filter function of Crypto1

4.1.1 Keystream Generation

During each clock cycle, the non-linear filter generator takes twenty bits from the LFSR (shown as downward arrows in Figure 4.1) and generates one keystream bit. The LFSR then shifts one bit to the left, discarding the leftmost bit (the 0^{th} bit) and inserting a new bit generated using the generating function, also known as the feedback function, on the right. If we use x_n to denote the bit value (0 or 1) at position n and $\oplus$ is the exclusive-or (XOR) logic function, the feedback function is then defined by

$$L(x_0x_1...x_{47}) := x_0 \oplus x_5 \oplus x_9 \oplus x_{10} \oplus x_{12} \oplus x_{14} \oplus x_{15} \oplus x_{17} \oplus x_{19} \oplus x_{24} \oplus x_{25} \oplus x_{27} \oplus x_{29} \oplus x_{35} \oplus x_{39} \oplus x_{41} \oplus x_{42} \oplus x_{43} \quad (4.1)$$

The bits chosen as inputs in the feedback function are shown in Figure 4.1 with upwards arrows.

The filter generator that generates one keystream bit at every clock cycle can be defined as a combination of three logic sub-functions f_a, f_b and f_c. Using the same notation x_n as above, and introducing the logical OR operator $\vee$ and the logical AND operator $\wedge$, we define the filter generator and the three sub-functions f_a, f_b and f_c below.

$$f(x_0x_1...x_{47}) := f_c(f_a(x_9, x_{11}, x_{13}, x_{15}), f_b(x_{17}, x_{19}, x_{21}, x_{23}), f_b(x_{25}, x_{27}, x_{29}, x_{31}), f_a(x_{33}, x_{35}, x_{37}, x_{39}), f_b(x_{41}, x_{43}, x_{45}, x_{47})) \quad (4.2)$$

$$f_a(a, b, c, d) := ((a \vee b) \oplus (a \wedge d)) \oplus (c \wedge ((a \oplus b) \vee d)) \quad (4.3)$$

$$f_b(a, b, c, d) := ((a \wedge b) \vee c) \oplus ((a \oplus b) \wedge (c \vee d)) \quad (4.4)$$

$$f_c(a, b, c, d, e) := (a \vee ((b \vee e) \wedge (d \oplus e))) \oplus ((a \oplus (b \wedge d)) \wedge ((c \oplus d) \vee (b \wedge e))) \quad (4.5)$$

One important point to note is that the inputs to the filter function are evenly spaced out odd state bits starting from the 9^{th} bit. This is a critical weakness that we will elaborate in Section 4.4.

4.2 Pseudo-random Number Generator

We recall from Section 2.2.3 the concept on the challenge-response mutual authentication. The authentication protocol used in the MIFARE Classic follows exactly such a challenge-response mutual authentication mechanism. The challenge is a 32-bit nonce that is generated by a pseudo-random number generator, and it was revealed in [NP07] to be a 16-bit LFSR which is a separate circuit from that of the 48-bit LFSR used for the cipher. We noticed immediately a weakness with the LFSR length — even though the generated nonce is 32-bit wide, the LFSR that generates the nonce uses only half that width, at 16-bit (see Section 4.4 for more information on this weakness).

During each clock cycle, the LFSR shifts one bit to the left, discarding the leftmost bit and inserting a rightmost bit generated using its feedback function. Again we use x_n to denote the bit value (0 or 1) at position n and $\oplus$ is the exclusive-or (XOR) logic function. Its feedback function is then defined by

$$L(x_0x_1...x_{15}) := x_0 \oplus x_2 \oplus x_3 \oplus x_5 \quad (4.6)$$

Moreover, if we define a 32-bit LFSR sequence $x_0x_1...x_{31}$ consisting of this 16-bit LFSR, then the next 32-bit sequence can be defined using a successor function suc where

$$suc(x_0x_1...x_{31}) := x_1x_2...x_{31}L(x_{16}x_{17}...x_{31}) \quad (4.7)$$

$$suc^n(x_0x_1...x_{31}) := suc^{n-1}(suc(x_0x_1...x_{31})) \quad (4.8)$$

Recall from Section 2.2.3 on the challenge-response authentication, the response to a challenge is the output from a pre-determined function, with the challenge as its input. The role of the successor function is precisely that of the pre-determined function. The next section explains in greater detail this challenge-response mutual authentication protocol.

4.3 Authentication

In official NXP documentation for the MIFARE Classic chip ([NXP08a] and [NXP08b]), authentication between the reader and tag is described as a three-pass mutual authentication. The reader first requests to be authenticated for a particular block. Then the following three passes occur.

1. PASS 1 — The tag sends a challenge nonce N_T to the reader.

2. PASS 2 — From this message onwards, communication is encrypted. We will explain in detail how the entire cipher is initialised in the following section. The reader replies with an encrypted reader nonce N_R and an encrypted reader response A_R to the challenge in PASS 1.

3. PASS 3 — If the reader response is correct, the tag will respond with an encrypted tag answer A_T. If the reader response is incorrect, the tag will not respond.

Mutual authentication is achieved with the successful completion of the three-pass mutual authentication. Once authenticated, any subsequent operation on the blocks within the same sector will be allowed. However, exactly how A_R and A_T are calculated is not disclosed in the NXP documentation. The encryption of the N_R, A_R and A_T is also not revealed. Figure 4.2 summarises a successful mutual authentication.

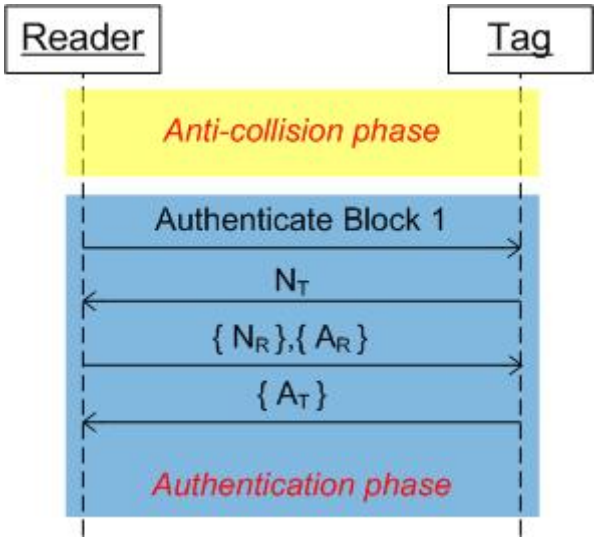

Figure 4.2: Example of authentication phase

Garcia *et.al.* were the first to uncover the details of the three-pass authentication protocol and the initialisation of the stream cipher in [GdM+08]. The reader and tag responses, A_R and A_T, are both derived from the tag nonce using the following relationships

$$A_R := suc^{64}(N_T) \tag{4.9}$$

$$A_T := suc^{96}(N_T) \tag{4.10}$$

Here we see how the successor function is used to calculate the response from the initial tag challenge. By checking the correctness of A_R after decrypting its ciphertext, the tag is able to verify that the reader has the right key. If A_R is incorrect, the tag is cease its communication with the reader. Likewise, the reader will check the correctness of A_T.

4.3.1 Cipher Initialisation & Keystream Generation

The stream cipher has to be initialised at both the reader and tag ends during the authentication session before it can be used for subsequent encryption and decryption. The parameters necessary are the 48-bit sector key K, the tag nonce N_T, the tag's unique identifier *uid* and the reader nonce N_R. The cipher is first initialised with the 48-bit sector key, i.e. the most significant bit (MSB) of the key is place in position x_0, the next MSB is placed in position x_1, and so on. Then $N_T \oplus uid$ is shifted in together with the feedback bit by bit, and finally N_R is shifted in together with the feedback bit by bit. Both the tag and reader must achieve the same cipher state at the end of initialisation for authentication to be achieved. We will use the following definition to formalise the initialisation of the cipher and the generation of the LFSR keystream.

Definition 1 — Cipher Initialisation Given sector key K, tag nonce N_T, unique identifier U, reader nonce N_R, where

$$K := k_0 k_1 k_2 ... k_{47} \tag{4.11}$$

$$N_T := n_{T,0} n_{T,1} ... n_{T,31} \tag{4.12}$$

$$U := u_0 u_1 ... u_{31} \tag{4.13}$$

$$N_R := n_{R,0} n_{R,1} ... n_{R,31} \tag{4.14}$$

then the internal cipher state at time i S_i, which is the snapshot of the values of the 48 bits in the cipher at time i, is defined as

$$S_i := s_i s_{i+1} ... s_{i+47} \tag{4.15}$$

and s_i is defined by the following relations.

$$s_i := k_i, \quad \forall i \in [0, 47] \tag{4.16}$$

$$s_{i+48} := L(s_i, s_{i+1}, ..., s_{i+47}) \oplus n_{T,i} \oplus u_i, \quad \forall i \in [0, 31] \tag{4.17}$$

$$s_{i+80} := L(s_{i+32}, s_{i+33}, ..., s_{i+79}) \oplus n_{R,i}, \quad \forall i \in [0, 31] \tag{4.18}$$

$$s_{i+112} := L(s_{i+64}, s_{i+65}, ..., s_{i+111}), \quad \forall i \in \mathbb{N} \tag{4.19}$$

Definition 2 — Keystream Generation Similarly, we define the keystream bit b_i at time i by

$$b_i := f(s_i s_{i+1} ... s_{i+47}), \quad \forall i \in \mathbb{N} \tag{4.20}$$

Definition 3 — Encryption of N_R, A_R and A_T Following common notation for cryptography, we use curly brackets {} to denote encryption.

$$\{n_{R,i}\} := n_{R,i} \oplus b_{i+32}, \quad \forall i \in [0, 31] \tag{4.21}$$

$$\{a_{R,i}\} := a_{R,i} \oplus b_{i+64}, \quad \forall i \in [0, 31] \tag{4.22}$$

$$\{a_{T,i}\} := a_{T,i} \oplus b_{i+96}, \quad \forall i \in [0, 31] \tag{4.23}$$

Definition 4 — Special Keystreams We also group the above keystream bits together using the following notation.

$$ks1 := b_{32}b_{33}...b_{63} \tag{4.24}$$

$$ks2 := b_{64}b_{65}...b_{95} \tag{4.25}$$

$$ks3 := b_{96}b_{97}...b_{127} \tag{4.26}$$

4.3.2 Parity Bits

The ISO-14443-A standard specifies that every byte sent is followed by a parity bit. A parity bit is a bit that is added to ensure that the number of bits with value of one in a given set of bits is always even or odd. Parity bits are used as the simplest error detecting code. An even parity bit is set to 1 if the number of ones in a given set of bits is odd (making the total number of ones, including the parity bit, even). An odd parity bit is set to 1 if the number of ones in a given set of bits is even (making the total number of ones, including the parity bit, odd). The ISO-14443-A standard specified odd parity to be used.

The MIFARE Classic computes parity over the plaintext instead of the ciphertext. There are four parity bits for each 32-bit word. Moreover, the parity bit is sent encrypted with the keystream bit that is also used to encrypt the next bit of the plaintext.

Definition 5 — Parity We define the parity bits p_j of the N_R and A_R messages, and their encryption $\{p_j\}$ as follows (the $\oplus 1$ at the end signify odd parity):

$$p_j := n_{R,8j} \oplus n_{R,8j+1} \oplus ... \oplus n_{R,8j+7} \oplus 1, \quad \forall j \in [0, 3] \tag{4.27}$$

$$p_{j+4} := a_{R,8j} \oplus a_{R,8j+1} \oplus ... \oplus a_{R,8j+7} \oplus 1, \quad \forall j \in [0, 3] \tag{4.28}$$

$$\{p_j\} := p_j \oplus b_{8j+8}, \quad \forall j \in [0, 7] \tag{4.29}$$

We will make use of the information from Definition 1 to 5 extensively when we develop our tools to communicate and authenticate with the reader or tag. With this knowledge, we are able to discuss the weaknesses of the Crypto1 algorithm that will be targeted in our attacks.

4.4 Weaknesses

In Section 2.3 we briefly listed the weaknesses that researchers have discovered of the MIFARE Classic chip. In this section, using the knowledge from this chapter of the Crypto1 cipher, we will be able to describe clearly the seven key weaknesses of the MIFARE Classic chip (adapted from [Ver09a]).

In general, the weaknesses can be classified under the following categories.

1. Weak pseudo-random number generator — This is the most critical weakness discovered of the MIFARE Classic. Many of the attacks target this directly, or a combination of this weakness and other weaknesses.

2. Weak cryptographic cipher — This category includes weaknesses relating to the stream cipher and the filter generator.

3. Weak communication protocol — This category includes weaknesses that are not due to the cipher itself, but the inappropriate use of the cipher in the communication.

4. Weak implementation — This includes weaknesses in the usage of the MIFARE Classic, and even though it has nothing to do with the chip *per se*, there is nothing to stop the adversary from making use of the weakness and attack the chip.

4.4.1 Weakness 1: Low Entropy of Random Generator

We start with the most critical weakness of the MIFARE Classic, which is the weakness in the pseudo-random number generator. We see from Equation 4.6 that the generator is only 16 bits wide, which results in an entropy of 2^{16} or 65,536. This is considered to be extremely low, and according to [NESP08], it takes only 0.6 seconds for the internal LFSR to generate all 65,536 possible nonces and wrap around. This is made worse by the fact that the generator will reset to a known state every time the tag starts operating. The adversary can easily wait a fixed number of clock cycles to elapse since the tag is powered up before requesting for the nonce. It was reported in [Ver09a] that it is experimentally possible to get the same nonce every 30 ms using the Proxmark 3.

4.4.2 Weakness 2: Only Odd State Bits Used to Generate Keystream

This is a weakness of the cryptographic cipher. From Equation 4.1, we see that the inputs to the filter function are only on the twenty odd-numbered bits, i.e. 9^{th}, 11^{th}, ..., and 47^{th} bits. By using a Divide-and-Conquer technique (a technique commonly used in cryptanalysis), we can first split the even and odd bits into two groups, and focus first on the odd group. The odd 9^{th} to 47^{th} bits are used to generate a keystream bit. Then, after two shifts to the LFSR, the 19 original bits in position 11 to 47 and a new bit (position 49^{th}) added at the back becomes the "new" 20-bit input to the filter function. They are then used to generate the next keystream bit. These 21 bits (9, 11, ..., 49) are in fact used to generate two consecutive keystream bits. If we know the values of two consecutive keystream bits, we can narrow down the possibilities for the 21 bits by eliminating those that do not generate the right keystream bits. Similarly we can repeat the process for the even group. We make use of this technique in our attack in Section 6.3.

4.4.3 Weakness 3: Leftmost LFSR Bit Not Used By Filter Generator

This is a weakness of the cryptographic cipher. From figure 4.3 we see that the 20 bits used to generate the keystream bit at each clock cycle is distributed over the last 40 bits. The first nine bits from position 0 to 8 are not used at all. This allows the adversary to perform an attack that is actually the reverse of the filter generation function to "rollback" the LFSR state bit by bit. It is possible to repeat the rollback enough number of times to recover the initial state of the LFSR, which we know from Section 4.3.1 to be the secret key. We describe below how it is possible to exploit this weakness.

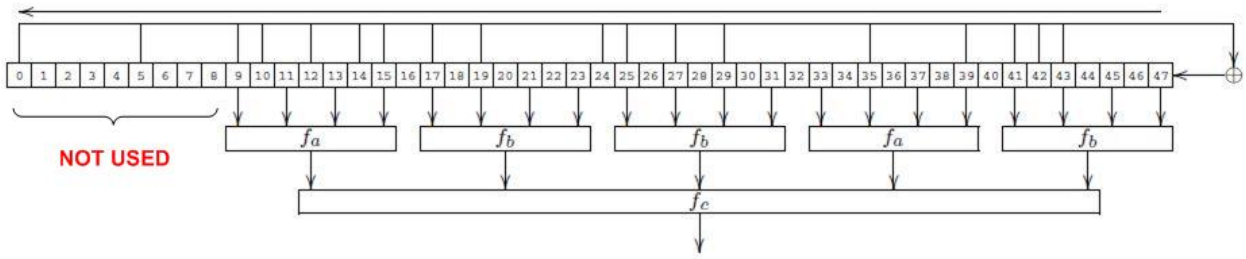

Figure 4.3: Leftmost bits not used by filter generator

Suppose we know the LFSR state immediately after N_R has been fed into the LFSR (we recall from Section 4.3.1 that the LFSR is fed with $(uid \oplus N_T)$ first, and then subsequently with N_R). First we shift the LFSR to the right. The rightmost bit will drop off, and we set the leftmost bit to an arbitrary value r. Then we use the filter generator to compute the keystream bit used to encrypt the bit $n_{R,31}$. Since this leftmost bit r that we have just inserted does not participate in the filter generation, the choice of r does not matter. With the keystream bit that we have just obtained, we take its exclusive-or (XOR) with $\{n_{R.31}\}$ to recover $n_{R,31}$. We can then use it to set r to the correct value using the feedback function of the LFSR (see Equation 4.18). We repeat this procedure 31 more times to recover the LFSR state before N_R is shifted in, and 32 more times to recover the secret key before $(uid \oplus N_T)$ is shifted in.

4.4.4 Weakness 4: Statistical Bias in Cipher

This is again a weakness of the cryptographic cipher. Courtois investigated in [Cou09] on the impact of varying a few bits of $\{N_R\}$ on the generation of keystream $ks1$ with Crypto1. In his experiments, he fixed the first three bytes of $\{N_R\}$ and varied only the last few bits of $\{N_R\}$. His results showed that with a probability of 0.75, the keystream $ks1$ is independent of the last three bits of $\{N_R\}$, which in fact the probability should ideally be 0.5. He also made use of this weakness in his card-only attack in [Cou09].

4.4.5 Weakness 5: Parity Weakness and Reuse of One-time Pad

This is a weakness in the communication protocol. The ISO-14443-A standard defines that every byte sent is followed a parity bit. However, the MIFARE Classic calculates the parity bit over the plaintext, instead of the ciphertext that is sent over the air. The internal LFSR does not shift when the parity bit is encrypted. This means that both the parity bit and the first bit of the next plaintext byte are encrypted with the same keystream bit (see Figure 4.4). This violates the property whereby a one-time pad should not be reused.

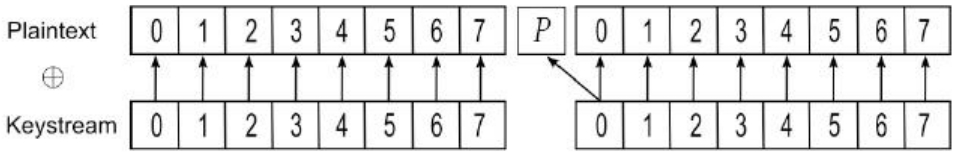

Figure 4.4: Reuse of One-time Pad

4.4.6 Weakness 6: Information Leak from Parity

This is a weakness in the communication protocol. During authentiction, when the reader sends $\{N_R\}$ and $\{A_R\}$, the tag checks the parity bits before checking the correctness of A_R. If one of the eight parity bits is incorrect, the tag does not respond. However, if all eight parity bits are correct, but the response A_R is incorrect, the tag will respond with a 4-bit error code `0x5` indicating a transmission error. Moreover, this 4-bit error code is sent encrypted. This happens

even though the reader has not authenticated itself and hence cannot be assumed to be able to decrypt the message. If we combine the error code `0x5` with its encrypted version, we can recover four keystream bits.

4.4.7 Weakness 7: Deploying Product with Default Keys

This is a weakness in the implementation. Most of the time, chip manufacturers ship their chips with default keys to facilitate easy testing for the companies that do system integration for clients. These default keys are well documented, and some well known ones are listed in Table 4.1.

FFFFFFFFFFFF	A0A1A2A3A4A5	B0B1B2B3B4B5	4D3A99C351DD
1A982C7E459A	000000000000	D3F7D3F7D3F7	AABBCCDDEEFF

Table 4.1: Some Default Keys In Use

This form of weakness almost always occurs, no matter how secure the system is. Some system integrators will choose to ignore the documentation warning them of these default keys and deploy products with them. Our Imperial College ID card is one such product using a default key.

We leave the weaknesses for now and move on to the next chapter, where we will describe the tools that we have used in our attacks.

Chapter 5

Attack Tools

Before we introduce our attacks on the MIFARE Classic, we need to first describe the tools that we are using. There are already several tools available for us to use, since RFID analysis has been conducted by researchers ever since it was invented. One tool that we will definitely need is a reader device that can communicate with the MIFARE Classic. They range from high-end sophisticated protocol analysers such as the Proxmark 3 and the OpenPCD, to low-end simple readers such as the ACR122 reader from Touchatag. Due to the requirements of our attacks, we chose both the Proxmark 3 and the ACR122 reader as our hardware tools for reasons that we will explain in the following sections.

5.1 Proxmark 3

Proxmark 3 is a device developed by Jonathan Westhues [Wes09] that enables the sniffing of RFID traffic and the reading and cloning of RFID tags. As the name suggests, it is the third generation of the proximity card sniffer he has developed. It is a powerful general purpose RFID tool at the size of a credit card and designed to snoop, listen and emulate everything from Low Frequency (125kHz) to High Frequency (13.56MHz). Its flexible design allows one to program enhancements to improve its capabilities. Gerhard de Koning Gans and Roel Verdult from Radboud University Nijmegen, who had been working on the analysis of the MIFARE Classic chip, identified this device to be ideal for their work and extended it with ISO-14443-A capabilities. These extensions, like many others, are available in the public domain at [dV08]. **However we emphasise that none of the attacks they have designed are available.**

5.1.1 Hardware

The Proxmark 3 supports both low frequency (125 kHz — 134 kHz) and high frequency (13.56 MHz) signal processing. This is achieved by implementing two parallel antenna circuits that can be used independently. Both circuits are connected to a 4-pin Hirose connector which functions as an interface to an external loop antenna. For the purpose of behaving like a reader, it is possible to drive the antenna coils with the appropriate frequency. This is not necessary when it is used for sniffing or when emulating a card, where in these cases, the field is generated by the reader.

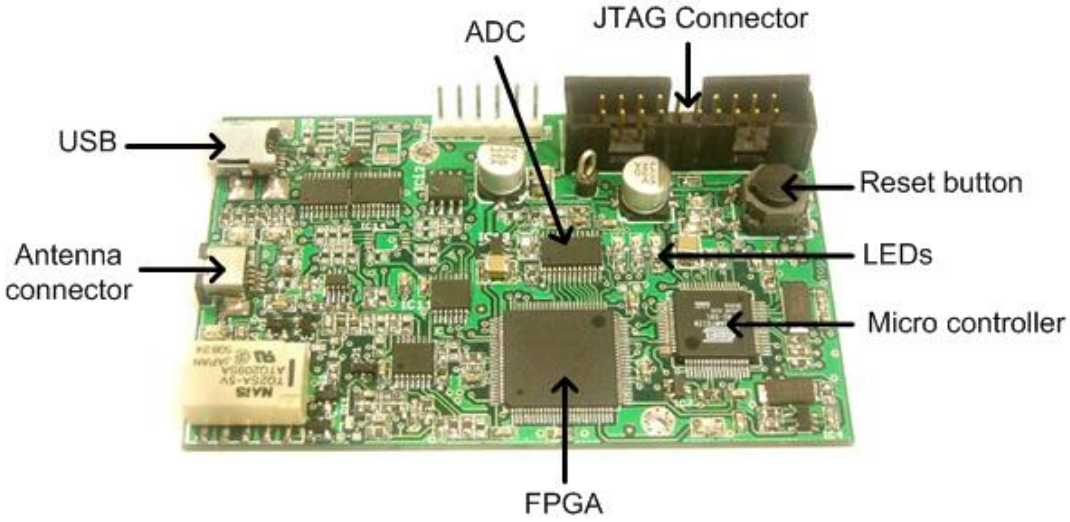

Figure 5.1: Components of Proxmark 3

Figure 5.1 shows the key components of the Proxmark 3. The analog signal that is received from the antenna circuit is fed into an 8-bit Analog to Digital Converter (ADC). The ADC then delivers 8 output bits in parallel which represent the current voltage retrieved from the field. The 8 output pins from the ADC are connected to 8 input pins of the Field Programmable Gate Array (FPGA). An FPGA has a great advantage over a normal micro controller as it is able to emulate any hardware that the user programs it to be. A hardware description can be compiled and flashed into an FPGA. Because basic arithmetic functions can be performed fast and in parallel by an FPGA it is faster than an implementation on a normal micro controller. Only a dedicated hardware implementation would be faster but this lacks the flexibility of an FPGA. As there is no display screen on the Proxmark 3, the user can only rely on three LEDs (red, yellow and green) that can be controlled to display the current status.

The FPGA has two main tasks. The first task is to demodulate the signal received from the ADC and relay this as a digital encoded signal to the ARM micro controller. Depending on the task, this might be the demodulation of a 100% Amplitude Shift Keying (ASK) signal from the reader or the load modulation of a card. The encoding schemes used to communicate with the ARM are Modified Miller for the reader signal and Manchester encoding for the card signal. The second task is to modulate an encoded signal that is received from the ARM into the field of the antenna. This can be for both the encoding of reader messages or card messages. For reader messages the FPGA generates a electromagnetic field on high power and drops the amplitude for short periods. We refer to the discussion in Section 2.1.2 on ISO-14443-A communication.

The ARM micro controller implements the Transport layer protocol. First it decodes the samples received from the FPGA. These samples are stored in a Direct Memory Access (DMA) buffer. The samples are binary sequences that represent whether the signal was high or low. When the Proxmark 3 is in sniffing mode this is done for both the Manchester and Modified Miller encodings at the same time. Whenever one of the decoding procedures returns a valid message, this message is stored in another buffer (BigBuf) and both decoding procedures are set to an un-synced state. It is not possible to stream the recorded messages in real-time to the PC as the ARM processor does not have a proper real-time operating system. Even though the BigBuf is limited to the available memory on the ARM, its size is programmable. When the BigBuf becomes full with recorded messages, the function normally returns. Another function call from the client is needed to download the BigBuf contents to the computer.

We had purchased a ready-built Proxmark 3 from [pro09] at approximately £280, but the HF antenna still needs to be built. There are step-by-step instructions available at [Ver09b] on how

to build one using a USB-Hirose cable, including the information on the length of the cable, how much wire to strip, and the number of coils needed to achieve a proper antenna gain. Figure 5.2 shows our Proxmark 3 with the custom made HF antenna.

Figure 5.2: Proxmark 3 and HF Antenna

5.1.2 Firmware & Client Program

The Proxmark 3 firmware consists of three components. The first component is the bootloader. It functions almost exactly like the BIOS of the PC. Upon power up, it handles the bootstrapping process, checks LEDs, and loads the next component of Proxmark 3, which is the operating system of the ARM processor. The operating system is the "brain" of the Proxmark 3. It is where we programs the logic behind each function that it provides. The operations are programmed in C, but with very much limited library support. The third component is the FPGA image. This is the description of the hardware that the user wants the FPGA to emulate.

To produce these three components that make up the firmware, there is a special compiling environment to use, which is available for download, together with the source code, at [dV08]. The source code is well organised into folders according to the function that it provides. For example, all source code for the ARM micro controller can be found in the `armsrc` folder. Compiling involves running the batch scripts in the `cockpit` folder, and three S19 files (bootrom.s19, osimage.s19 and fpgaimage.s19) corresponding to the three components mentioned above are created.

The compilation process also produces a client program (`prox.exe`). The main purpose of the command-line client program is to upload the S19 files one after the other to the Proxmark 3, by specifying each S19 file as an argument. It also has a graphical user interface (GUI) that can be invoked for interaction with the Proxmark 3. Figure 5.3 shows the GUI environment with the output from **tune** command.

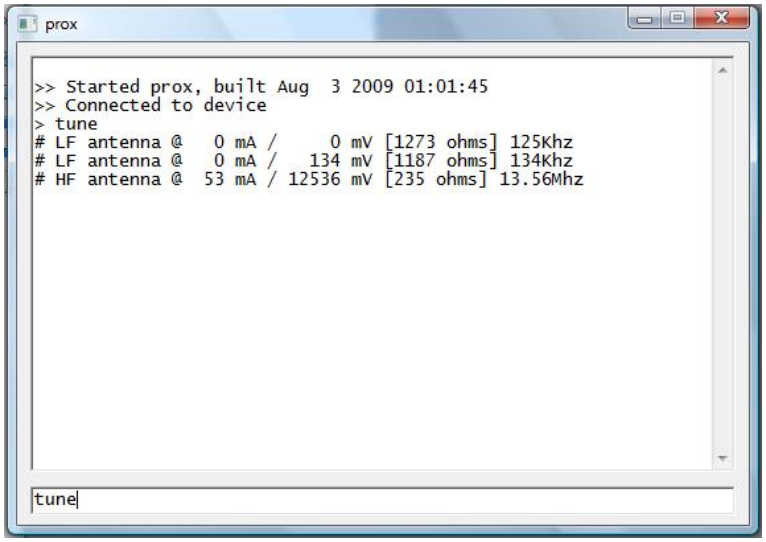

Figure 5.3: Prox GUI

There are several commands already present in the latest firmware (version 20090409). The online community constantly releases new commands to extend the capabilities of the Proxmark 3, which by now far exceeds what the original developer had envisioned. We highlight here important commands that we either use directly or based our new commands on.

- **tune** — This command serves many functions. It allows the user to tune the antenna to the right voltage so that it can detect HF and LF tags in the vicinity. It also allows the user to tell if an unknown tag is a HF or LF one.
- **hi14alist** — The command downloads the content stored in the memory buffer (BigBuff), as mentioned earlier.
- **hi14areader** — The Proxmark 3 will act as a reader by generating a field and using 100% ASK and Modified Miller encoding to communicate with a tag. The responses from the tag are retrieved after this by issuing **hi14alist**.
- **hi14asnoop** — The Proxmark 3 will go into sniffing mode to capture communication between a reader and a nearby tag. Messages from both directions are captured and stored in BigBuff.
- **hi14asim** — The Proxmark 3 will simulate a ISO-14443-A tag with the given UID. Both **hi14areader** and **hi14asim** will handle only the anti-collision phase with a legitimate tag or reader respectively.

5.2 Libnfc

Libnfc is an open source library in C for Near Field Communication (NFC) [Ver08]. NFC technology is a simple extension of the ISO-14443 standard that combines the interface of a smartcard and a reader into a single device. An NFC device can communicate with both existing ISO-14443 smartcards and readers, as well as with other NFC devices, and is thereby compatible with existing contactless infrastructure already in use for public transportation and payment. The purpose of the libnfc library is to provide developers a way to work with NFC hardware at a higher level of abstraction at no cost. Released recently in February 2009 and currently at version 1.2.1, the set of supported features is still growing. Some important functions include support for ISO-14443-A modulation, MIFARE Classic protocol implementation and ability to transform any USB-based NFC device into a reader or tag. The full API can be found at [Ver08]. Unlike the Proxmark 3 which requires a special compiling environment, libnfc can be compiled using Windows Visual Studio 2005 or later versions. The package also comes with some pre-built binaries and their source code to help us get started. We highlight here three binaries that we use directly, or developed using the source code as a working base.

- **nfc-anticol** — This demonstration tool negotiates the anti-collision phase with a ISO-14443-A tag. We have extended it to include the authentication phase for the MIFARE Classic.
- **nfc-list** — This tool detects the tag type and completes an anti-collision phase with it. We used it to identify an unknown tag before running any attack.
- **nfc-mftool** — We use this tool to dump the contents of the Classic chip. It makes use of a binary file type which is an image of the Classic chip. This image contains the data, keys and access bits on their defined off-set. A dump file is always 4KB, even when a 1KB tag is dumped. The dump file follows exactly the memory layout as described in [NXP08b].

Compared to the Proxmark 3, programming using libnfc is much simpler for beginners learning to develop RFID applications. Furthermore, libnfc supports many existing (and inexpensive) commercial readers so there is no need to custom-build hardware. However, libnfc and Proxmark 3 have one common feature — no attack software written in libnfc has been released.

5.2.1 Touchatag (ACR122) Reader

The Touchatag reader (see Figure 5.4) is based on the ACR122(U) NFC reader from Advanced Card Systems Limited. Compared to the Proxmark 3, the Touchatag reader is inexpensive (since it does not house an FPGA) and is easily available for approximately €30. It is fully compatible with libnfc. It is USB-powered, but unlike the Proxmark 3, it does not process any of the signals (as it does not have an on-board processor) and sends received messages directly back via the USB interface.

Figure 5.4: Touchatag reader and Oyster card

5.3 Crapto1

Crapto1 is a C library that not only implements the Crypto1 cipher in software, but also provides some functions supporting the attack from [GdM+08]. It can be downloaded from [bla09]. Currently at version 2.4, it has recently been extended to provide support for attacks in [Cou09] and [GvVS09]. However since it does not provide a user interface program or a wrapper, users will have to implement one themselves. Moreover, it is simply a software library and does not provide any form of interaction with hardware readers or tags. There is no support for the source code at all. Other than scarce comments in the code, there is no online documentation provided. Hence it is often necessary to trace the C code by hand to know exactly what it is doing. We describe here the Crapto1 functions that we have used to implement the Crypto1 cipher.

- `crypto1_create(key)` — This command initialises the LFSR with the given key.
- `crypto1_get_lfsr` — It will return the current LFSR state.
- `crypto1_word` — It will return a 32-bit keystream, and updates the LFSR if necessary.
- `prng_successor` — It implements the *suc* function as described in Equation 4.7.

We also describe the Crapto1 functions that implements the attack from [GdM+08].

- `lfsr_recovery64` — This function produces the LFSR state after the given 64 keystream bits are generated.
- `lfsr_rollback` — This function is usually called after `lfsr_recovery64`. This implements the attack for the weakness reported in Section 4.4.3. It will rollback the LFSR 32 times to get the previous state.

We wrote an extension to the `lfsr_rollback` function, called the `lfsr_rollback_n`, which has an additional input n. This n tells the function to perform rollback n number of times.

We decided to use Crapto1, even though there is an obvious lack of support, because given the time constraints of our project, we felt it would be impossible to implement our own software implementation of Crypto1. Furthermore we felt that we can achieve more in other areas with the time saved.

Even though the tool that we have chosen — the Proxmark 3, libnfc and Crapto1 — have functionality that already will work for us out of the box, we still have plenty of work to do. In the next chapter, we will describe the attacks that we have implemented with the help of these tools.

Chapter 6

Attacks

After we have evaluated the weaknesses in Section 4.4, and introduced the tools that we can use to attack the MIFARE Classic chip in the previous chapter, we will in this chapter describe in greater detail the three attacks that we have implemented. These three attacks can be carried out in theory on any application using the MIFARE Classic. We will begin by describing each attack in terms of the following characteristics

- Weakness exploited, if any
- Attack tool used
- Difficulties met when carrying out attack, and our solutions

The first and last attacks are focussed on getting the secret keys from the tag, but with varying extent and impact. The second attack is different from the other two as it is focussed on attacking the application after we have obtained the keys and data.

6.1 Key Recovery from Intercepted Authentication

The objective of this attack is to recover the key used to encrypt an intercepted authentication phase between a tag and a genuine reader. The key recovery is done offline after the communication has been successfully intercepted. The extent of the attack is that the key of the sector involved in the authentication can be recovered. The impact of this is that all data from that sector can hence be obtained with this key.

Weakness exploited

This attack makes use of two key weaknesses in the Crypto1 stream cipher. Firstly it allows us to make use of a divide-and-conquer method to recover the LFSR state due to the weakness that only odd bits are used in the filter generator (see Section 4.4.2). Secondly, with the LFSR state, we can make use of the rollback attack to recover the initial state of the LFSR due to the weakness that the first nine bits of the LFSR are not used by the filter generator (see Section 4.4.3).

Attack tool used

The tool that we are using is the Proxmark 3. It had been extended earlier by Gerhad de Koning Gans in [dK08] for sniffing ISO-14443-A communication. We ran the `hi14asnoop` command on the windows client to start the interception on the Proxmark 3, and successfully captured the communication between a Imperial College ID card and a genuine reader (see Figure 6.1). Figure 6.2 shows a complete trace of a single authentication session captured by the Proxmark 3. However, we have deliberately masked the information necessary to obtain the key.

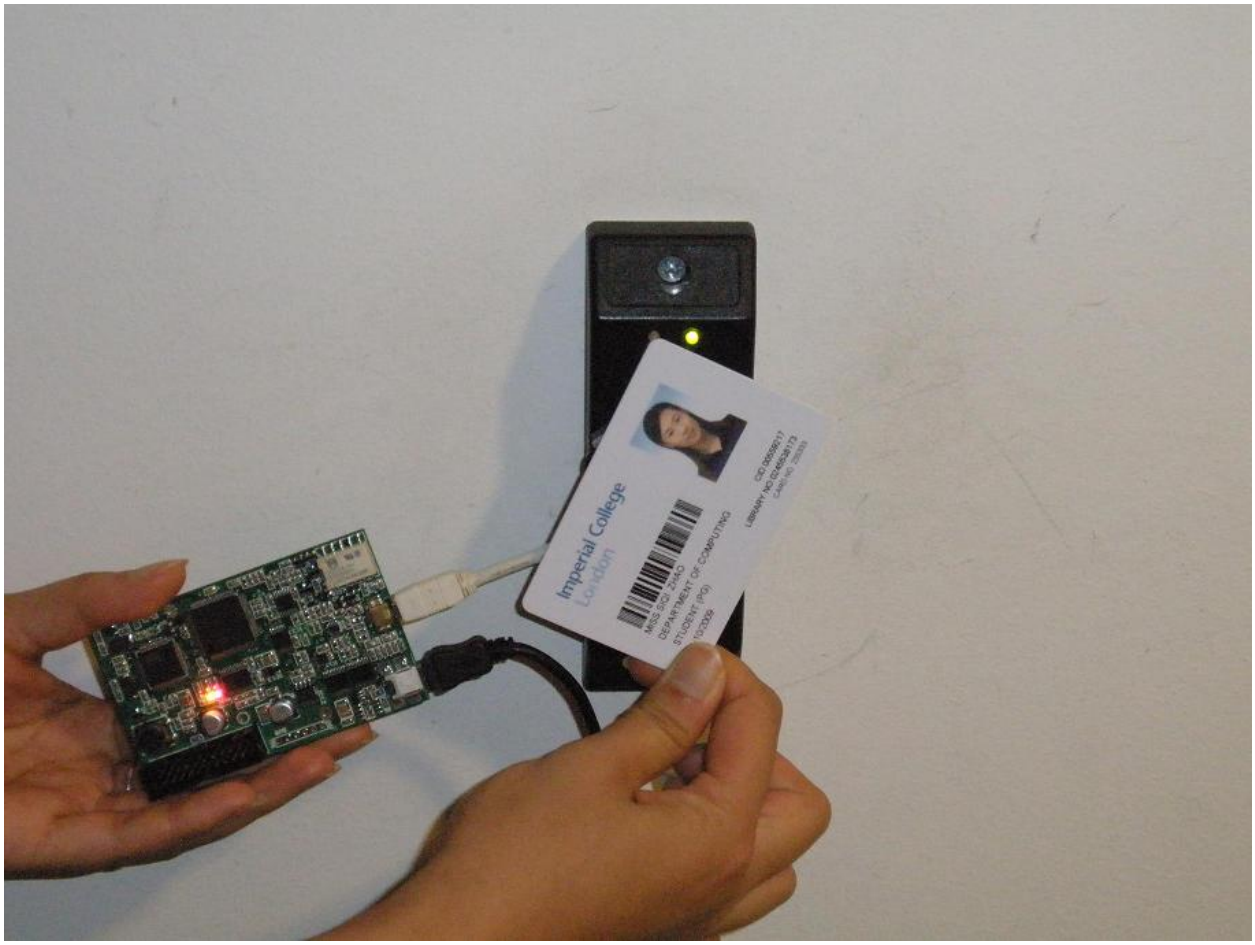

Figure 6.1: Proxmark 3 in sniffing action

```
|+  62958:    :      26
 +     64:    0: TAG 04  00
 +    664:    :      93  20
 +     64:    0: TAG 4a  28  [redacted]
 +   1448:    :      93  70  4a  28  [redacted]  47  0d
 +     64:    0: TAG 08  b6  dd
 +   1464:    :      60  03  6e  49
 +    112:    0: TAG d9  d4  d2  ad
 +   1056:    :      00  60  41  e3  ff  f9  75  2a      !crc
 +  18524:    :      26
 +     64:    0: TAG 04  00
 +    656:    :      93  20
 +     64:    0: TAG 4a  28  [redacted]
 +   1448:    :      93  70  4a  28  [redacted]  47  0d
 +     64:    0: TAG 08  b6  dd
 +   1456:    :      60  04  d1  3d
 +    112:    0: TAG 05  [redacted]
 +   1064:    :      0c  cb  4d  c7  e5  01  8d  25      !crc
 +     62:    0: TAG f3! ee! b0  63
 +    904:    :      58  b3  56  0e      !crc
 +     72:    0: TAG cf! 5f  df  bd! 25! 89  e5! f2! 8e! cd! cc! f9! 0b  29! 4a! 10! 5f! c1      !crc
 +   2120:    :      87  ec  38  44      !crc
 +     72:    0: TAG 2a! 58! f6  b8  04  35  b8  c4  27  dc  05  aa! d9! 74! 1f  41  77  e2!     !crc
 +   2136:    :      ac  c7  d8  61      !crc
 +     72:    0: TAG 5a! 77! 03! a5! c9  00! d5! a4  b3  97! 3c! 77  a8! 34  47  ec! 5f  6e      !crc
 + 108350:    :      26
 +  17717:    :      26
 +     64:    0: TAG 04  00
 +    680:    :      93  20
 +     64:    0: TAG 4a  28  39  b9  e2
 +   1448:    :      93  70  4a  28  39  b9  e2  47  0d
 +     64:    0: TAG 08  b6  dd
 +    968:    :      50  00  57  cd
```

Figure 6.2: Complete trace captured by intercepting IC card communication

After we have obtained a complete authentication trace using the Proxmark 3, we need to perform an offline recovery of the secret key. For that we will use some functions from Crapto1. We recall from Section 5.3 that it does not provide any user interface. Hence, we have written a C program

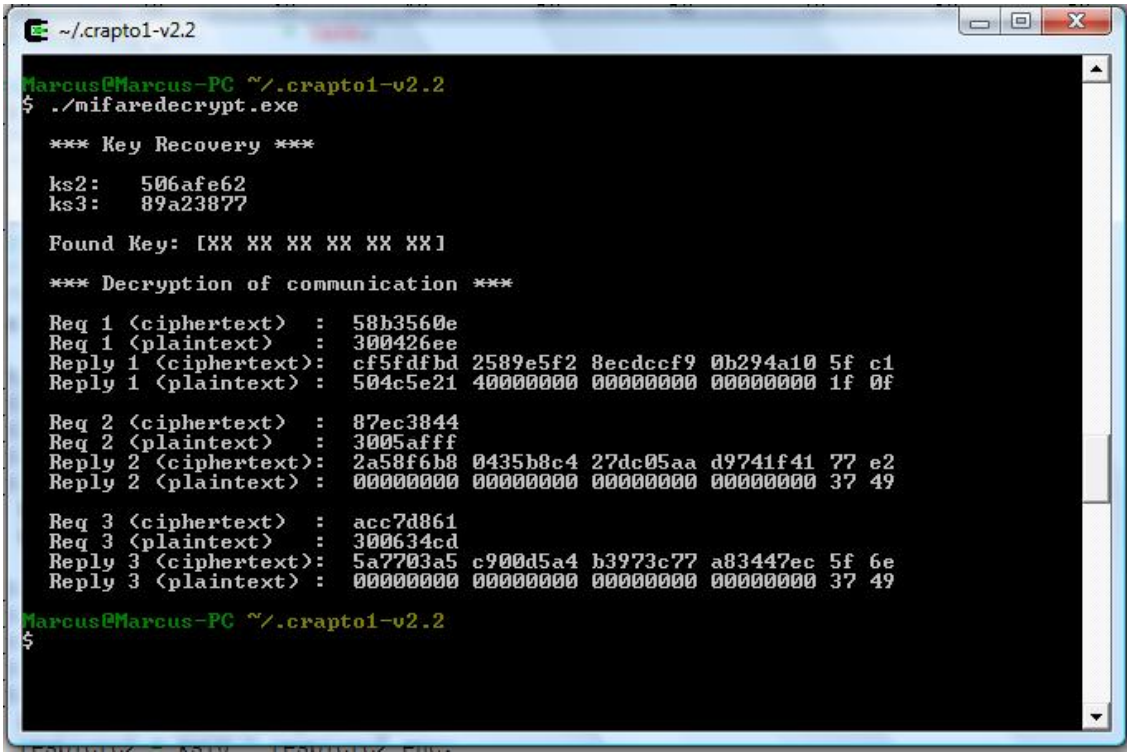

Figure 6.3: Decrypting intercepted communication from Figure 6.2

called `mifaredecrypt.c` that makes use of some Crapto1 functions to recover the sector key that is used to encrypt that communication. With the key and the intercepted nonces, our program next initialises the stream cipher in Crapto1 following the descriptions in Section 4.3.1 to obtain keystreams which are then used to decrypt the rest of the encrypted communicate (see Figure 6.3 for the decryption of an example trace of the Imperial College's ID card).

Difficulties met and our solutions/contributions

Firstly, we noticed that the positioning of the Proxmark 3 HF antenna was an important factor in getting complete intercepts. Even when placed within the proximity of both the card and reader, we occasionally encountered cases where either the antenna did not pick up any tag response at all, or it picked up incomplete tag responses. Our experiments showed that, for best results, the antenna must be placed in between the reader and tag during interception, so that the electronic field directly passes through the HF antenna.

Secondly, while carrying out interception of communication using the Proxmark 3, it needs to be connected to a laptop throughout the duration of the attack for two reasons. One, it draws its power from the USB connection and the intercepted data stored in its internal buffer will be lost once it loses its power. Two, the Windows client needs to be connected to the Proxmark 3 in order to issue the attack command. This drawback prevents us from carrying out our attack inconspicuously.

To overcome this, we modified a version of the firmware to automatically boot the Proxmark 3 into interception mode upon power up. It will capture any communication within its antenna range until it runs out of BigBuff space or when the Reset button is pressed. We next purchased a portable backup power supply that is normally used to charge portable devices via USB, and a dual-power USB Y-cable. By connecting one end of the Y-cable to the backup power supply, it powers the Proxmark 3 for the duration of the attack. After the attack is done and when it becomes convenient for us, we will connect the other end of the Y-cable to a laptop's USB port to

retrieve the BigBuff contents. We will not lose the contents of BigBuff in this way, and we can hide the Proxmark 3 and the power supply in a bag or pouch.

6.2 Card Emulation

The objective of this attack is to emulate a tag and fool the genuine reader into thinking that it is communicating with the actual tag. To perform the card emulation, we first need to obtain a complete communication that normally takes place between the genuine tag and reader, so that we know what the reader is requesting and how the tag will respond. However, we cannot record the communication and simply replay the entire tag responses in our attack, as the reader challenge N_R would be different and the cipher state would hence be different. Secondly we will need the keys and data requested by the reader and program our tool to use the key to encrypt the data for transmission. In principle, we can emulate ANY tag as long as we have the complete communication trace with a genuine reader. We have decided to emulate an Imperial College ID card in this attack.

Attack tool used

We have obtained a complete trace of the communication between an Imperial College ID card and a genuine reader from the first attack. We will use the Proxmark 3 to emulate a valid card. The firmware from [dK08] allows us to emulate a ISO-14443-A tag with any UID using the `hi14asim` function. However, it stops after the anti-collision phase, and does not handle any authentication requests. Using that function as a reference, we designed and wrote a new function `simic` that allows us to emulate an Imperial College ID card and successfully communicate with a valid reader. This new function would

1. Negotiate the anti-collision phase using the UID of the card we are emulating
2. Handle the authentication requests from the reader by deciding what to respond
3. Encrypt and send the data from the requested blocks

The encryption is done using limited code snippets from Crapto1. Even though the Crapto1 software can be used to model the Crypto1 cipher state and generate keystreams for encryption or decryption, the code cannot be compiled directly for the FPGA environment. This is due to the fact that it uses dynamic memory allocations and other object-oriented programming constructs that are not supported on the FPGA. We then had to extract the relevant code and port them into code suitable for the FPGA.

We have successfully tested our attack on several readers throughout the South Kensington campus, and can enter any door using our emulated card as if we have the genuine card.

Difficulties met and our solutions/contributions

There is a slight difference between the way bytes are represented in tools such as Proxmark 3 and the way they are being transmitted over the air. The former writes the most significant bit of the byte on the left, while the latter writes the lease significant bit on the left. Therefore if we want to represent any value in the tool, we have to write each byte reversed. Table 6.1 shows the order of actual 32 bits transmitted over the air, and Table 6.2 shows the actual representation in the Proxmark 3.

0,1,2,3,4,5,6,7	8,9,10,11,12,13,14,15	16,17,18,19,20,21,22,23	24,25,26,27,28,29,30,31
Byte 0	**Byte 1**	**Byte 2**	**Byte 3**

Table 6.1: Representation of bit positions transmitted over the air

7,6,5,4,3,2,1,0	15,14,13,12,11,10,9,**8**	23,22,21,20,19,18,17,**16**	31,30,29,28,27,26,25,**24**
Byte 0	**Byte 1**	**Byte 2**	**Byte 3**

Table 6.2: Representation of actual bit positions in the Proxmark 3

For example, if we want to represent the hexadecimal word `0x12 34 56 78`, we should write it as `0x48 2C 6A 1E` in the tool. We had to take into account this important fact during encryption and decryption. Moreover, as described earlier in Section 4.4.5, the parity bit of a byte is encrypted with the first keystream bit of the next byte. However with the bit ordering as shown above, the keystream bit to use would be the last bit of the next byte. Hence in our representation in Table 6.1, the bits with its corresponding keystream bit reused would be those in bold. If we used the wrong keystream bits, the encrypted parity bits would be incorrect and the genuine reader would immediately send a HALT command.

6.3 Key Recovery Using Card Only

Many of the attacks described so far require access to both a card and a genuine reader. This significantly exposes the attacker to the authorities as, most often than not, gates or doors protected by such authentication methods have additional human or camera surveillance. The risk of being caught by either one discourages the attacker from attempting the attack. Recently however, several key recovery attacks that only require access to a card have been published, namely in [Cou09] and in [GvVS09]. The objective of this attack is to recover all 16 sector keys one by one without the need for a genuine reader. The extent of this attack is hence much larger than the previous attack, and the impact of this is immense. All data from all sectors can now be recovered.

Weaknesses exploited

Our attack originates from the one describe in Section 5 of [Cou09], but with several modifications. It targets several weaknesses of the MIFARE Classic at the same time. Firstly it makes use of the weakness of the low entropy in the pseudo-random number generator (see Section 4.4.1) to obtain a constant tag nonce. Secondly, it makes use of the weakness that only odd bits are used as inputs to the non-linear filter generator as described in Section 4.4.2. Thirdly, it makes use of the statistical bias weakness as described in Section 4.4.4, which is, with a probability of 0.75, the keystream $ks1$ is independent of the last three bits of $\{N_R\}$, when in fact the probability should be 0.5.

Courtois also reported a discovery in his paper in Section 5.3. If $ks1$ does not depend on N_R, the difference of the state of the LFSR during the generation of $ks3$ depends only on the differences in the last byte of N_R. We develop further on this discovery with the following key finding:

OUR KEY FINDING — The difference (XOR) between the odd bits of any two states depends directly on the difference in $\{N_R\}$ and nothing else. This state difference is constant, and can be precalculated and used to deduce the odd bits of the next state if the difference in N_R is the same. In addition, we can deduce the difference in states used to generate each bit of $ks3$ used for encrypting the transmission error message. The state differences are calculated and listed in Table 6.3.

$\{N_R\}$ **Difference**	**State Difference (Odd Bits)**			
	ks3(0)	**ks3(1)**	**ks3(2)**	**ks3(3)**
00000001	0x012F14	0x007658	0x025E29	0x00ECB1
00000002	0x007658	0x025E29	0x00ECB1	0x04BC53
00000004	0x025E29	0x00ECB1	0x04BC53	0x01D962

Table 6.3: State Differences

Attack tool used

We will use the libnfc library to program the attack, and the Touchatag reader to communicate with our tag. Libnfc provides a binary tool called `anticol.exe` that performs only the anti-collision phase with the MIFARE Classic chip. We will use it as a starting point and extend it with out attack, starting with the authentication phase.

Our Attack Descriptions

1. **Stage 1 (nicattack1.exe)** We attempt 1,000 authentications to the tag with a random N_T, random 8-byte ($\{N_R\}$, $\{A_R\}$) ciphertext and 8 random encrypted parity bits. We should get between three to ten cases out of 1,000 where the tag answers with a 4-bit encrypted transmission error message. A file with a list of nonces obtained from the 1,000 authentications is obtained as a result.

2. We sort and group the nonces in the file to determine the number of times each nonce appears in the 1,000 authentications. We choose the nonce that appears most frequently for which the tag answers with a 4-bit encrypted transmission error message earlier.

3. **Stage 2 (nicattack2.exe)** We repeatedly attempt authentications with the tag but only continue if the nonce is the same as our chosen N_T. We will then keep the first 29 bits of $\{N_R\}$ constant, the entire $\{A_R\}$ constant, and the first three encrypted parity bits constant. We will try in order each of the eight possible values for $\{N_R\}$ on three bits and assign random values for the last five encrypted parity bits. We should get a 4-bit reply for each possible combination for $\{N_R\}$.

4. **Stage 3 (nicattack3.exe)** From now, the attack is offline. We first recover the keystream used to encrypt the transmission error message, by simply taking the exclusive-or (XOR) of the ciphertext with the known plaintext `0x5`.

5. Starting with a search space of 2^{21} elements, we remove those that do not generate even bits $ks3(0)_1$ and $ks3(2)_1$, using the filter function in Equation 4.1. This should reduce the search space by four.

6. **Stage 3a** We XOR the remaining elements with the state difference from Table 6.3 that corresponds to the difference in $\{N_{R,1}\}$ and $\{N_{R,2}\}$ to get the possible next states.

7. **Stage 3b** We next remove those possible next states that do not generate $ks3(0)_2$ and $ks3(2)_2$. Again the search space should be reduced by four.

8. We repeat **Stage 3a** and **3b** for the remaining six data points. We should finally get approximately 2^5 possible states.

9. We repeat the steps above for the odd bits $ks3(1)_1$ and $ks3(3)_1$, $ks3(1)_2$ and $ks3(3)_2$, and so on, to get the other 2^5 possible states.

10. With the two lists of 2^5 possible states, we combine them to get a list of 2^{10} states of 42-bits wide. We then extend the list to 2^{16} states of 48-bits wide.

11. We rollback each of the 2^{16} possible keys and check if it generates all eight sets of parity bits, stopping immediately once we find one that does so.

Difficulties met and our solutions/contributions

Firstly, without dedicated hardware such as the Proxmark 3, getting the same tag nonce repeatedly is difficult to achieve. Our attack, which was written using libnfc and ran on the Touchatag reader, was able to get a tag nonce to repeat approximately 30% of the time. To overcome this problem, we authenticate 1,000 times and choose the tag nonce with the highest occurrence rate from Stage 1 for use in Stage 2. Even though the timing that we have is longer than that reported by Courtois, our Stage 1 attack takes about three minutes, Stage 2 attack takes less than 2 minutes and Stage 3 attack is almost instantaneous. Hence overall our attack to get a single secret key takes less than five minutes. Most importantly, our attack does not require expensive dedicated hardware.

Secondly, we have to be aware of the bit ordering challenge reported in the previous section in developing our attack here. The last three bits (29^{th}, 30^{th} and 31^{st}) of N_R that we need to vary are in fact the first three bits of the last byte. Similarly, the keystream used to encrypt the transmission error message are reversely ordered $ks3(3)ks3(2)ks3(1)ks3(0)$. We will use the following notation in Table 6.4.

Last three bits of C_3	**ks3 Bit Order**			
000	$ks3(3)_1$	$ks3(2)_1$	$ks3(1)_1$	$ks3(0)_1$
001	$ks3(3)_2$	$ks3(2)_2$	$ks3(1)_2$	$ks3(0)_2$
010	$ks3(3)_3$	$ks3(2)_3$	$ks3(1)_3$	$ks3(0)_3$
011	$ks3(3)_4$	$ks3(2)_4$	$ks3(1)_4$	$ks3(0)_4$
100	$ks3(3)_5$	$ks3(2)_5$	$ks3(1)_5$	$ks3(0)_5$
101	$ks3(3)_6$	$ks3(2)_6$	$ks3(1)_6$	$ks3(0)_6$
110	$ks3(3)_7$	$ks3(2)_7$	$ks3(1)_7$	$ks3(0)_7$
111	$ks3(3)_8$	$ks3(2)_8$	$ks3(1)_8$	$ks3(0)_8$

Table 6.4: Permutation of the "last" three bits and corresponding ks3 bit ordering

We have successfully implemented these three attacks, and carried them out on two important case studies. We will share our findings in the following chapter.

Chapter 7

Case Studies

After we have implemented the three attacks as described in the previous chapter, we test them on two different applications that use the MIFARE Classic chip. The first is an access control application where the tag contents are used to identify a person to the system and grant that person access through doors. The second is a ticketing application for public transportation, where the tag functions as a stored value ticket. Not only does it store the remaining credit, it also keeps a certain number of historical transactions. We will describe our findings, and highlight any weaknesses with the implementation for each case study below.

7.1 Imperial College ID Card

The Imperial College ID card is a photocard that is issued to every staff and student. Information such as the owner's name, department, role, College Identifier (CID) and expiry date are printed on the card. It provides two different options for automated authentication - magnetic stripe (contact-based) and RFID (contactless). Each card has an embedded MIFARE Classic 1K chip that stores information necessary to authenticate itself with the readers. Figure 7.1 shows the communication that takes place during authentication. This information was obtained by carrying out our first attack as detailed in Section 6.1.

7.1.1 Our Findings

We analysed the ATQA and SAK values from the tag according to Section 3.3.3. The ATQA is `0x04 00` and the SAK is `0x08`, so we are certain it is a MIFARE Classic 1K card from NXP. We next looked at the decrypted communication and we were surprised that only data blocks 4, 5 and 6 are requested. Since they are found within the same sector, we only need to recover one sector key and the contents of these blocks in order to emulate the card. After we have successfully done all that, we analysed at the access bits stored in Sector Trailer 1. The 6^{th}, 7^{th} and 8^{th} bytes contain the access bits and their inverse, and are found to represent the values `0x78`, `0x77` and `0x88` respectively. The bits are identified accordingly, as shown in Table 7.1.

Binary(0x78)	0	1	1	1	1	0	0	0
Access bit	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
Binary(0x77)	0	1	1	1	0	1	1	1
Access bit	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
Binary(0x88)	1	0	0	0	1	0	0	0
Access bit	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$

Table 7.1: Access bits in Sector Trailer 1(IC Card)

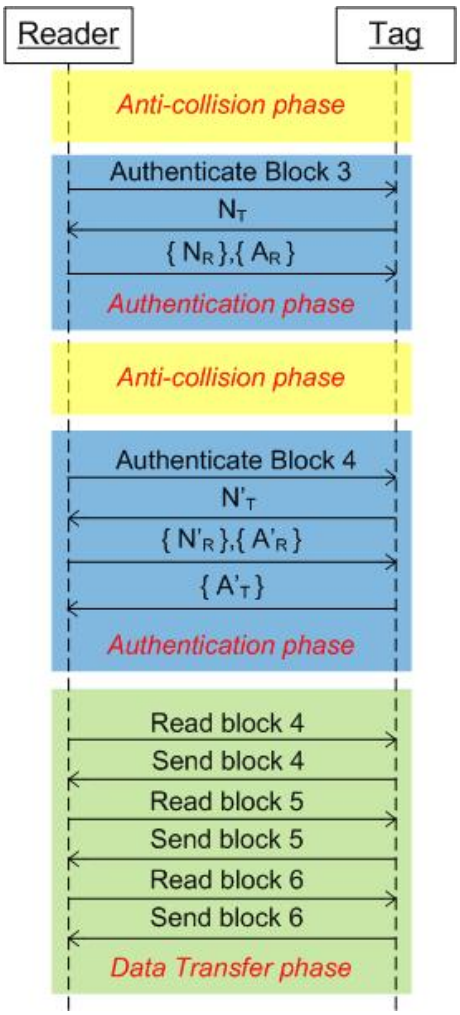

Figure 7.1: Messages passed during authentication for IC card

We next looked up Tables 3.2 and 3.3 based on these bits to find the access conditions, as shown in Tables 7.2 and 7.3.

Access bits			**Access condition for**					
			KEY A		**Access bits**		**KEY B**	
$C1_3$	$C2_3$	$C3_3$	**Read**	**Write**	**Read**	**Write**	**Read**	**Write**
0	1	1	never	key B	key A or B	key B	never	key B

Table 7.2: Access conditions for Sector Trailer 1(IC Card)

Access bits			**Access condition for**				**Application**
C1	**C2**	**C3**	**Read**	**Write**	**Increment**	**Decrement, Transfer, Restore**	
1	0	0	key A or B	key B	never	never	read/write block

Table 7.3: Access conditions for Data Blocks 0, 1 and 2 of Sector 1(IC Card)

The access conditions clearly indicate that the data blocks are used as read/write blocks, which was what we expected in an access control application. Both keys A and B cannot be read directly (even though we have managed to crack key A) and key B will be needed to change any values in the data blocks (having key A is insufficient in this case).

In Section 4.4, we described a category of weakness called implementation weakness. We point out here three such weaknesses that we have found.

Weakness 1: Default Keys

The keys of sectors not involved in the authentication have a default value `0xFF FF FF FF FF FF`. We discovered this when we tried using any of the default keys from the manufacturer documentation to dump the card content using libnfc's `nfc-mftool`. While this does not seem to be a major weakness in itself (since they are not utilised in authentications), the attacker can deduce that these sectors are not used and hence focus efforts on recovering sector keys that are not the default value.

Weakness 2: Same Key for All Cards

The sector that contains the data blocks used for authentication is protected by the same key for all cards issued. It has remained unchanged since the launch of the system. It is the same key in cards for students (for which access rights given should be minimum) and in cards for security staff (for which access rights given should be maximum). This means that the attacker, once she has that key, is able to obtain any victim's identity using the right equipment just by walking past the victim.

Weakness 3: All-zero Blocks for Authentication

Data from blocks 0x04, 0x05 and 0x06 are sent from tag to reader during authentication. However, blocks 0x05 and 0x06 contain all zeros for all cards. Similarly, block 0x04 contains all zeros, except for the first five bytes. Even though these zeros are encrypted using keystream bits and seem random in sniffed traffic, one may question the purpose of sending encrypted zeros. If the attacker knows this weakness, she simply needs to obtain the first five bytes of block 0x04 from a victim.

7.1.2 Recommendations

Bearing in mind the cost of changing the entire access control system, including readers, backend software and databases, to one that is more secure, and the overheads needed to replace issued cards, we propose the following short term improvements to the mitigate the weaknesses above.

1. Blocks should be chosen randomly for authentication, instead of using only blocks 0x04, 0x05 and 0x06. This should significantly slow down the attacker as she has to obtain all 16 sector keys and data from all blocks.

2. All sector keys should be assigned non-default values, even if a sector is not involved in authentication.

3. All data blocks should be assigned non-zero 16-byte values.

7.2 Oyster Card

The Oyster Card is a form of electronic ticketing used on public transport services within London. It is a credit-card sized, contactless, stored-value card which can hold a variety of single tickets (called Pay-As-You-Go), period tickets (called Travelcard) and travel permits (such as BusPass) added to the card prior to travel. When passengers enter or leave the transport system, they will pass their cards over readers to deduct funds (for Pay-As-You-Go) or validate (for Travelcard and BusPass). The value in the card can be increased or recharged in person, by direct debit or by online purchase. The maximum amount of credit that can be stored is £90.

The card holds a MIFARE Classic 1K chip with an aerial running along the perimeter of the card. Every Oyster card issued has a unique set of 16 sector keys. We believe the system is asynchronous, i.e. the current balance and ticket data are held electronically on the card first and the data is then transferred to the central database later. The central database is only updated periodically with information obtained from the card by barriers readers and validators. The last ten transactions can be retrieved from the card at the Underground Station readers.

We analysed the ATQA and SAK values from the tasg according to Section 3.3.3, and we got two sets of different results. On the normal Pay-As-You-Go Oyster card, the ATQA is `0x04 00` and the SAK is `0x08`, so we are certain it is a MIFARE Classic 1K card from NXP. But on a Oyster Photocard (cards with a photo and identity associated and used by students for discounted travel), the ATQA is `0x04 00` and the SAK is `0x88`, so we conclude that it is a MIFARE Classic 1K card from Infineon. However this difference has no impact on our attack as chips from both manufacturers are vulnerable to the same weaknesses.

We used our card-only attack to recover all keys from different types of Oyster cards. After dumping out the data block content from these card types for analysis, we discovered that the access bits for all Oyster cards are the same. The access bits in all sector trailers represent the values 0x7F, 0x07 and 0x88, with the exception of those in Sector Trailer 1, which represent the values 0x67, 0x87 and 0x89. We first looked at the access bits in the case of the former (see Figure 7.4).

Binary(0x7F)	0	1	1	1	1	1	1	1
Access bit	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
Binary(0x07)	0	0	0	0	0	1	1	1
Access bit	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
Binary(0x88)	1	0	0	0	1	0	0	0
Access bit	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$

Table 7.4: Access bits in non-sector 1 trailers(Oyster Card)

Again, we looked up Tables 3.2 and 3.3 based on these bits to find the access conditions, as shown in Tables 7.5 and 7.6.

Access bits			**Access condition for**					
			KEY A		**Access bits**		**KEY B**	
$C1_3$	$C2_3$	$C3_3$	**Read**	**Write**	**Read**	**Write**	**Read**	**Write**
0	1	1	never	key B	key A or B	key B	never	key B

Table 7.5: Access conditions for non-sector 1 trailers(Oyster Card)

Access bits			**Access condition for**				**Application**
C1	**C2**	**C3**	**Read**	**Write**	**Increment**	**Decrement, Transfer, Restore**	
0	0	0	key A or B	key A or B	key A or B	key A or B	transport configuration

Table 7.6: Access conditions for Data Blocks 0, 1 and 2 of sectors other than 1(Oyster Card)

These access conditions indicate that even though the MIFARE Classic chip is used in a stored value application, the data blocks are not configured as value blocks. In fact, they are set in transport configuration, meaning **either key A or B can be used to modify them**. We next looked at the access bits in Sector Trailer 1 (see Table 7.7) and their corresponding access conditions (see Tables 7.8, 7.9 and 7.10).

Binary(0x67)	0	1	1	0	0	1	1	1
Access bit	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
Binary(0x87)	1	0	0	0	0	1	1	1
Access bit	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
Binary(0x89)	1	0	0	0	1	0	0	1
Access bit	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$

Table 7.7: Access bits in Sector Trailer 1(Oyster Card)

Access bits			Access condition for					
			KEY A		Access bits		KEY B	
$C1_3$	$C2_3$	$C3_3$	**Read**	**Write**	**Read**	**Write**	**Read**	**Write**
1	1	1	never	never	key A or B	never	never	never

Table 7.8: Access conditions for Sector Trailer 1(Oyster Card)

Access bits			Access condition for				Application
C1	**C2**	**C3**	**Read**	**Write**	**Increment**	**Decrement, Transfer, Restore**	
0	0	0	key A or B	key A or B	key A or B	key A or B	transport configuration

Table 7.9: Access conditions for Data Blocks 1 and 2 of Sector 1(Oyster Card)

Access bits			Access condition for				Application
C1	**C2**	**C3**	**Read**	**Write**	**Increment**	**Decrement, Transfer, Restore**	
0	1	0	key A or B	never	never	never	read-only block

Table 7.10: Access conditions for Data Block 0 of Sector 1(Oyster Card)

The access conditions for Sector Trailer 1 indicate higher restrictions set as compared to other sector trailers, since all bits cannot be reprogrammed and only access bits can be read. Blocks 1 and 2 are the same as the blocks from other sectors, but block 0 is a read-only block. Due to time constraints, we were unable to verify the role of this read-only block. However we make an intelligent guess that it is used as a secondary key to mask the contents of the other data blocks, as described in the following section.

7.2.1 Attack Methodology

We notice that the data blocks do not store a direct readable representation of the data. For example, £1.00 is not stored as `0x01 00`, but `0x90 01` instead. To analyse the data structure of the Oyster card, we perform an attack that is similar to something commonly known as a chosen plaintext attack. We assume that a second level of "encryption" is performed on the data before storing. Since we have the ability to choose our travel routes using the Oyster card, we have the capability to choose the "plaintext" to be encrypted and obtain the corresponding "ciphertext". The "plaintext" in this case would be the information such as the date and time, cost of trip, service number, start and end stations. The "ciphertext" would be the new data stored on the card.

For this attack to be successful, we need to be systematically making a card dump before the start of a trip, and again after a trip. In the case of travelling on the London Underground, we make a card dump before going through the gantry at the starting station, again in the train, and once more after exiting the gantry at the finishing station. Then we compare the dumps to note the differences in the data blocks. Furthermore, depending on what we wish to find out, we can vary our trips to get the card dumps. For example, if we want to find out which bytes store the date, we can take the same trips a few times on different days. If we want to find out which bytes

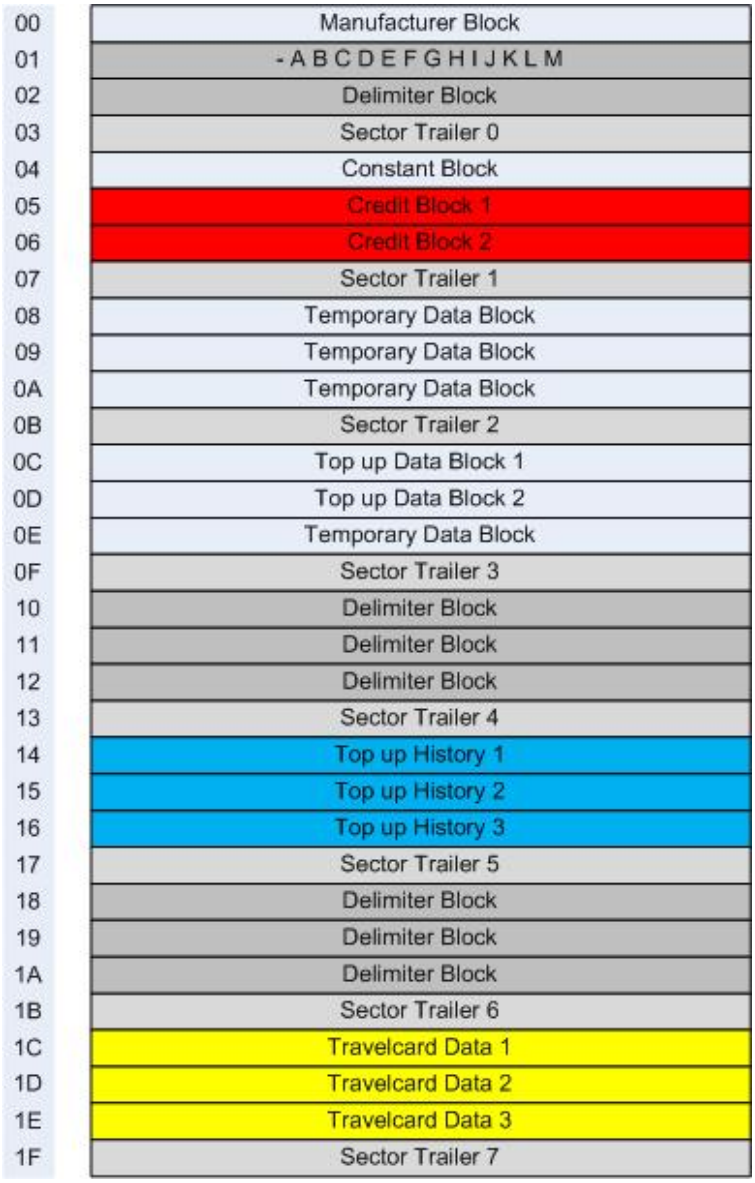

Figure 7.2: Functionality of first eight sectors

store the remaining credit, we can make repeated consecutive top ups of the same amount at the same station on the same day.

7.2.2 Our Findings

The data structure of the different types of Oyster cards are exactly the same. Other than grouping the blocks at the sector level, there are further logical subgroups that correspond to their functions. Figures 7.2 and 7.3 show the role of each data block. Important blocks are the credit blocks (shown in red), the Travelcard data blocks (shown in yellow), and the transaction history blocks (shown in blue). The significance of the colour here is that red and yellow blocks, if modified correctly, can allow one to travel for free, while the blue blocks disclose the private historical information of the card owner. We will describe some of these important blocks below.

Credit Blocks — Blocks 0x05 and 0x06 are the two most important blocks in the card, since they store the current remaining credit of the card. They are updated alternately every time the user taps the card on the genuine reader, i.e. if block 0x05 is updated during this tap, block 0x06 will be updated on the next tap. The counter that keeps track which block to be updated for the current tap is stored in byte 1. When the counter values in bytes 1 of blocks 0x05 and 0x06 are the

20	Travelcard History 1
21	Travelcard History 2
22	Travelcard History 3
23	Sector Trailer 8
24	Temporary Transaction Data 1
25	Temporary Transaction Data 2
26	Temporary Transaction Data 3
27	Sector Trailer 9
28	Temporary Transaction Data 4
29	Delimiter Block
2A	Travel History 1
2B	Sector Trailer 10
2C	Travel History 2
2D	Travel History 3
2E	Travel History 4
2F	Sector Trailer 11
30	Travel History 5
31	Travel History 6
32	Travel History 7
33	Sector Trailer 12
34	Travel History 8
35	Travel History 9
36	Travel History 10
37	Sector Trailer 13
38	Empty
39	Empty
3A	Empty
3B	Sector Trailer 14
3C	Empty
3D	Empty
3E	Empty
3F	Sector Trailer 15

Figure 7.3: Functionality of last eight sectors

same, the next tap will update block 0x06, and the counter is incremented. If the counter in block 0x05 is smaller than that in block 0x06, the next tap will update block 0x05, and the counter is also incremented. Hence we can tell from the counter value which block holds the current remaining credit.

Bytes 3, 4 and 5 collectively represent the current remaining credit of the card. The best way to confirm this is to make repeated top ups of the same amount on the same day at the same station. We list in Table 7.11 some of these mappings of hexadecimal values of bytes 3, 4 and 5 to the actual credit remaining in the card.

Hex Values of Bytes 3, 4 and 5	**Credit/£**
C8 00 00	0.00
90 01 00	1.00
54 01 00	1.70
58 02 00	2.00
E0 01 00	2.40
20 03 00	4.00
AC 03 00	4.70
68 01 00	5.80
38 FF 07	-1.00

Table 7.11: Hexadecimal values of bytes 3, 4 and 5 vs Actual credit values

Top-up History Blocks — Blocks 0x14, 0x15 and 0x16 store the top up history of the card. They are updated in a round robin fashion (after 0x16 is updated, the next update will be on 0x14). Bytes 0 to 2 store the date and time of the top up. Bytes 7 and 8 store the top up station ID. Bytes 9 and 10 store the top up amount, but like the credit blocks they do not store the amount in a readable way.

Temporary Transaction Blocks — Blocks 0x24, 0x25, 0x26 and 0x28 are used to store temporary data relating to the current travel. They are also updated in a round robin fashion (after 0x28 is updated, the next update will be on 0x24). In the case of travelling on the London Underground, when the user passes a station gantry, the date and time when the user enters and the station ID are stored in bytes 0 to 1 and 6 to 7 respectively. When exiting, the next Temporary Transaction Block is updated together with the Travel History Block (see below) such that bytes 0 to 11 are exactly the same. Bytes 8 to 11 will store the credit deducted and bytes 6 to 7 will store the start and end stations. We believe these blocks are used as a means for the system to track if the user has validated the card before taking the transportation. However in the case of taking the bus, the user only taps once upon boarding, and hence the data captured will be exactly the same as the Travel History Block.

Travel History Block — Blocks 0x2A, 0x2C, 0x2D, 0x2E, 0x30, 0x31, 0x32, 0x34, 0x35 and 0x36 (ten blocks altogether) are used to store the last ten travel histories. These are also the exact ten histories the user can retrieve using the genuine readers at Underground stations. If there are any top-up histories more recent than these ten, the top-ups will show up instead. Similarly, these blocks are updated in a round robin fashion (after 0x36 is updated, the next update will be on 0x2A). When taking the Underground, the Travel History Block is updated only when exiting the station. Moreover the data format is slightly different as compared to the entry for taking a bus. The only similarity is the date and time are stored in bytes 0 and 1. For the Underground entry, bytes 8 to 11 store the credit deducted and 6 to 7 store the start and end stations. We list in Table 7.12 some values for bytes 6 and 7 that correspond to the station pairs. For the bus entry, the service number is stored in bytes 5 to 7, and the credit deducted is stored in bytes 11 to 14.

Hex Values of Bytes 6 and 7	Start Station	Stop Station
68 20	Victoria	Brixton
69 20	Victoria	Victoria
A3 1C	Gloucester Road	South Kensington
43 0F	Hammersmith	Hammersmith
47 0F	South Ealing	Hammersmith
23 1C	Hammersmith	South Ealing
27 1C	South Kensington	South Ealing
A7 1C	South Ealing	South Kensington

Table 7.12: Hexadecimal values of bytes 6 and 7 vs Actual Start and Stop Stations

7.3 Barclaycard OnePulse

The Barclaycard OnePulse is a innovative product that combines three different functions into one single card. It is a normal Chip & Pin credit card, a contactless payment card that allows the holder to pay for small purchases less than £10, as well as an Oyster card. It is the Oyster functionality that we are interested to analyse. It works just like an ordinary Oyster card, with pay-as-you-go as well as auto top-up functions. It also allows the holder to pre-load as a Travelcard or Bus-Pass. With these functionality, it meant that the card must hold a chip that is able to communication with the existing readers via ISO-14443-A protocol. Furthermore, the card was launched after researchers had disclosed the MIFARE Classic card's vulnerabilities. We were interested to find out if Barclays had been aware of the findings and if they were, whether they had deployed the MIFARE Classic chip in their latest product.

We managed to apply for a Barclaycard OnePulse and promptly tested it using our attack. We analysed the ATQA and SAK values from the tag according to Section 3.3.3. The ATQA is `0x04 00` and the SAK is `0x28`, so we are certain it is a JCOP41 version 2.3.1 (latest version) from NXP. We knew immediately that our 3-stage card-only attack would not successful, since it is not vulnerable to the weaknesses of the MIFARE Classic chip that we had targeted. Our attack results showed that it was impossible to get the chip to repeat any tag nonce. However, the chip still replied with a encrypted transmission error message if the parity bits were correct.

In summary, we have obtained some critical findings, especially with regards to the Oyster card. We have shown that given the access conditions in those blocks in the Oyster, it is possible to edit certain bytes and travel for free. Unless Transport for London diligently carries out consistency checks on all Oyster card transactions, they stand to lose millions of pounds. However we emphasise here that we have not tampered with the values on the Oyster card in any way even though we have the technical ability and knowledge to achieve it.

Chapter 8

Evaluation of Project

We started out only knowing there were vulnerabilities with the MIFARE Classic chip from the various papers published by researchers. Like the Dutch researchers who were keen to know the insecurity of the chip that was to be used in their new tickets for public transportation, we were interested to find out more about the MIFARE Classic as we had two ideal platforms to test our work on. Firstly there is the Imperial College ID card, on which we can freely test the exploits and at the same time perform a security audit of the College's access control system. Secondly, there is the Oyster card which is used extensively for London's public transport network. There is important economical and social impact to our work. Transport for London stands to lose millions of dollars. Commuters' travel history can be recovered unknowingly. However there is still a big gap between knowing the attacks exist and actually getting all the secret keys. None of the researchers has released their attack code. Even though their explanation on the attacks are easy to comprehend, we felt that they had left out some details and that would have made it less easy to implement the attacks by others.

8.1 Evaluation of Attack Tools

We needed tools that allow us to perform traffic interception and simulation. The Proxmark allowed as to intercept any legitimate communication for analysis. While it is an extremely powerful tool, there is a huge learning curve to overcome. First we had to understand what the various hardware components are and how they work together. Then we had to learn how to modify and upload our new firmware to the board. There are frustrating times when we had uploaded our firmware that compiled correctly on our PC, but after uploading, it still corrupted the bootrom. We had no choice other than to flash the faulty bootrom with a low-level JTAG cable, which was entirely another challenge in itself. We had to purchase a JTAG cable, and learn how to use the user interface provided to flash a working bootrom onto the Proxmark 3. However, having a FPGA on the board has its obvious advantages. Signal and other processing is done on-chip, resulting in a faster response time, which is necessary during card emulation. The memory buffer also records any communication for retrieval later, as compared to sending every message to the PC via the slower USB connection.

On the other hand, libnfc is simpler to work with. The API is straightforward to use and there are examples provided to help anyone get started quickly. The reader is also a simple plug-and-play device, and there is no need to worry about drivers or firmware. However, the main disadvantage of using libnfc is the speed of communication. Every command has to be translated using the Windows USB library `libusb0.dll` and sent over the USB connection to the reader to be transmitted over the air. Similarly the received signal has to be processed by the reader, and then sent via USB to the PC to be translated. This delay makes it difficult to control the tag nonce necessary for our card-only attack.

The use of Crapto1 library functions help save us valuable time and effort. Having this software implementation of Crypto1 already available means we do not have to implement it again. However, the downsides are that the code lacks proper documentation and it cannot be used directly for the Proxmark 3 environment.

8.2 Card-Only Attack Comparison

Both Courtois and Garcia *et.al.* described card-only attacks in their recent papers [Cou09] and [GvVS09] respectively (the latter team described two similar card-only attacks). Their attacks have one thing in common — both targeted the weakness of the MIFARE Classic chip replying with an encrypted 4-bit transmission error code when the parity bits are correct even though the reader response is wrong. However there are many differences between their attacks, and we shall discuss them below.

Courtois's attack relies on a combination of a few other weaknesses. It makes use of the low entropy in the nonce generation such that it can be controlled to remain constant. It also utilises the weakness that only odd bits are used as inputs to the non-linear filter generator. This is similar to the correlation attack commonly carried out on stream ciphers where the filter generator combines output from several LFSRs to form the keystream bit. In fact, by splitting the odd and even bits due to the regularity of the filter taps, we can view the Crypto1 stream cipher as having two LFSRs. Thirdly, Courtois discovered that there is a statistical bias in the Crypto1 stream cipher where the keystream $ks1$ is independent of the last three bits of N_R. Overall, these weaknesses combine to such a great effect that the average number of queries his attack requires to gather data is only 300. However the final search computation time is not negligible, as there are approximately 2^{16} possible keys to verify. The strength of the attack lies in the low number of queries to the card. This makes reading off an unknowing victim's card in a crowded train something that is easily achievable.

In Garcis *et.al.*'s first card-only attack, the statistical bias that they found is that the keystream $ks1$ is independent of the last bit of every byte of N_R. The average number of queries this attack needs to make to find such a N_R would be 28,500. However once it has obtained this N_R, it only has 10^9 possible keys to verify.

In Garcis *et.al.*'s second card-only attack, it also makes use of the low entropy of the nonce generation such that it can be controlled to remain constant. However, it needs to first authenticate repeatedly for a N_T such that $\{N_R\}$, $\{A_R\}$, and all encrypted parity bits are 0 and the encrypted transmission error message is `0x5`. This can take, on average, 4096 authentications before generating such a combination. Furthermore they would need to generate a large table which contains all possible LFSR states that can generate this scenario. Even though generating this table is a one-time effort, it would contain 2^{36} entries (approximately 300GB in size) and searching through it to find the key would take a long time.

They also proposed an improved version of the attack to cut short the search time. This new attack keeps N_T constant, but sets $\{N_R\}$ to `0xFF FF FF FF`, $\{A_R\}$ to `0xFF FF FF FF` and try all possible combinations for the encrypted parity bits, to get the encrypted transmission error response. The effect of this optimization is to split up the large table into 4,096 smaller tables where the index is the twelve bits obtained from the last query result. However there would still be 2^{24} entries to search in each smaller table, and 128 more authentication attempts on average are required.

Our attack is a modified version of the Courtois attack. We make use of the weakness that only odd bits are used as inputs to the non-linear filter generator. We also make use of his discovery that there is a statistical bias in the Crypto1 stream cipher where the keystream $ks1$ is independent of

the last three bits of N_R. However we make our own findings about the state difference constants and use it to effectively reduce the search space from 2^{21} to 2^5. Finally, even though we make use of the weakness of the low entropy in the pseudo-random number generator, due to hardware constraints, we had to customise the attack at the expense of 2,000 queries. However our attack is achievable using inexpensive hardware that is easily obtainable.

Chapter 9

Conclusion

We have achieved successfully what we had started out for at the beginning. We wanted to exploit the weaknesses of the MIFARE Classic chip via attacks and evaluate the impact of such weaknesses in critical applications. In particular, we had accomplished the following objectives.

1. **Understand structure and evaluate weaknesses of MIFARE Classic** — Even though the Crypto1 stream cipher is a proprietary algorithm, it has not stopped researchers from uncovering its entire structure. We have also analysed in great detail the weaknesses that have been discovered by the researchers.

2. **Understand and develop attacks based on weaknesses** — We have seen proposed attacks that evolved from initially requiring access to both a genuine card and reader, to the latest ones that require only access to the card. We have also adapted our card-only attack from the original descriptions to run on inexpensive tools such as libnfc. Our greatest achievement would have been to derive the constants for the state differences and use them for our card-only attack. Our attack would not have been successful without this achievement.

3. **Apply attacks on real-world case studies** — We have successfully launched our attacks on the two case studies. As a result, we have found additional weaknesses in the implementation of the MIFARE Classic in the Imperial College's ID card. We can recover all 16 sector keys from any Oyster card that makes use of the MIFARE Classic.

4. **Uncover data structure of case studies** — We have dissected the internal data structure of the Oyster card using a systematic manner of dumping the contents before and after journeys and comparing the differences in the data blocks. This is similar to the chosen-plaintext attack that is commonly used for cryptanalysis. While we have not tampered with the Oyster card to justify our findings (such actions would amount to committing fraud), we have done numerous experiments to ensure the correctness.

From our study (as well as from other researchers' study) of the MIFARE Classic chip, it is obvious that the MIFARE Classic chip is simply broken beyond repair. MIFARE Classic cards can now be cracked and cloned in a matter of seconds discreetly. Using it in access control systems to protect valuable assets would mean that none of the assets is secure any more. In the case of the Oyster card, even if they had bind the contents of the card cryptographically to some other key, it is still possible to employ a chosen plaintext attack to understand its structure. If transport authorities do not replace the MIFARE Classic with a opened source, well tested alternative, they will continue to lose money.

9.1 Future Work

We propose two possible areas for future work. Firstly, in the Oyster card, we have located successfully the bytes that represent the current remaining credit and the travel history. Even

though we have obtained the ciphertext (byte values) for some chosen plaintext (actual credit remaining), we have not managed to uncover the actual algorithm. Hence one possible future work would be to perform cryptanalysis on this algorithm.

Secondly, Integrated Transport Smartcard Organisation (ITSO), a non profit organisation responsible for transportation ticketing specification, certification, licences, and business rules in the UK, has decided to phase out the MIFARE Classic 1K and 4K chips from its ITSO specifications [Ser09]. As a consequence, the ITSO environment will not support new MIFARE Classic chips after 31 December 2009 and will fully phase it out from the ITSO environment by 31 December 2016. They recommend alternatives that include the JCOP 41 chip, that is already being used in the Barclays OnePulse card, the Calypso chip, and the MIFARE DESFire. The analysis of these chips to find weaknesses can be a possible next course of work.

Bibliography

[AM08] Ross Anderson and Tyler Moore. Information security economics and beyond. In *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, July 2008.

[bla09] blapost. crapto1 - open implementations of attacks against the crypto1 cipher, as used in some rfid cards. `http://code.google.com/p/crapto1/`, 2009.

[Cou09] Nicolas T. Courtois. The dark side of security by obscurity. In *International Conference on Security and Cryptography*. Springer, 2009.

[dHG08] Gerhad de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A Practical Attack on the MIFARE Classic. In *Smart Card Research and Advanced Applications*, volume 5189/2008 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin / Heidelberg, 2008.

[dK08] Gerhad de Koning Gans. Analysis of the MIFARE Classic Used in the OV-Chipkaart Project. Master's thesis, Radboud University Nijmegen, 2008.

[dV08] Gerhard de Koning Gans and Roel Verdult. PROXMARK.org. `http://www.proxmark.org`, 2008.

[GdM+08] Flavio D. Garcia, Gerhad de Koning Gans, Ruben Muijrers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling MIFARE Classic. In *Computer Security - ESORICS 2008*, volume 5283/2008 of *Lecture Notes in Computer Science*, pages 97–114. Springer Berlin / Heidelberg, 2008.

[GvVS09] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a mifare classic card. In *30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 3–15. IEEE, 2009.

[Jou09] RFID Journal. The history of rfid technology - rfid journal. `http://www.rfidjournal.com/article/view/1338/1`, 2009.

[NESP08] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse-engineering a cryptographic rfid tag. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 185–194. USENIX Association, July 2008.

[NP07] Karsten Nohl and Henryk Plötz. Mifare: Little Security, Despite Obscurity. In *24th Chaos Communication Congress*, December 2007.

[NXP08a] NXP Semiconductors. *MIFARE Standard 1 KByte Card IC Functional Specification*, 5.3 edition, January 2008.

[NXP08b] NXP Semiconductors. *MIFARE Standard 4 KByte Card IC Functional Specification*, 4.1 edition, January 2008.

[oL07] Mayor of London. Press release - mayor to give away 100,000 free oyster cards. `http://www.london.gov.uk/view_press_release.jsp?releaseid=11611`, 2007.

[pro09] proxcat. Get a proxmark3. `http://www.proxmark3.com/`, 2009.

[Ser09] ITSO Services. Mifare classic phase out — itso services. http://www.itsoservices.org.uk/page252/Mifare-Classic-Phase-Out, 2009.

[Sta05] William Stallings. *Cryptography and Network Security Principles and Practices*. Prentice Hall, fourth edition, November 2005.

[Ver08] Roel Verdult. libnfc.org - Public platform independent Near Field Communication (NFC) library. http://www.libnfc.org, 2008.

[Ver09a] Roel Verdult. Classic mistakes. In *Hacking At Random*, 2009.

[Ver09b] Roel Verdult. Proxmark developers community. http://www.proxmark.org/forum/, 2009.

[Wes09] Jonathan Westhues. A Test Instrument for HF/LF RFID. http://cq.cx/proxmark3.pl, 2009.