Mifare Classic – Eine Analyse der Implementierung

Henryk Plötz

Betreuer: Prof. Dr. Ernst-G. Giessmann

4. Analyse des Funk-Protokolls 41
   4.1. Erste Sniffing-Ergebnisse 41
   4.2. Wiederholte Challenge-Response-Läufe 43
   4.3. Protokollmanipulationen 44

5. Physikalisches Reverse Engineering 47
   5.1. Chip öffnen und photographieren 47
   5.2. Fotos auswerten 48

6. Analyse-Ergebnisse 51
   6.1. Pseudozufallszahlengenerator 51
   6.2. Crypto-1-Chiffre 52

III. Angriffe 55

7. Angriffe auf das Funk-Protokoll 57
   7.1. Online Brute Force – Rohe Gewalt 57
   7.2. Relay-Attacken 58
      7.2.1. Gegenmaßnahmen 59
   7.3. Man-in-the-Middle-Attacken 60
   7.4. Replay-Attacken 62
   7.5. Keystream-Recovery 63
      7.5.1. Gegenmaßnahmen 64

8. Angriffe auf die Crypto-1-Chiffre 65
   8.1. Schwächen in der Initialisierung 65
   8.2. Schwächen in der Schlüsselstromgenerierung 66
   8.3. Algebraischer Angriff 66

IV. Abschluss 67

9. Zusammenfassung und Ausblick 69

A. Crypto-1-Beispielimplementierung 71

Abbildungsverzeichnis 89
Tabellenverzeichnis 91
Verzeichnis der Listings 93
Verzeichnis der Algorithmen 95
Literaturverzeichnis 97
Vorwort

Diese Arbeit fasst die Ergebnisse und Zwischenschritte einer Analyse der Funktion und Sicherheitseigenschaften eines weit verbreiteten Typs von kontaktlosen Speicherkarten (Mifare Classic) zusammen.


Dennoch sind noch viele angreifbare Systeme in aktiver Benutzung. Aus diesem Grund wird die Implementation hier nicht bis ins letzte Detail betrachtet werden, sondern nur ausreichend detailliert, um die Schwächen analysieren und bewerten zu können.

Dank an Jan Krissler und Karsten Nohl, die die handwerklich-praktischen bzw. kryptologisch-theoretischen Teile der Analyse durchgeführt haben, während ich mich mit den Protokollfragen und der Programmierung eines Emulators beschäftigen konnte.

Dank auch an Harald Welte, Milosch Meriac und Brita Meriac, ohne deren grundlegende Hard- und Software diese Arbeit so nicht möglich gewesen wäre.
Abkürzungsverzeichnis

APDU ............ Application protocol data unit, Dateneinheit auf Applikationsprotokollebene
ASK ............. Amplitude Shift Keying, Amplitudenmodulation
ATQA .......... Answer To Request, Type A, Anfrageantwort Typ A
ATS ............ Answer To Select, Selektionsantwort
BCC ........... Bit Count Check
BPSK .......... Binary Phase Shift Keying, Binäre Phasenumtastung
CRC ............ Cyclic Redundancy Check
CSS ........... Content Scrambling System
CSV ........... Comma Separated Values, Kommaseparierte Werte
CT ............. Cascade Tag
DST ........... Digital Signature Transponder
EEPROM ....... Electrically erasable programmable read-only memory, Elektrisch löscharbarer, programmierbarer Nur-Lese-Speicher
EOF ........... End of Frame, Ende des Datenrahmens
FDT ........... Frame Delay Time, Mindestabstand zwischen zwei Datenrahmen
GSM ........... Global System for Mobile communication
HTLA ........... Halt, Type A, Abschalten, Typ A
IC .............. Integrated Circuit, Integrierter Schaltkreis
IRQ ........... Interrupt Request, Unterbrechungsanforderung
LFSR ........... Linear Feedback Shift Register, Linear rückgekoppeltes Schieberegister
MAC ........... Message Authentication Code
MSB ........... Most Significant Bit, Höchstwertiges Bit
PCD ........... Proximity Coupling Device
PICC .......... Proximity Integrated Circuit Card
PLL ........... Phase Locked Loop, Phasenregelschleife
PPS ........... Protocol and Parameter Selection, Protokoll- und Parameterauswahl
PRNG ........... Pseudo Random Number Generator, Pseudozufallszahlengenerator
PWM ........... Pulse Width Modulation, Pulsweitenmodulation
RATS .......... Request for Answer to Select, Anfrage nach Selektionsantwort
REQA .......... Request, Type A, Anfrage, Typ A
SAK ........... Select Acknowledge, Typ A, Auswahlbestätigung, Typ A
SOF ........... Start of Frame, Beginn des Datenrahmens
SSC ........... Synchronous Serial Controller, serieller synchroner Controller
Vorwort

UART ........ Universal Asynchronous Receiver Transmitter, Universeller asynchroner Sender/Empfänger
UID ........ Unique Identifier, eindeutige Identifikationsnummer
VCD .......... Vicinity Coupling Device
VICC .......... Vicinity Card
WUPA .......... Wake-up, Type A, Aufwecken, Typ A
Teil I.

Einführung

Einführung und Erklärung zu Grundlagen der RFID-Technologie
Einleitung

RFID-Systeme haben in den letzten 10 Jahren eine stark zunehmende Verbreitung erfahren. Sicherheits- und Datenschutzbedenken sind dabei oft auf der Strecke geblieben, gerade bei preiswerten Systemen.

Zu oft wird auch der Grundsatz ’Security by Obscurity’, also ’Sicherheit durch Verwahrung’ verfolgt, indem wichtige Teile der Sicherheitssysteme geheim gehalten werden. Es hat sich in der Vergangenheit immer wieder gezeigt, dass dieses Vorgehen nicht auf lange Sicht erfolgreich sein kann.

Mifare Classic war ein herausragendes Ziel um die Schwächen des Security-by-Obscurity-Ansatzes zu zeigen, da das System relativ alt ist (eingeführt vor mehr als 14 Jahren) aber immer noch sehr verbreitet eingesetzt wird.

Überblick


Verwandte Arbeiten

Neben der OpenPICC-Emulatorhardware die in Abschnitt 3.2 beschrieben werden wird, gibt es noch mindestens drei andere Projekte, die ähnliche Fähigkeiten aufweisen:

- Der Proxmark III von Jonathan Westhues ([Wes])
- Der RFID-Guardian der Forschungsgruppe um Melanie Rieback ([RGC+06])
- Der Ghost von Roel Verdult ([Vero8])
1. Einleitung

Wer mit Ungeheuern kämpft, mag zusehn, dass er nicht dabei zum Ungeheuer wird. Und wenn du lange in einen Abgrund blickst, blickt der Abgrund auch in dich hinein.

Jenseits von Gut und Böse,
FRIEDRICH NIETZSCHE

Kapitel 2

RFID

Radio Frequency IDentification (Funkfrequenz-Identifikation, RFID) ist ein Oberbegriff für Identifikationssysteme über Funkkanäle. Wenn man den Begriff sehr weit definiert, fallen darunter bespielsweise auch Freund-Feind-Identifikationssysteme aus Kampfflugzeugen oder die Transponder in den meisten Verkehrsflugzeugen. Die häufiger gebrauchte, engere Definition beschränkt sich auf Systeme mit einem schwachen, kleinen Tag (oder Transponder) und einem davon deutlich unterschiedenen, meist größeren Lesegerät.


2.1. Grundsätzliche Funktionsweise

RFID-Systeme unterscheiden sich in aktive Systeme, bei denen das Tag oder der Transponder über eine eigene Energiequelle verfügt, und passive Systeme, bei denen das Tag seine Energie vom Lesegerät bezieht. Aktive Systeme haben in der Regel größere Reichweiten als passive Systeme, dafür aber auch meist größere, klobigere Bauformen und entsprechend eingeschränkte Einsatzgebiete und natürlich eine Lebenserwartung, die die der verwendeten Energiequelle nicht überschreiten kann.

Passive RFID-Systeme

Bei den passiven RFID-Systemen werden zwei Verfahren verwendet: Backscatter-Systeme im elektromagnetischen Fernfeld und UHF-Frequenzbereich sowie Lastmodulations-Systeme im elektromagnetischen Nahfeld.
2. RFID

2.2. Frequenzbereiche

Bei den verbreiteten RFID-Kartensystemen werden hauptsächlich zwei Frequenzbereiche eingesetzt: Niederfrequenz bei ca. 120 kHz–135 kHz und Hochfrequenz bei 13,56 MHz.

2.2.1. 13.56MHz

ISO 14443

ISO 14443 ([ISOa]) definiert Proximity Coupling, also Nahbereichs-Ankopplung mit einer ungefähren Maximalreichweite von 10 cm, für Karten, abgekürzt PICC (Proximity Integrated Circuit Card) und Lesegeräte, abgekürzt PCD (Proximity Coupling Device).


Im verabschiedeten Standard werden zwei verschiedene Typen, A und B, definiert, die sich in Modulation und Kodierung sowie Antikollisionsverfahren und Aktivierung unterscheiden. In einer Kommunikationssitzung mit einer Karte kann nur jeweils einer dieser beiden Typen verwendet werden. Sowohl PCD als auch PICC können aber beide Typen implementieren (und ggfl. zusätzliche, andere Interfacetypen). Das PCD wechselt dann, wenn es keine aktive Verbindung hat, zwischen den Typen hin und her und versucht die jeweils definierte Initialisierungssequenz auszuführen.

Die Basisbitrate ist in beiden Fällen ungefähr 105,9 kbit/s, was sich aus ~9,44 µs per Bit ergibt, siehe Abschnitt 2.3.2 auf Seite 8.

**Typ A**

ISO 14443 Typ A verwendet in der PCD→PICC-Richtung 100% ASK (Amplitude Shift Keying, Amplitudenmodulation), d.h. die Stärke des Trägerfelds wird zwischen voller Stärke und 100% weniger, also kein Trägerfeld, binär umgeschaltet. Da dabei im modulierten Zustand keine Energieübertragung zur Karte stattfindet, muss die Zeit, in der das Feld moduliert wird, so kurz wie möglich gehalten werden. Zu diesem Zweck kommt eine modifizierte Miller-Kodierung zum Einsatz. Die modifizierte Miller-Kodierung schaltet das Feld jeweils nur für sehr kurze Zeit ab – 2 µs bis 3 µs, entsprechend ungefähr einer viertel Bitlänge – und verhindert direkt aufeinanderfolgende Feldabschaltungen weitestgehend.

In der Rückrichtung wird Lastmodulation mit einem 847 kHz-Subträger und Manchester-Kodierung verwendet. Um sicherer zwischen der gewünschten Lastmodulation und anderen Effekten (Entfernung zwischen Lesegerät und Karte, Anzahl der Karten, etc.) unterscheiden zu können, kommt ein Subträger zum Einsatz. Das bedeutet, dass die Last nicht einfach nur direkt vom Modulationssignal gesteuert wird, sondern das Modulationssignal vorher...
mit einem 847 kHz-Signal gemischt wird. Wenn dieses Subträger-Lastmodulationssignal auf das Trägersignal gegeben wird, entstehen zwei Seitenbänder: eins bei 13,56 MHz+847 kHz und eins bei 13,56 MHz-847 kHz. Das Lesegerät kann einen Bandpassfilter verwenden, um nur gezielt eines der beiden Seitenbänder zu empfangen (und die Trägerfrequenz im Empfänger zu unterdrücken).

**Typ B** ISO 14443 Typ B verwendet in PCD→PICC-Richtung 10% Amplitudenmodulation – das Feld wechselt zwischen voller Stärke und 90% Stärke – und NRZ-L-Kodierung, d.h. die zu sendenden Daten werden direkt auf den Träger aufmoduliert. Dabei entspricht ein unmoduliertes (starkes) Feld einer logischen 1 und ein moduliertes (abgeschwächt) Feld einer logischen 0.

In PICC→PCD-Richtung verwendet Typ B Lastmodulation mit einem 847 kHz-Subträger und BPSK\(^1\)-modulierter NRZ-L-Kodierung. Die Datenbits werden also auf einem dauerhaft aktiven Subträger übertragen, indem die Phase des Subträgers moduliert wird. Zur Synchronisation wird am Anfang jeder Übertragung vom PICC ein Referenzsubträger gesendet, dessen Phase die initiale Phasenlage darstellt. Eine Bitperiode mit dieser Phase entspricht dann einer logischen 1, während eine Bitperiode mit einer um 180° gedrehten Phase einer logischen 0 entspricht.

Durch die geringere Modulationstiefe übermittelt Typ B mehr Energie an die Karte als Typ A und wird daher vor allem bei kontaktlosen Smartcards eingesetzt, um dem vergleichsweise hohen Energieverbrauch der Kryptographielogik entsprechen zu können.

**ISO 15693**
ISO 15693 ([ISOb]) spezifiziert Vicinity-Coupling, also Nachbarschafts-Ankopplung, mit einer ungefähren Maximalreichweite von 1 m. Die Karten heißen hierbei VICC (Vicinity Card) und die Lesegeräte VCD (Vicinity Coupling Device). Durch die größere Entfernung wird weniger Energie zur Karte übertragen, weswegen diese in der Regel keine komplexe Kryptographielogik enthalten kann und mit diesem Standard meist nur einfache Speichertransponder (insbesondere zum Einsatz in der Logistik) implementiert werden.

Die zur Verfügung stehende Übertragungsgeschwindigkeit ist je nach eingesetzter Modulation deutlich geringer als bei ISO 14443, in jedem Fall aber kleiner oder gleich 26,69 kbit/s.

### 2.3. ISO 14443 Typ A im Detail


#### 2.3.1. Part 1

ISO 14443 Part 1 definiert die physikalischen Eigenschaften von PICCs im ID-1-Format (85,60 mm x 53,98 mm, per Referenz auf ISO 7810 [ISOc]). Das betrifft sowohl die Größe der

\(^1\)BPSK (Binary Phase Shift Keying, Binäre Phasenumtastung)
Karten, als auch Widerstandsfähigkeit gegen UV-Strahlung und Röntgenstrahlung, Zug- und Knickfestigkeit, Widerstandsfähigkeit gegen alternierende oder statische elektrische oder magnetische Felder sowie die Betriebstemperatur.

### 2.3.2. Part 2

Part 2 beschreibt das Funkinterface zur Energie- und Datenübertragung.

Die Bitdauer $t_{\text{Bit}}$, wie die meisten Basiseinheiten des Funkprotokolls (zum Beispiel auch die Subträgerfrequenz $f_s$), ist über einen festen Teiler in Abhängigkeit von der Trägerfrequenz $f_c$ spezifiziert:

\[
    f_c = 13,56 \text{ MHz} \pm 7 \text{ kHz} \\
    t_{\text{Bit}} = \frac{128}{f_c} \approx 9,44 \mu\text{s} \\
    f_s = \frac{f_c}{16} \approx 847,5 \text{ kHz}
\]


### 2.3.3. Part 3

Part 3 schließlich ist der umfangreichste der vier Teile. Dieser Teil beschreibt das Rahmenformat für die Kommunikation sowie eine Zustandsmaschine und einen Kommandosatz für die Antikollision und Selektion.

Die FDT (*Frame Delay Time*, Mindestabstand zwischen zwei Datenrahmen) wird als Formel ausgehend vom Ende der letzten Modulation des Feldes durch das Lesegerät bis zum Beginn der ersten Modulation durch die Karte beschrieben: $FDT = (n \ast 128 + 84) / f_c$, 

### Abbildung 2.1. Visualisierung der Modulation und Kodierung in PCD→PICC- und PICC→PCD-Richtung, nach ISO 14443-2 Abbildung 1
Tabelle 2.1. Kodierungen für beide Kommunikationsrichtungen, jeweils bezogen auf eine Bitlänge

<table>
<thead>
<tr>
<th>Symbol</th>
<th>PCD→PICC</th>
<th>PICC→PCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOF</td>
<td>Mod. am Anfang</td>
<td>Mod. während der ersten Hälfte</td>
</tr>
<tr>
<td>0</td>
<td>nach 1-Bit: keine Modulation</td>
<td>Mod. während der zweiten Hälfte</td>
</tr>
<tr>
<td></td>
<td>sonst: Mod. am Anfang</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Modulation nach der Hälfte</td>
<td>Mod. während der ersten Hälfte</td>
</tr>
<tr>
<td>EOF</td>
<td>ein 0-Bit, dann eine Bitdauer lang</td>
<td>keine Modulation</td>
</tr>
<tr>
<td></td>
<td>keine Mod.</td>
<td></td>
</tr>
</tbody>
</table>

wenn das letzte Datenbit des PCD-Frames ein 1-Bit ist, und \( FDT = \frac{(n \times 128 + 20)}{f_c} \), wenn das letzte Datenbit ein 0-Bit ist. Dieses Verhältnis ist in Abbildung 2.2 auf der nächsten Seite dargestellt. Wie dort zu erkennen ist, führt diese Definition dazu, dass das PICC-Frame in jedem Fall \( (n \times 128) \) Trägertaktzyklen nach der Mitte der ersten Bitperiode des EOF-Signals des PCD beginnt.

Die Variable \( n \) ist für alle Kommandos, die zur Antikollisionsprozedur gehören (REQA, WUPA, ANTICOLLISION, SELECT), fest auf 9 gesetzt. Das stellt sicher, dass alle Karten im Leserfeld während der Antikollision gleichzeitig antworten. Für alle anderen Kommandos gilt \( n \geq 9, n \in \mathbb{N} \).

Es werden drei verschiedene Frametypen definiert, die sich in Länge, Paritätsprüfung und CRC unterscheiden:

**Short Frame** wird für die Initialisierung benutzt. Es besteht aus SOF, 7 Bit Daten und EOF. Das MSB (Most Significant Bit, Höchstwertiges Bit) wird zuerst übertragen. Es werden keine Paritätsbits oder CRC hinzugefügt.

**Standard Frame** wird für die normale Kommunikation benutzt. Es besteht aus SOF, den Daten (8 bit, MSB zuerst, dann ein zusätzliches Paritätsbit, so dass die Parität der 8+1 Bits ungerade ist) und EOF.

**Bitorientiertes Antikollisionsframe** wird nur in der Antikollisionsphase benutzt. Es besteht aus zwei Teilen: Ein Teil, der vom PCD an die PICC gesendet wird und einem Teil, der von dem (oder den) PICC an das PCD gesendet wird. Der erste Teil besteht aus SOF und \( x \) Datenbits, der zweite Teil besteht aus \( 56 - x \) Datenbits und EOF. Es gilt \( 16 \leq x \leq 55 \).

Die CRC von Typ A wird als \( \text{CRC}_A \) bezeichnet und wird über Referenz auf ISO/IEC 13239 definiert, wobei der Initialzustand \( 63 \ 63h \) betragen soll und keine Invertierung des Registerinhalts am Ende erfolgt. Listing 2.1 auf Seite 12 zeigt eine Beispielimplementierung.
Abbildung 2.2. FDT zwischen Ende der PCD-Kommunikation und Beginn der PICC-Kommunikation, nach ISO 14443-3 Abbildung 1

in C. Die CRC_A wird für alle Standardframes aus ISO 14443 Part 3 verwendet, bis auf ATQA-Frames².

Abbildung 2.3 auf der nächsten Seite zeigt die Zustandsmaschine, die ISO 14443-3 für Typ-A-Karten definiert. Die Zustandsübergänge sind dabei:

REQA, WUPA REQA (*Request, Type A, Anfrage, Typ A*) bzw. WUPA (*Wake-up, Type A, Aufwecken, Typ A*) bei der PICC empfangen (und beantwortet)

AC, SELECT Antikollisionskommando bzw Selektionskommando mit zutreffender UID bei der PICC empfangen (und beantwortet)

HLTA, DESELECT HTLA (*Halt, Type A, Abschalten, Typ A*) bzw. Deselektionskommando bei der PICC empfangen

* Sonstige Daten bei der PICC empfangen, insbesondere bei fehlerhaftem Empfang (Paritätsprüfung bzw. CRC stimmt nicht) oder Kommandos in falscher Reihenfolge (zum Beispiel REQA nach AC)


²Während der Mifare-Classic-Authentisierungsphase gibt es ebenfalls Standardframes ohne CRC.
Abbildung 2.3. Zustandsmaschine für Tags nach ISO 14443-3 Typ A, vereinfacht nach ISO 14443-3 Abbildung 6

Abgeschaltet

IDLE

ACTIVE

HALT

READY*

SELECT

ACTIVE*

ISO/IEC 14443-4

Proprietäre Kommunikation

HLTA

ISO 14443 Part 4

DESELECT

HLTA
### Listing 2.1 Implementierung von CRC_A

```c
#include <stdio.h>
#include <sys/types.h>

#define POLYNOM 0x8408
#define INITIAL 0x6363

/* Das folgende Makro gibt Bit Nr. bitpos aus dem Bitstring ab Adresse buffer aus. */
#define SELECT_BIT(buffer, bitpos) (!!(buffer[bitpos/8] & (1<<(pos%8))) )

u_int16_t calc_crc(u_int8_t *daten, size_t len)
{
    u_int16_t reg = INITIAL;
    size_t pos;
    for(pos=0; pos < 8*len; pos++) {
        char bit = (reg ^ SELECT_BIT(daten, pos)) & 1;
        reg >>= 1;
        if(bit) reg ^= POLYNOM;
    }
    return reg;
}

int main(void)
{
    u_int8_t testdaten[] = {0x12, 0x34};
    printf("CRC_A(testdaten) = %04x\n", calc_crc(testdaten, sizeof(testdaten)));
    return 0;
}
```

### Abbildung 2.4. ANTICOLLISION- und SELECT-Kommandos

<table>
<thead>
<tr>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>ANTICOLLISION</td>
</tr>
<tr>
<td>SELECT</td>
</tr>
</tbody>
</table>

Das Kommando REQA ist 26h (wird auf dem Funkkanal also in der Reihenfolge SOF 0 1 1 0 0 1 0 EOF gesendet) und dient dazu, Karten aus dem Ruhezustand IDLE in den Bereitschaftszustand READY zu versetzen. WUPA (52h) verhält sich identisch, weckt aber zusätzlich auch Karten auf, die im Haltezustand HALT sind.

Nach Empfang von REQA senden alle Karten synchron (FDT mit \( n = 9 \)) ihre ATQA (Answer To Request, Type A, Anfrageantwort Typ A), welche Informationen über die UID-Länge (einfach, doppelt oder dreifach) sowie unterstützte Antikollisionsverfahren enthält. ISO 14443-3 definiert dabei nur bitweise Antikollision.

Die Kommandos SELECT und ANTICOLLISION (dargestellt in Abbildung 2.4) sind verwandt und voneinander abgeleitet: Der Selektionscode SEL ist entweder 93h, 95h oder 97h (das entspricht den Kaskadierungsebenen 1, 2 oder 3). Das Feld NVB gibt die Anzahl der gültigen Bytes und Bits im Kommando an, als Wert \( xyh \), wobei \( x \) die Anzahl der vollen
2.3. ISO 14443 Typ A im Detail

Bytes ist und \( y \) die Anzahl der Bits im unvollständigen Byte. So steht also \( NVB = 70h \) für 7 gültige Bytes plus 0 Bits oder \( NVB = 34h \) für 3 vollständige Bytes plus 4 Bits. Die Anzahl an Bytes zählt die Bytes SEL und NVB mit, ist also immer mindestens 2 und maximal 7. Wenn 7 vollständige Bytes übertragen werden, heißt das Kommando SELECT und es folgt eine CRC, sonst heißt es ANTICOLLISION, ohne CRC.


Ablauf der bitorientierten Antikollision

Die bitorientierte Antikollision führt eine binäre Baumsuche über die UID (Unique Identifier, eindeutige Identifikationsnummer) von allen im Feld befindlichen Karten durch, bis mindestens eine UID vom Lesegerät vollständig und fehlerfrei empfangen wurde. Das Lesegerät sendet dann ein Selektionskommando für diese UID, woraufhin alle anderen Karten in den IDLE- bzw. HALT-Zustand zurückfallen. Dieses Vorgehen garantiert, dass zu jedem Zeitpunkt nur höchstens eine Karte im Leserfeld im ACTIVE- bzw. ACTIVE*-Zustand ist und es im Verlauf der normalen Kommunikation nicht zu Kollisionen kommen kann, die durch den Versuch, mehrere Karten gleichzeitig zu senden, verursacht werden und die einwandfreie Datenübertragung stören würden.

Für den Zweck des Antikollisionsverfahrens hat jede Karte eine UID der Länge 4, 7 oder 10 Byte. Diese UID kann fest eingeprogrammiert sein (üblich bei Mifare Classic und Ultralight) oder durch einen Zufallsgenerator bei jedem Übergang vom abgeschalteten in den IDLE-Zustand neu generiert werden (üblich zum Beispiel bei elektronischen Reisepässen). In letzterem Fall muss das erste Byte der UID 08h sein.

Um nicht übermäßig häufig die gleichen UID-Bytes übertragen zu müssen (im Fall der 7 und 10 Byte langen UIDs), wird der Vorgang in 3 Kaskadierungsebenen unterteilt. In jeder Kaskadierungsebene werden nur maximal 32 Bit der UID (plus 8 Bit BCC) behandelt, wobei bei der ersten bzw. zweiten Ebene einer UID mit doppelter bzw. dreifacher Länge das erste Byte auf 88h gesetzt wird (das sogenannte CT (Cascade Tag), es darf nicht als erstes Byte einer UID verwendet werden).

Der Antikollisionsalgorithmus beginnt (nachdem das Lesegerät durch REQA oder WUPA festgestellt hat, dass sich Karten im Feld befinden) in der ersten Kaskadierungsebene damit, dass das Lesegerät ein ANTICOLLISION-Kommando mit \( SEL=93h \) und \( NVB=20h \) (also ohne UID-Bits, nur 2 Bytes gesamt) sendet. Nach einer FDT mit \( n = 9 \) antworten alle Karten im Feld, die sich im READY- oder READY*-Zustand befinden, mit ihrer vollen UID und BCC\(^3\). Das Lesegerät unterscheidet jetzt, ob sich eine oder mehrere Karten im Feld befinden durch Auswertung, ob die Manchester-Kodierung in der Antwort korrekt eingehalten wurde. Bei korrekter Manchester-Kodierung wird immer nur genau während

\(^3\)Die BCC (Bit Count Check) ist eine Prüfsumme, die durch byteweises XOR aller 4 vorhergehenden Bytes generiert wird.
2. RFID

einer Bithälfte moduliert. Wenn das Lesegerät im empfangenen Signal eine Bitdauer findet, in der während der gesamten Zeit Modulation vorhanden ist, so ist das eine Kollision. An dieser Stelle unterscheiden sich die UIDs von mindestens zwei im Feld befindlichen Karten.

Wenn keine Kollision stattgefunden hat, dann hat das Lesegerät erfolgreich eine UID empfangen und sendet als nächstes ein SELECT-Kommando mit dem aktuellen SEL-Code, NVB=70h, der empfangenen UID und BCC und anschließender CRC. Die dadurch angesprochene Karte bestätigt mit einem SAK (Select Acknowledge, Typ A, Auswahlbestätigung, Typ A), welches ein Standardframe mit einem Byte Nutzdaten plus 2 Byte CRC ist. Das SAK-Frame gibt an, ob die UID vollständig ist (also keine weiteren Kaskadierungsabläufe notwendig sind) oder nicht (weitere Kaskadierung notwendig) sowie ob die Karte konform zu ISO 14443-4 ist. Wenn weitere Kaskadierung notwendig ist, wird der aktuelle SEL-Code erhöht und die Antikollisionsprozedur wird auf dieser Ebene mit neun bekannten UID-Bytes neu begonnen.

Falls als Antwort auf das ANTICOLLISION Kommando mehrere Karten ihre UIDs gesendet haben, sucht das Lesegerät die Stelle der ersten Kollision und sendet ein neues ANTICOLLISION Kommando mit dem aktuellen SEL-Code und allen UID-Bits bis direkt vor die Stelle der Kollision, gefolgt von einem 0- oder einem 1-Bit (implementierungsspezifisch). Darauf antwortet dann mindestens eine Karte weniger als zuvor und der Vorgang wird solange wiederholt wie es Kollisionen gibt.

Algorithmus 2.1 zeigt eine Variante des Antikollisionsalgorithmus vollständig, welche nur die erste gefundene UID selektiert. Der Einfachheit halber wird dort statt NVB mit einem simplen Integer für die UID-Länge gearbeitet sowie die Übertragung von den Paritätsbits abstrahiert betrachtet.

2.3.4. Part 4


2.4. Mifare

Mifare (Mikron FARE System, also Mikron Fahrgeld-System) wurde Anfang bis Mitte der 1990er Jahre von der „Mikron Gesellschaft für Integrierte Mikroelektronik“ in Gratkorn, Österreich entwickelt. Die ersten Mifare-ICs wurden 1994 ([NXP]) eingeführt und haben sich durch zwei Besonderheiten hervor: eine hohe Datenübertragungsrate von \( \approx 106 \text{kbit/s} \)

---

4Ein Algorithmus welcher alle UIDs auflistet ohne eine bestimmte Karte zu selektieren ist auch möglich, aber aufwendiger.

5APDU (Application protocol data unit, Dateneinheit auf Applikationsprotokollebene)
Algorithmus 2.1 Bitorientiertes Antikollisionsverfahren nach ISO 14443-3 Typ A

function DoOneCascadeLevel(SEL)
    \( n \leftarrow 0 \) // Die Anzahl der gültigen UID-Bits (als Integer)
    UID \leftarrow \emptyset // UID als Bitstring
    repeat
        Sende ANTICOLLISION(SEL, n, UID)
        \( r \leftarrow \) empfangene UID-Bits
        if Kollision detektiert then
            \( c \leftarrow \) Bitposition der ersten Kollision
            UID \leftarrow UID \parallel r_0 \parallel \ldots \parallel r_{c-1} \parallel 0 \) // Alternativ \( \parallel 1 \)
        end if
        \( n \leftarrow \) LÄNGE(UID)
    until \( n = 40 \)
    Sende SELECT(SEL, n, UID)
    SAK \leftarrow \) Empfangenes SAK
    return UID, SAK
end function

function SelectFirst
    SEL \leftarrow 93h
    UID \leftarrow \emptyset
    repeat
        \( u, SAK \leftarrow \) DoOneCascadeLevel(SEL)
        if SAK zeigt an, dass die UID nicht vollständig ist then
            UID \leftarrow UID \parallel u_8 \ldots u_{39} \) // \( u_0 \ldots u_7 = 88h \)
            SEL \leftarrow SEL + 2
        else
            UID \leftarrow UID \parallel u_0 \ldots u_{39}
        end if
    until SAK zeigt an, dass die UID vollständig ist
    return UID, SAK // Volle UID sowie letztes SAK
end function

(während konkurrierende System üblicherweise nur ca. 10 kbit/s erreichten) und sichere, verschlüsselte Datenübertragung mit einer 48-bit Stromchiffre. Die verwendete Chiffre wurde Crypto-1 genannt und als Geschäftsgeheimnis nie publiziert und es fand auch keine unabhängige, externe Begutachtung statt.

2.4.1. Geschichte


Das von den Mifare-Chips verwendete, proprietäre Funk-Kommunikationsprotokoll fand ca. 1998 Einzug in ISO 14443 als Kommunikationsinterface Typ A.

2.4.2. Varianten


Mifare Classic wird als Oberbegriff für die Teilfamilie verwendet, die nur das alte Crypto-1-Verfahren als Verschlüsselungsalgorithmus unterstützen und daher mehr oder weniger direkt von den ursprünglichen Mifare-Chips abgeleitet sind. Die verschiedenen Varianten dieses Chips haben wegen ihres vergleichsweise guten Preis/Leistungsverhältnis weltweit den größten Markanteil bei kontaktlosen Speicherkarten mit Sicherheitsfunktionen. Verschiedene Schätzungen beziffern die Anzahl der verkauften Mifare-Classic-Karten auf ca. 1 Milliarde bis 2 Milliarden.

Mifare Classic 1k ist der Ur-Mifare-Chip, enthält 1 Kilobyte EEPROM, aufgeteilt in 16 Sektoren zu je 4 Blöcken à 16 Byte (64 Blöcke insgesamt). Der letzte Block jedes Sektors enthält die Zugriffsschlüssel und Zugriffsbedingungen und ist nicht generell für Benutzerdaten zu verwenden. Der erste Block (Block 0) enthält außerdem die UID der Karte sowie einige herstellerspezifische Daten und ist permanent schreibgeschützt (sog. Manufacturer Block), siehe Abbildung 2.5 auf Seite 20. Der reine, für Nutzdaten verfügbare Speicherplatz beträgt also 752 Byte.

Mifare Classic 4k ist eine erweiterte Version des Classic 1k mit 4 Kilobyte Speicher, aufgeteilt in 32 Sektoren zu je 4 Blöcken plus 8 Sektoren zu je 16 Blöcken. Der reine Nutzdatenbereich beträgt demnach 3440 Byte.

Mifare Mini ist eine reduzierte Version für Anwendungen mit reduziertem Speicherbedarf mit 320 Byte in 5 Sektoren zu 4 Blöcken. Der reine Nutzdatenbereich ist damit auf 224 Byte beschränkt.

Mifare Light war ebenfalls eine reduzierte Version mit nur 384 Bit aufgeteilt in 12 Seiten mit je 4 Byte, wovon 8 Byte für UID und Verwaltungsdaten verbraucht wurden (nur lesbar)

---

6Bis auf das Funkinterface natürlich.

7Dieser Kartentyp ist vollständig verschwunden. Es gibt auf den NXP- und Philips-Webseiten keinen Hinweis mehr auf seine Existenz.
sowie weitere zwei mal 8 Byte für die Schlüssel A und B (nicht lesbar) und Zugriffskontrollinformationen.

**Mifare Ultralight** ist die kostengünstigste Variante der gesamten Familie und enthält keine Verschlüsselungsfunktionen und nur 512 Bit (64 Byte) Speicher. Hiervon sind 10 Byte werksseitig schreibgeschützt, da sie die UID (sowie ein herstellerspezifisches Byte) enthalten, 6 Byte, die nur einmal beschreibbar sind (inklusive 2 Byte die die Lock-Bits enthalten) und 48 Bytes frei für Nutzerdaten verwendbar. Diese Variante ist hauptsächlich für Verwendung in Einmal- oder Wegwerftickets vorgesehen.


**Mifare DESfire EV1** (vormals Mifare DESfire8) ist die Weiterentwicklung der ursprünglichen DESfire-Karten mit Unterstützung für 128 Bit AES-Verschlüsselung und unterschiedlichen Speichergrößen von 2, 4 oder 8 Kilobyte.


2.4.3. Mifare-Classic-Zugriffsbedingungen

Mifare Classic erlaubt bis zu zwei verschiedene Schlüssel pro Sektor (Schlüssel A und Schlüssel B), denen blockweise unterschiedliche Zugriffsrechte eingeräumt werden können. Diese Schlüssel und Zugriffsrechte werden im letzten Block jedes Sektors, dem sogenannten Sektor-Trailer, definiert. Abbildung 2.5 auf Seite 20 zeigt das vorgegebene Speicherlayout einer Mifare-Classic-Karte und ist in dieser Form gültig für Mifare Classic 1k (16 Sektoren), Mifare Mini (5 Sektoren) und die ersten 32 Sektoren von Mifare Classic 4k (die restlichen 8 Sektoren der Classic 4k sind größer).

Block 0 in Sektor 0 jeder Karte ist permanent schreibgeschützt. Die Rechte für die anderen Blöcke ergeben sich aus dem Schlüssel, der in der mutual authentication verwendet wurde, der Blocknummer und dem Wert des ACL-Felds im für den Block zuständigen Sektor-Trailer. Es gibt keinen Speicherzugriff ohne vorherige mutual authentication und bei jedem mutual-authentication-Vorgang muss der Block (als durchgehende Nummer von 0 beginnend, d.h. Block 0 in Sektor 1 wird als Block 5 angegeben) und der zu verwendende Schlüssel angegeben werden. Im Lieferzustand sind alle Schlüssel einer Karte auf einen bestimmten Wert gesetzt: Je nach Hersteller entweder FF FF FF FF FF FFh für beide Schlüssel (Infineon) oder A0 A1 A2 A3 A4 A5h für Schlüssel A und B0 B1 B2 B3 B4 B5h für Schlüssel B (NXP).

Um einzelne gekippte Bits in den ACLn-Feldern zu erkennen und bestimmte Arten von Angriffen8 erkennen zu können, werden alle Bits dieser Felder doppelt gespeichert: einmal invertiert und einmal nicht invertiert. Die Karte prüft bei jedem Speicherzugriff, ob dieses Format eingehalten wurde und sperrt den betroffenen Sektor unwiderruflich, falls sie einen Formatverstoß findet.

Pro Block i in Sektor n finden sich 3 Bits im ACLn-Feld: C1i, C2i und C3i (sowie zusätzlich die invertierten Bits C1i, C2i, C3i). Aus diesen Bits leiten sich die Zugriffsbedingungen nach zwei Tabellen ab: Eine für den Sektor-Trailer (Tabelle 2.3 auf Seite 22) und eine für die anderen Blöcke (Tabelle 2.4 auf Seite 23). Neben den ACLn-Feldern gibt es in Byte 9 jedes Sektor-Trailers noch ein unbenutztes Byte, welches von der Karte nicht ausgewertet wird und dem Benutzer frei zur Verfügung steht. Für Zwecke der Zugriffskontrolle gehört dieses Byte zum ACL-Feld, es gelten also immer die gleichen Berechtigungen für die Bytes 6 bis 9 (inklusive) jedes Sektor-Trailers.

8Ein Standardangriff auf Smartcard-Systeme besteht darin, die Chipoberfläche teilweise freizulegen und dann gezielt mit UV-Licht zu bestrahlen, was in der Regel dazu führt, dass die EEPROM-Speicherzellen im bestrahlten Bereich zurückgesetzt (alle 0 oder alle 1) werden. [BS96]
<table>
<thead>
<tr>
<th>Kartentyp</th>
<th>EEPROM</th>
<th>Verschlüsselung</th>
<th>Kommunikation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Brutto</td>
<td>Netto</td>
<td>crypto</td>
</tr>
<tr>
<td>Classic</td>
<td>Classic 1k</td>
<td>1024 B</td>
<td>752 B</td>
</tr>
<tr>
<td></td>
<td>Classic 4k</td>
<td>4096 B</td>
<td>3440 B</td>
</tr>
<tr>
<td></td>
<td>Mini</td>
<td>320 B</td>
<td>224 B</td>
</tr>
<tr>
<td></td>
<td>Lightc</td>
<td>48 B</td>
<td>24 B</td>
</tr>
<tr>
<td></td>
<td>Plus (alt)c</td>
<td>8 kB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Ultralight</td>
<td>64 B</td>
<td>48 B d</td>
</tr>
<tr>
<td></td>
<td>ProX</td>
<td>4 kB, 8 kB oder 16 kB</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>SmartMX</td>
<td>bis 72 kB</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>DESfire</td>
<td>4 kB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DESfire ev1</td>
<td>2 kB, 4 kB oder 8 kB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Plus (neu)</td>
<td>✓</td>
<td>-</td>
</tr>
</tbody>
</table>

a) Kontaktlos, ISO 14443-3 Typ A
b) Kontaktbasiert, ISO 7816-3
c) Historisch
d) 4 B OTP (One-Time Programmable, nur einmal beschreibbar)
e) Optional, per Emulation
f) Austattungsabhängig
2. RFID

Abbildung 2.5. Mifare-Classic-Speicherlayout

<table>
<thead>
<tr>
<th>Sektor</th>
<th>Block</th>
<th>Byte innerhalb des Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>UID</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Schlüssel A&lt;sub&gt;0&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Schlüssel A&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Schlüssel A&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup>Prüfsumme, byteweises XOR der vier Bytes aus UID

Abbildung 2.6. Format der Zugriffsbedingungen im Mifare Classic Sektor-Trailer

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 6</td>
<td>C&lt;sub&gt;3&lt;/sub&gt;</td>
<td>C&lt;sub&gt;2&lt;/sub&gt;</td>
<td>C&lt;sub&gt;1&lt;/sub&gt;</td>
<td>C&lt;sub&gt;0&lt;/sub&gt;</td>
<td>C&lt;sub&gt;13&lt;/sub&gt;</td>
<td>C&lt;sub&gt;12&lt;/sub&gt;</td>
<td>C&lt;sub&gt;11&lt;/sub&gt;</td>
<td>C&lt;sub&gt;10&lt;/sub&gt;</td>
</tr>
<tr>
<td>Byte 7</td>
<td>C&lt;sub&gt;13&lt;/sub&gt;</td>
<td>C&lt;sub&gt;12&lt;/sub&gt;</td>
<td>C&lt;sub&gt;11&lt;/sub&gt;</td>
<td>C&lt;sub&gt;10&lt;/sub&gt;</td>
<td>C&lt;sub&gt;33&lt;/sub&gt;</td>
<td>C&lt;sub&gt;32&lt;/sub&gt;</td>
<td>C&lt;sub&gt;31&lt;/sub&gt;</td>
<td>C&lt;sub&gt;30&lt;/sub&gt;</td>
</tr>
<tr>
<td>Byte 8</td>
<td>C&lt;sub&gt;33&lt;/sub&gt;</td>
<td>C&lt;sub&gt;32&lt;/sub&gt;</td>
<td>C&lt;sub&gt;31&lt;/sub&gt;</td>
<td>C&lt;sub&gt;30&lt;/sub&gt;</td>
<td>C&lt;sub&gt;23&lt;/sub&gt;</td>
<td>C&lt;sub&gt;22&lt;/sub&gt;</td>
<td>C&lt;sub&gt;21&lt;/sub&gt;</td>
<td>C&lt;sub&gt;20&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Da es den Grund für getrennte Schlüssel ad absurdum führen würde, wenn man mit Schlüssel A auf Schlüssel B lese zugreifen könnte, ist Schlüssel B in den Konfigurationen, in denen er gelesen werden kann, deaktiviert. Ein Authentisierungsversuch mit Schlüssel B wird in diesem Fall abgewiesen. Im Prinzip können die 6 Byte, die normalerweise für Schlüssel B reserviert sind, dann als normaler Benutzerspeicher verwendet werden. Bei einem Lesevorgang auf den Sektor-Trailer, bei dem nicht alle drei Felder lesbar sind, werden die unlesbaren Felder als 00h-Bytes gelesen.

Die zur Verfügung stehenden Zugriffsbedingungsvarianten erlauben eine Konfiguration in drei Dimensionen:

1. Blöcke können lese/schreibbar sein, nur lesbar oder komplett gesperrt.
2. Blöcke können als Wertblöcke definiert werden, für die ein besonderes Format und besondere Kommandos (Inkrement/Dekrement) gelten, siehe den Abschnitt dazu ab Seite 25.

Die Kombination aus dem ersten und dem dritten Punkt ergibt beispielsweise die Möglichkeit, dass Authentisierung mit Schlüssel A nur zum Lesen berechtigt, während Authentisierung mit Schlüssel B auch Schreibzugriffe ermöglicht. Schlüssel A könnte dann veröffentlicht werden und jedermann das Lesen der betreffenden Blöcke erlauben, während nur der Inhaber von Schlüssel B sie ändern kann.


Die Kombination der ersten beiden Punkte wiederum erlaubt nicht-wiederaufladbare Wertkarten, deren Wert nach der Erstausgabe nur dekrementiert werden kann.

Da der erste Punkt in letzter Konsequenz auch für den Sektor-Trailer gilt, kann ein ganzer Sektor komplett und unwiderruflich gesperrt werden: Man setzt die Blöcke 0-2 unlesbar/unschreibbar und macht dann im Sektor-Trailer nur die Zugriffsbedingungen lesbar aber nicht beschreibbar.

2.4.4. Mifare-Classic-Kommandoablauf

Mifare Classic setzt auf dem standardisierten ISO-14443-3-Protokoll ein proprietäres Mifare-spezifisches Protokoll für Kommandos und Antworten ein. Genaue Details dieses Protokolls auf Funkebene sind nicht vollständig öffentlich dokumentiert und mussten teilweise über Reverse Engineering gewonnen werden ([KGHG08]). Lediglich die Teilmenge der Kommandos die auch bei den Mifare-Ultralight-Tags verwendet wird, ist offiziell dokumentiert([NXP08a]).

Die Kommandos zur Antikollision, Selektion und Deselektion wurden in Abschnitt 2.3.3 auf Seite 8 beschrieben, hier folgt daher nur eine Übersicht über die Mifare-Classic-spezifischen Kommandos.

Authentisierung

Die gegenseitige Authentisierung (mutual authentication) soll garantieren, dass Lesegerät und Karte im Besitz eines gemeinsamen geheimen Schlüssels sind. Die Karte sendet eine Anfrage an das Lesegerät, welche dieses korrekt beantworten muss. Das Lesegerät sendet seinerseits eine Anfrage an die Karte, welche diese ebenfalls korrekt beantworten muss.

### Tabelle 2.3 Bedeutung der Zugriffsbefugnisse in den Sektor-Failers

<table>
<thead>
<tr>
<th>Zugriffsberechtigung</th>
<th>Schlüssel A</th>
<th>Schlüssel B</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>verboten</td>
<td>verboten</td>
<td>verboten</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>verboten</td>
<td>verboten</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>lesen</td>
<td>lesen</td>
<td>lesen</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Schluessel B lesen</td>
<td>lesen</td>
<td>lesen</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Schluessel A lesen</td>
<td>lesen</td>
<td>lesen</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Auslieferung</td>
<td>lesen</td>
<td>lesen</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Tabelle 2.4. Bedeutung der Zugriffsbedingungen für normale Blöcke

<table>
<thead>
<tr>
<th>Zugriffsbits</th>
<th>Zugriff auf</th>
<th>Abbuchen, Transfer, Restore</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lesen</td>
<td>Schreiben</td>
</tr>
<tr>
<td>C1 C2 C3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>Schl. A oder B(^a)</td>
<td>Schl. A oder B(^a)</td>
</tr>
<tr>
<td>0 0 1</td>
<td>Schl. A oder B(^a)</td>
<td>verboten</td>
</tr>
<tr>
<td>0 1 0</td>
<td>Schl. A oder B(^a)</td>
<td>verboten</td>
</tr>
<tr>
<td>0 1 1</td>
<td>Schlüssel B(^a)</td>
<td>Schlüssel B(^a)</td>
</tr>
<tr>
<td>1 0 0</td>
<td>Schl. A oder B(^a)</td>
<td>Schlüssel B(^a)</td>
</tr>
<tr>
<td>1 0 1</td>
<td>Schlüssel B(^a)</td>
<td>verboten</td>
</tr>
<tr>
<td>1 1 0</td>
<td>Schl. A oder B(^a)</td>
<td>Schlüssel B(^a)</td>
</tr>
<tr>
<td>1 1 1</td>
<td>verboten</td>
<td>verboten</td>
</tr>
</tbody>
</table>

\(^a\)Wenn Schlüssel B im Sektor-Trailer gelesen werden kann, kann er nicht zur Authentisierung verwendet werden.
Abbildung 2.7: Kommunikation zwischen Host, Leser-IC und Karte bei der Authentisierung mit Schlüssel $x \in \{A, B\}$

<table>
<thead>
<tr>
<th>Host</th>
<th>Leser-IC</th>
<th>Karte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schlüssel</td>
<td>OK</td>
<td>AUTH1$x$(Block), UID</td>
</tr>
<tr>
<td>OK</td>
<td>AUTHENT2</td>
<td></td>
</tr>
<tr>
<td>AUTH1$x$(Block)</td>
<td>$r_B$</td>
<td></td>
</tr>
<tr>
<td>$r_A$, RESPONSE($r_A$,$r_B$)</td>
<td>RESPONSE($r_A$,$r_B$)</td>
<td></td>
</tr>
<tr>
<td>OK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

zu erst und die Karte antwortet. Da der Leser-Chip keine Funkkommunikation von sich aus anstößt, muss er insgesamt zwei Kommandos erhalten, um eine komplette Authentisierung durchzuführen. Wenn die mutual authentication erfolgreich ausgeführt wurde, haben Karte und Leser-IC einen gemeinsamen Zustand in ihren Crypto-Schaltkreisen, der als Sitzungs schlüssel wirkt und alle Kommunikation wird transparent ver- und entschlüsselt.

Um die Authentisierung einzuleiten, dient entweder das AUTH1A- oder AUTH1B-Kommando. Diese Kommandos erhalten als Parameter den eigentlichen Schlüssel, die Nummer des zu lesendes Blocks sowie die UID der Karte (letztere Information wird nur im Leser-IC verwendet und nicht an die Karte weitergesendet). Die Wahl des Schlüssels A oder B erfolgt implizit durch die Wahl des Kommandos AUTH1A oder AUTH1B. Im Erfolgfall antwortet die Karte mit einer zufälligen nonce und erwartet, dass sich das Lesegerät zuerst authentisiert.

Der Leser-IC erhält dann das AUTHENT2-Kommando ohne Parameter, welches den zweiten Teil der Mifare-Authentisierung startet: Der Leser-IC sendet seine Antwort auf die Challenge der Karte sowie eine eigene Nonce, woraufhin die Karte im Erfolgsfall die passende Response übermittelt, welche vom Leser-IC überprüft wird.

Dieser Vorgang ist in Abbildung 2.7 vereinfacht dargestellt. Die Nomenklatur der Nonces ($r_A$ und $r_B$) entspricht dabei der die in ISO 9798-2 ([ISOc]) Abschnitt 5.2.2 verwendet wird.

---

9 In der Regel ist es möglich, den Schlüssel in einem internen EEPROM (Electrically erasable programmable read-only memory, Elektrisch löscharer, programmierbarer Nur-Lese-Speicher) im Leser-IC abzulegen, so dass statt des Schlüssels nur noch der Speicherindex im EEPROM angegeben werden braucht.

10 Es ist ein generelles Designschema von mutual-authentication-Protokollen, dass die „stärkere“ Partei sich zuerst gegenüber der „schwächeren“ Partei als berechtigt ausweist, bevor diese beginnt, Ressourcen zu verbrauchen, um sich gegenüber Ersterer auszuweisen.
2.4. Mifare

Abbildung 2.8. Wertblock-Format auf Mifare-Classic-Karten

<table>
<thead>
<tr>
<th>Byte-Nummer</th>
<th>0 . . . 4 . . . 8 . . . 12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inhalt</td>
<td>Wert</td>
<td>Wert</td>
<td>Wert</td>
<td>Addr</td>
</tr>
</tbody>
</table>

Lesen und Schreiben von Datenblöcken

Die Lese- und Schreibkommandos arbeiten jeweils auf Blöcken von 16 Byte. Der einzige Parameter des Lesekommandos ist die Blocknummer (von 0 beginnend durchgehend gezählt), worauf die Karte mit den 16 Byte Inhalt des Blocks antwortet, falls der Zugriff gestattet wurde. (Im Fall eines Sektor-Trailers werden alle unlesbaren Felder ausmaskiert und als 00h gelesen.) Das Schreibkommando hat zwei Parameter: Blocknummer und 16 Byte Daten.

Verwenden von Wertblöcken


Increment hat als Argumente eine Blocknummer \( i \) und einen Wert \( m \) (Ganzzahl) und lädt das Ergebnis der Addition des Wertes aus Block \( i \) und \( m \) in ein internes Zwischen-speicherregister.

Decrement hat ebenfalls eine Blocknummer und einen Wert als Argumente, lädt aber das Ergebnis der Subtraktion in das interne Speicherregister.

Transfer hat nur eine Blocknummer als Argument und schreibt den Wert des internen Speicherregisters in den Block mit der angegebenen Nummer.

Restore hat auch nur eine Blocknummer als Argument, aber lädt den Wert des angegebenen Blocks in das interne Speicherregister.
Wertblock-Operationen, wie Aufwerten oder Abbuchen, müssen immer von einem *Transfer*-Kommando gefolgt werden, um einen Effekt auf den EEPROM-Inhalt der Karte zu haben. Das *Restore*-Kommando kann im Zusammenhang mit dem *Transfer*-Kommando benutzt werden, um einen Block zu Sicherungszwecken zu kopieren, also Laden aus Block \( x \) und Schreiben nach Block \( y \) mit \( x \neq y \). Dazu ist nur die *Decrement/Restore/Transfer*-Berechtigung erforderlich und nicht die wesentlich mächtigere Schreib-Berechtigung, die es ermöglichen würde, den Block bei der Kopie zu verändern.
Verwendete Hardware

OpenPCD und OpenPICC sind Schwesterprojekte für ein offenes RFID-Lesegerät bzw. einen offenen RFID-Emulator, entwickelt von Harald Welte, Milosch Meriac und Brita Meriac, bei denen sowohl die Firmware als auch die Baupläne (Schaltplan, Platinenlayout, Gerber Files) offen zugänglich und unter einer freien Lizenz veröffentlicht sind (GPL für die Firmware, Creative Commons für die Baupläne).

Von kommerziellen RFID-Lesern hebt sich OpenPCD zusätzlich dadurch ab, dass es direkten und bequemen Zugang zu vielen Zwischensignalen erlaubt, womit die Analyse der übertragenen Daten mit externen Hilfsmitteln (Logikanalysator oder Oszilloskop) leichter fällt. Außerdem bedeutet die offene Firmware sehr weitgehende Einflussmöglichkeiten über das Kommunikationsprotokoll. OpenPCD ist weiterhin gut als Hilfsmittel beim Design eines RFID-Schaltkreises geeignet, da es nicht nur direkten Zugang auf die empfangenen Signale erlaubt, sondern auch hilfreiche Testsignale zum Test und Kalibration der Analogteile der Empfängerseite senden kann (etwa ein PWM-Signal$^1$).

OpenPICC seinerseits kann beim Design eines RFID-Lesers helfen, da es ebenfalls direkten Zugang zu den empfangenen Signalen und die Möglichkeit zum Senden von Testsignalen liefert.


Die Firmware für beide Projekte entstammte ursprünglich einem gemeinsamen Quellcodebaum, aber im Laufe dieser Diplomarbeit habe ich für den OpenPICC eine neue Firmware

$^1$PWM (Pulse Width Modulation, Pulsweitenmodulation)
3. Verwendete Hardware

Abbildung 3.1. OpenPCD-Platine


Einen Übersichtsschaltplan mit allen relevanten Details zeigt Abbildung 3.2 auf der nächsten Seite. Der Trigger-Ausgang ist ein Digitalausgang, der über ein Firmwarekommando kurzzeitig auf High geschaltet werden kann und dazu dient, auf Wunsch ein Oszilloskop auf bestimmte Zeitpunkte in der Firmware-Verarbeitung synchronisieren zu können.

Der Mikroprozessor im OpenPCD wird von einem 18,432 MHz-Quarz getaktet, welcher über eine interne PLL²-Schaltung zu einer Arbeitsfrequenz von etwa 48 MHz führt.

²PLL (Phase Locked Loop, Phasenregelschleife)
3.1. OpenPCD

Abbildung 3.2. Vereinfachter Übersichtsschaltplan für OpenPCD

3.1.1. RC632


Zur Unterstützung/Fehlersuche beim Design bietet der RC632 zwei verschiedene Testausgänge an:

**AUX** Auf diesen Ausgang kann ein ausgewähltes analoges Testsignal gelegt werden, welches den Abgriff (und das anschließende Darstellen auf einem Oszilloskop) von verschiedenen Zwischenschritten bei der internen analogen Signalverarbeitung (hauptsächlich des Empfangssignals) ermöglicht.

**MFOUT** Auf diesen Ausgang können verschiedene digitale Testsignale gelegt werden, um Zwischenschritte der digitalen Signalverarbeitung darzustellen. Darüberhinaus kann MFOUT zusammen mit MFIN für etwas genutzt werden, was NXP *Active Antenna* nennt, siehe nächster Abschnitt.

\(^3\)IC (*Integrated Circuit*, Integrierter Schaltkreis)  
\(^4\)IRQ (*Interrupt Request*, Unterbrechungsanforderung)
3. Verwendete Hardware

3.1.2. MFIN/MFOUT


MFIN und MFOUT sind aber auch mit dem Mikroprozessor verbunden, was es unter anderem erlaubt, beliebige 13,56 MHz-basierte Protokolle zu implementieren. Der Mikroprozessor muss dazu nur das MFOUT-Signal abtasten und die passende Dekodierung für das jeweilige Protokoll in Software durchführen sowie ein passendes, kodiertes Sendesignal an einem Digitausgang bereitstellen. (MFIN ist außerdem mit einem speziellen Pin des Mikroprozessors verbunden, der ein beliebiges pulswellenmoduliertes Signal ausgeben kann.) Um Sende- bzw. Empfangssignale in diesem Modus mit dem Trägersignal synchron halten zu können (eine Vorraussetzung für die meisten Protokolle), sind die internen Timer-Schaltkreise des Mikroprozessors auch an den 13,56 MHz-Quarz des RC632 angeschlossen und ermöglichen synchrone Datenaus- bzw. Eingabe mit einstellbaren Phasenlagen und Teilerverhältnissen.

3.2. OpenPICC

OpenPICC (dargestellt in Abbildung 3.3 auf der nächsten Seite) ist ein RFID-Emulator und -Testwerkzeug, welches in der Lage sein sollte, die meisten 13,56 MHz-basierten Protokolle zu implementieren. Eine grobe Übersicht über die wichtigsten funktionalen Bestandteile
zeigt Abbildung 3.4. Der Mikroprozessor und die Kommunikationsschnittstellen (USB, UART, zwei LEDs) entsprechen den Fähigkeiten des OpenPCD. OpenPICC hat aber im Gegensatz zum OpenPCD keinen integrierten Schaltkreis für das Funk-Interface sondern benutzt ein diskret aufgebautes Analogfrontend.

Das Format der OpenPICC-Platine entspricht ID-1 und die Antenne ist auf der Leiterplatte als umlaufende Leiterbahnen implementiert, wie bei 'normalen' RFID-Karten üblich.
3. Verwendete Hardware

3.2.1. Analogteil

Der Analogteil der Schaltung besteht aus den notwendigen Bauteilen für die eingebaute 13,56 MHz-Antenne, der Amplitudendemodulationsschaltung sowie dem Lastmodulations teil.

Amplitudendemodulator

Zur Amplitudendemodulation wird das Antennensignal zunächst mit einer Diode durch Abschneiden einer Wellenhälfte gleichgerichtet und dann in einer mehrstufigen OpAmp\textsuperscript{5}-Schaltung verstärkt und tiefpassgefiltert. Das entstehende Analogsignal stellt die Hüllkurve des einkommenden Funkfeldes dar. Es ist auf einer Buchse ausgeführt, um einfach auf einem Oszilloskop darstellbar zu sein.


Lastmodulator

Der Lastmodulator besteht im Prinzip aus einem Widerstand (der Last), welcher über einen Feldeffekttransistor schaltbar mit der Antenne verbunden ist. Für größere Flexibilität enthält die OpenPICC-Platine zwei parallel geschaltete Lastmodulatoren mit unterschiedlich starken Widerständen (560 Ω und 1,2 kΩ). Der Mikroprozessor kann einen oder beide dieser Modulatoren auswählen und dann, gesteuert über das MOD-Signal, mit der Antenne verbinden.

3.2.2. Digitalteil

Der Digitalteil besteht hauptsächlich aus dem Trägerregenerator (siehe aber auch Abschnitt 3.2.4 auf Seite 35) und dem Mikroprozessor mit einiger interner und externer Beschaltung.

Trägerregeneration


Zu diesem Zweck enthält die OpenPICC-Platine eine PLL-Schaltung, welche auf die Trägerfrequenz locked und dem Trägersignal dann phasengenau folgt. Sobald der Amplitudendemodulator eine Modulation des Trägersignals detektiert (der Träger vom PCD

\textsuperscript{5}Operational Amplifier, Operationsverstärker
3.2. OpenPICC

also kurz abgeschaltet wird), wird die Regelschleife der PLL unterbrochen und sie überbrückt die Ausfallszeit als normaler Frequenzgenerator. Wenn das Trägerfeld wiederkehrt, wird die PLL-Regelschleife wieder geschlossen und die PLL resynchronisiert sich auf das Trägersignal.

Auf diese Art wird (in der Theorie zumindest) ein kontinuierlicher Trägertakt zu jeder Zeit gewährleistet.

**Prozessor**

Der Prozessor wird mit einem 18,432 MHz-Quarz versorgt, was (über eine interne PLL) zu einer Taktfrequenz von ca. 48 MHz führt.

OpenPICC macht ausführlichen Gebrauch davon, dass die Mikroprozessoren der AT91SAM7-Reihe nicht nur den reinen Prozessorkern, sondern auch viele integrierte Peripheriegeräte enthalten. Das betrifft nicht nur die Schaltungen für den USB- und UART-Anschluss, sondern (unter anderem) auch einen PWM-Ausgabecontroller, einen SSC (Synchronous Serial Controller, serieller synchroner Controller) sowie mehrere Timer.

**Timer-Design**

Es werden zwei Timer verwendet: Ein Timer (tc_cdiv) ist als Frequenztweiler konfiguriert und gibt ein Ausgabesignal aus, welches aus dem regenerierten Trägertakt, geteilt durch einen konfigurierbaren Divisor (mit einstellbarer Phase), besteht. Der andere Timer (tc_fdt) ist für die FDT zuständig und zählt die Trägertakte seit dem Ende des empfangenen Frames. Er aktiviert ein Signal (TF, für Transmit Frame), welches verwendet werden kann, um mit dem Senden zu genau der richtigen Zeit zu beginnen.

**Empfangs- und Sendeperipherals**

Es kann entweder der PWM-Controller oder der SSC mit dem Modulationssignal MOD verbunden werden.

Wenn der PWM-Controller (gesteuert vom regenerierten Trägertakt) verwendet wird, kann das OpenPICC ein Testsignal auf das Trägerfeld modulieren, welches den Test und die Kalibration des verwendeten RFID-Lesegeräts ermöglicht.

Wenn der SSC mit dem Modulationssignal verbunden ist, kann das OpenPICC beliebige (digitale) Wellenformen mit einer Samplerate von bis zu $f_c/2$ senden und empfangen.

Der SSC ist prozessorintern mit einem DMA-Controller verbunden, um Speichertransfers ohne direkte Mitwirkung des Prozessorkerns zu ermöglichen. Er hat außerdem fünf Signale zur Außenwelt: Eingang (vom Amplitudendemodulator), Ausgang (zum Lastmodulator), Takt (vom tc_cdiv), Empfangsstart (genannt FRAME) und Sendestart (von TF). Das Empfangsstart-Signal ist über einen Flip-Flop als Flankendetektor mit dem Amplitudendemodulator verbunden, wird also (wenn es vom Prozessor zurückgesetzt wurde) bei der ersten einkommenden Flanke ausgelöst und bleibt dann aktiv, bis es wieder vom Prozessor zurückgesetzt wird.
3.2.3. Empfangs- und Sendeablauf

Abbildung 3.5 zeigt den generellen Empfangs- und Sendeablauf (MFOUT ist das gesendete modified-Miller-Modulationssignal, CLOCK ist der tc_cdiv-Ausgang, TF ist das Transmit-Frame-Signal, MOD ist das Modulationssignal; RED und GREEN sind die beiden Leuchtdioden, die zu Debugging-Zwecken verschiedene Positionen im Firmware-Ablauf anzeigen) am Beispiel eines ISO-14443A-REQA und ATQA:

- Im Ruhezustand ist das FRAME-Signal nicht ausgelöst und der SSC für eine einkommende Übertragung bereit. Der tc_cdiv-Teiler ist auf ein Teilungsverhältnis von 64 eingestellt.

- Sobald die erste Modulationsflanke am Ausgang des Amplitudendemodulators erkannt wird, wird das FRAME-Signal ausgelöst. Der SSC beginnt jetzt, bei jeder steigenden Flanke des tc_cdiv-Signals ein Bit vom Amplitudendemodulator zu sam-
3.2. OpenPICC

Abbildung 3.6. OpenPICC-Platine v0.2 mit Modifikationen

plen und (über den Umweg eines Schieberegisters) in den RAM des Prozessors zu laden.

Die Firmware erhält beim Start des Empfangs einen IRQ und beobachtet den Fortschritt des Empfangs während sie auf das Ende der Datenübertragung vom PCD wartet.

Mit der letzten empfangenen Flanke wird tc_fdt zurückgesetzt und beginnt zu zählen.

Die Firmware wertet den Speicherinhalt, der vom SSC beschrieben wurde aus und führt dort eine modified Miller-Dekodierung durch. Sie setzt den Zielwert des tc_fdt-Zählers entsprechend den Vorgaben des Standards (Abschnitt 2.3.3 auf Seite 8).

Die Firmware bereitet ihre zu sendende Antwort vor, indem sie die zu modulierende Wellenform (also Manchster-kodierte Subträger-Modulation) in den Speicher schreibt und die Startadresse und Länge an den SSC übergibt. Außerdem setzt die Firmware den tc_cdiv-Teiler neu auf 8. Damit kann der \( \frac{f_c}{16} \)-Subträger für eine halbe Bitperiode erzeugt werden, indem die Bitfolge 10101010 gesendet wird.

An der steigenden TF-Flanke (durch Ablauf der FDT) beginnt der SSC mit dem Senden der eingestellten Bitfolge.

3.2.4. Probleme und Modifikationen

Die ursprüngliche Version des OpenPICC funktionierte nicht zufriedenstellend nach den theoretischen Vorstellungen, die dem Design zugrunde lagen. Das betraf sowohl den Empfang von längeren Übertragungen, als auch das Senden eigener Übertragungen nach der vom Standard vorgeschriebenen exakten FDT. Ich habe daher mehrere Veränderungen am Design vorgeschlagen und getestet (eine modifizierte OpenPICC-Variante zeigt Abbildung 3.6), die in die nächste Platinenversion eingehen werden:
Trägerregeneration

Die Trägerregeneration hat sich als unzuverlässig und instabil erwiesen. Bereits nach nur 3 empfangenen Trägermodulationen begann die PLL instabil zu werden und Regelschwingungen aufzuweisen. Das genaue Timing des tc_cdiv-Taktes zur Benutzung als SSC-Eingabetakt während der Empfangsphase war nicht gewährleistet. Das hätte durch starkes Oversampling – also die Erfassung von deutlich mehr Samples pro Bit als theoretisch minimal notwendig – ausgeglichen werden können, würde dann aber zu erheblich vergrößertem Dekodieraufwand in der Firmware führen.


In der modifizierten OpenPICC-Version ist die Trägerregeneration vollständig entfernt und der Trägertakt ist direkt mit dem Mikroprozessor (und den Timern darin) verbunden.

Empfang

Um dennoch Daten nach ISO 14443 Typ A empfangen zu können, kommt eine veränderte Empfangsmethodologie – ursprünglich vorgeschlagen von Milosch Meriac – zum Einsatz: Es wird der tc_fdt-Timer benutzt, um die Anzahl der Trägertakte zwischen zwei Modulationen zu zählen. Dazu wird der Timer so geschaltet, dass er am Ende einer Modulation automatisch zurückgesetzt wird (was ohnehin für die Nutzung als FDT-Timer passiert) und dann ein IRQ im Mikroprozessor ausgelöst, sobald der Anfang einer Modulation erkannt wird. Der IRQ-Handler liest den Timerwert zu Beginn der Modulation aus und speichert diese Zeitmessungen in einer Liste zur späteren Verarbeitung.

Aus dieser Liste von Zeitmessungen kann dann das vollständige Datensignal rekonstruiert werden, da die genauen Längen der Modulationen für die Dekodierung nicht relevant sind.

Senden

Das ausgehende Modulationssignal stört den Amplitudendemodulator und wird als einkommende Modulation detektiert. Das hat Einfluss auf andere Teile der Schaltung, die von diesem Signal abhängig sind, zum Beispiel den tc_fdt. Um das zu verhindern, wird der Ausgang des Komparators mit dem Mikroprozessor steuerbar verbunden und kann von diesem für die Dauer der ausgehenden Sendung deaktiviert werden.

Teil II.

Analyse
Zeitlinie


2. Januar 2008  Erster niederländischer Online-Bericht zum Mifare Classic Reverse Engineering ([Win08]).


10. März 2008  Erste kryptanalytischen Ergebnisse zu Bias in der Filterfunktion von Mifare Classic ([Noh08]).

10. März 2008  NXP stellt Mifare Plus (neu) vor ([NXPo8b]).


15. April 2008  Vorstellung von algebraischen Attacken auf Mifare Classic ([CNOo8]).

Mitte Juni 2008  Wouter Teepe und Bart Jacobs von der Radboud-Universität demonstrieren einen Cloning-Angriff auf die Mifare-Classic-basierte Oyster-Card in London ([Lib08]).

18. Juli 2008  NXP versucht die Veröffentlichung der Forschungsergebnisse der Radboud-Gruppe per einstweiliger Verfügung zu untersagen, das Gericht entscheidet jedoch im Sinne der Forscher ([Digo8a]). Der Radboud-Angriff wird also voraussichtlich Anfang Oktober auf dem 13th European Symposium on Research in Computer Security (esorics08) veröffentlicht werden.

Das Amplitudendemodulatorsignal des OpenPICC und das MFOUT-Signal des OpenPCD sind abgegriffen worden und werden in einen Logikanalysator (nicht im Bild) geführt. Der Logikanalysator ist an einen PC angeschlossen und die aufgezeichneten Daten werden als CSV-Daten exportiert. Eine selbst geschriebene Software übernimmt die modified Miller- und Manchester-Dekodierung der Daten.

4.1. Erste Sniffing-Ergebnisse


Die Bedeutung der einzelnen Frames ist wie folgt:

1. REQA vom Lesegerät
2. ATQA von der Karte, zeigt an, dass die Karte bitorientierte Antikollision unterstützt

\footnote{CSV (Comma Separated Values, Kommaseparierte Werte)}
4. Analyse des Funk-Protokolls

Abbildung 4.1. Versuchsaufbau, um die Kommunikation des OpenPCD mit einer Karte bidirektional mitzuschneiden

3. ANTICOLLISION-Kommando vom Lesegerät. NVB ist 20h, also 2 Bytes und 0 Bits.
4. UID-Antwort der Karte, enthält die vollständige 4-Byte UID und das BCC-Byte
5. SELECT-Kommando vom Lesegerät mit 7 Bytes und 0 Bits (plus CRC)
6. ATS von der Karte, zeigt an, dass die UID vollständig ist (keine weiteren Kaskadierungszebenen) und die Karte nicht ISO-14443-4-konform ist
7. AUTH1A-Kommando vom Lesegerät, beginnt die mutual authentication für Block 0 in Sektor 0
8. Challenge von der Karte
9. Challenge und Response vom Lesegerät, verschlüsselt (daher inkorrekte Paritätsbits)
10. Response von der Karte, verschlüsselt
11. Lesekommando für Sektor 0 von der Karte, verschlüsselt (inkorrekte Paritätsbits und CRC)
### 4.2. Wiederholte Challenge-Response-Läufe


Ein interessantes Ergebnis war, dass sich bei diesem Vorgehen Karten-Challenges überdurchschnittlich häufig wiederholten und auch Lesegerät-Challenges nicht einzigartig sind. Das ist überraschend, denn da die Challenges jeweils aus 32 Bit bestehen, sollte die Wahrscheinlichkeit, zwei gleiche Challenges hintereinander zu erhalten, 1 zu $2^{32} \approx 4$ Milliarden sein.


In einem erweiterten Versuchslauf mit 27 Challenge-Response-Paaren (nicht abgebildet) fand ich 12 mal (44%) die Karten-Challenge 7D DA 7E 41 und 6 mal (22%) das Kartenchallenge-Lesegerätchallenge-Paar (7D DA 7E 41, 1E 98 73 FB ...).

---

**Tabelle 4.1.** Erste bidirektionale Sniffing-Ergebnisse einer Mifare-Classic-Sitzung, in der ein Sektor gelesen wird

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>7 Bits</td>
<td>L</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>+157</td>
<td>2 Bytes</td>
<td>K</td>
<td>04 00</td>
</tr>
<tr>
<td>3</td>
<td>+34158</td>
<td>2 Bytes</td>
<td>L</td>
<td>93 20</td>
</tr>
<tr>
<td>4</td>
<td>+270</td>
<td>5 Bytes</td>
<td>K</td>
<td>B4 79 F7 D7 ED</td>
</tr>
<tr>
<td>5</td>
<td>+46431</td>
<td>9 Bytes</td>
<td>L</td>
<td>93 70 B4 79 F7 D7 ED C7 27</td>
</tr>
<tr>
<td>6</td>
<td>+865</td>
<td>3 Bytes</td>
<td>K</td>
<td>08 B6 DD</td>
</tr>
<tr>
<td>7</td>
<td>+23127</td>
<td>4 Bytes</td>
<td>L</td>
<td>60 00 F5 7B</td>
</tr>
<tr>
<td>8</td>
<td>+492</td>
<td>4 Bytes</td>
<td>K</td>
<td>F3 FB AE ED</td>
</tr>
<tr>
<td>9</td>
<td>+10515</td>
<td>8 Bytes</td>
<td>L</td>
<td>7C 74 07 EB 0F 7B D5 1B</td>
</tr>
<tr>
<td>10</td>
<td>+775</td>
<td>4 Bytes</td>
<td>K</td>
<td>3B 0E A0 E2</td>
</tr>
<tr>
<td>11</td>
<td>+59213</td>
<td>4 Bytes</td>
<td>L</td>
<td>65 8D 65 1F</td>
</tr>
<tr>
<td>12</td>
<td>+449</td>
<td>18 Bytes</td>
<td>K</td>
<td>52 F6 46 35 BA E2 E9 B2 2D FB CD AE C8 6C B2 DE 04</td>
</tr>
</tbody>
</table>

12. Inhalt des Sektors 0 von der Karte plus CRC, verschlüsselt

---

2Das veranlasste mich zuerst, einen Fehler im Versuchsaufbau zu vermuten der evt. alte Aufzeichnungen wieder einspielen könnte. Das hat sich aber nicht bewahrheitet.
4. Analyse des Funk-Protokolls

Tabelle 4.2. Erste vier aufgezeichnete Challenge-Response-Paare

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Von der Karte</th>
<th>Vom Lesegerät</th>
<th>Von der Karte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FF CF 80 E3</td>
<td>3F C5 B5 45 44 D5 04 7F</td>
<td>DF 58 61 B3</td>
</tr>
<tr>
<td>2</td>
<td>7D DA 7E 41</td>
<td>7E 98 43 FB D6 CD 65 E5</td>
<td>A6 23 0A 9C</td>
</tr>
<tr>
<td>3</td>
<td>7D DA 7E 41</td>
<td>53 03 8F 3A 66 B5 D5 48 87 BE 75 D3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7D DA 7E 41</td>
<td>7E 98 43 FB D6 CD 65 E5</td>
<td>A6 23 0A 9C</td>
</tr>
</tbody>
</table>

Diese Beobachtungen legen nahe, dass der Zufallszahlengenerator der Karte (und des Lesegeräts) nur vom genauen Timing seit Einschalten der Stromversorgung abhängt.

4.3. Protokollmanipulationen

Eines der ersten Zwischenergebnisse aus dem physikalischen Reverse Engineering (Kapitel 5 ab Seite 47) war, dass die Kryptographielogik nicht genügend Eingänge hat, um sowohl die Karten-UID als auch den geheimen Schlüssel bei der Initialisierung während der Mutual Authentication zu verwenden.


Bereits das erste Experiment mit einer modifizierten UID zeigte, dass die Authentisierung nicht mehr funktioniert, wenn eine andere UID als die der Karte verwendet wird.

Da aber der interne Zustand nur 48 Bits groß ist und sich als Funktion aus dem verwendeten geheimen Schlüssel (48 Bit) und der UID (32 Bit) ergibt, muss es unterschiedliche Kombinationen von UID und Schlüssel geben, die den gleichen Initialzustand erzeugen. Um diese Annahme zu bestätigen und den genauen Zusammenhang zwischen UID und Schlüssel herauszufinden, habe ich die Firmware erweitert, um auch den geheimen Schlüssel zu modifizieren, bevor er an das Authentisierungskommando übergeben wird.

Meine erste Theorie bestand darin, dass UID und Schlüssel paarweise XOR-verknüpft werden, bevor sie in den Initialzustand der Chiffre eingehen. Um das zu testen, habe ich, ausgehend von den korrekten Werten für Schlüssel und UID, jeweils ein Bit in UID und Schlüssel gekippt. Das funktioniert mit den Bits 0 bis 4, d.h. wenn man Bit 0 der UID ändert und Bit 0 im Schlüssel ändert, funktioniert die Authentisierung (und damit die Ver/Entschlüsselung), obwohl die UID und der Schlüssel nicht den Werten der Karte entsprechen.

Wenn man Bit 5 der UID und Bit 5 des Schlüssels kippt, funktioniert die Authentisierung allerdings nicht, UID und Schlüssel sind also nicht einfach paarweise XOR-verknüpft. Ich habe meine Suche auf Zwei-Bit-Modifikationen ausgedehnt und festgestellt, dass wenn
man Bit 5 im Schlüssel kippt, man die Bits 0 und 5 in der UID kippen muss. Kippt man Bit 6 im Schlüssel, muss man die Bits 1 und 6 in der UID kippen, usw.

Ich habe den Vorgang automatisiert (entsprechend Algorithmus 4.1) und damit für jedes der ersten 31 Schlüsselbits die Menge der zugehörigen UID-Bits gefunden. Tabelle 4.3 auf der nächsten Seite zeigt den gesamten Verlauf des Versuchs für die Schlüssel-Bits 0 bis 31 (ab Schlüsselbit 10 werden die Fehlversuche nicht mehr extra aufgeführt).

Algorithmus 4.1 Algorithmus, um die zu den Schlüsselbits zugehörigen UID-Bits zu finden

```
function BitFlippen(a, b) // Flippe Bit b im Bitstring a
  return a\ldots a_{b-1} \overline{a}_b a_{b+1}\ldots a_{\text{Länge}(a)}
end function

procedure FindEUIDBits (UID, KEY)
  M ← ∅  // Menge der momentan zu kippenden UID Bits
  i ← 0
  repeat
    k ← BitFlippen(\text{KEY}, i)
    u ← UID
    for j ∈ M do
      u ← BitFlippen(u, j)
    end for
    Authentisiere mit UID u und Schlüssel k
    if Authentisierung erfolgreich then
      Gebe i und M aus
      i ← i + 1
      M ← \{j + 1 | j ∈ M\} // Erhöhe die Nummer aller Bits in M
    else
      if 0 ∈ M then
        return // Fehler
      else
        M ← M ∪ \{0\}  // Kippe das 0. Bit zusätzlich zu den bisherigen Bits
      end if
    end if
  until i = 32
end procedure
```

Wie später in Abschnitt 6.2 ab Seite 52 zu erkennen sein wird, erlauben diese Ergebnisse einen Rückschluss auf ein bestimmtes Strukturelement der Chiffre, nämlich die Feedback-Taps des Zustands-LFSR. Man vergleiche dazu auch die mit einem → markierten Schlüsselbitoffsets mit den Bitoffsets in der Funktion mifare_update in Listing A.3 auf Seite 77.
4. Analyse des Funk-Protokolls

**Tabelle 4.3. Verlauf des Experiments, die zu den Schlüsselbits zugehörigen UID-Bits zu finden**

<table>
<thead>
<tr>
<th>Bit im Schlüssel</th>
<th>Bits in UID</th>
<th>Erfolg</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>→</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>→</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>→</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>→</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>→</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>→</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>→</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>→</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>→</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>→</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>→</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>→</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>2</td>
</tr>
</tbody>
</table>
Kapitel 5

Physikalisches Reverse Engineering

Um die Struktur eines unbekannten Verschlüsselungsalgorithmus aufzuklären, von dem man mindestens eine Implementierung besitzt, gibt es im Prinzip drei Wege:

- **Software-Reverse-Engineering**, also die Beschaffung und Analyse des Programmcodes, falls der Kryptoalgorithmus als Firmware oder Software vorliegt. Dazu gab es in der Vergangenheit einige Beispiele, etwa die Chiffren A5/1 und A5/2 die in GSM¹-Kommunikation eingesetzt werden ([And94]). Auch der CSS²-Algorithmus wurde durch genaue Analyse eines Software-DVD-Players gewonnen ([Pat99]).

- Bei manchen Algorithmen führt auch eine reine *Black-Box*-Analyse – also nur das Beobachten der Ein- und Ausgänge, evt. kombiniert mit gezieltem Setzen der Eingänge, ohne die Implementierungsdetails beobachten zu können – zum Ziel. In der Vergangenheit war das etwa beim DST (*Digital Signature Transponder*) von Texas Instruments der Fall ([BGS+05]).

- Wenn der Algorithmus in Hardware implementiert ist, kann er durch Analyse dieser Hardware rekonstruiert werden.

Da es von Crypto-1 keine Firmware- oder Softwareimplementierung gab, war diese letzte Möglichkeit, das Hardware Reverse Engineering, der einzige verbleibende Weg ([NEsPo8, KNPo8]).


5.1. Chip öffnen und fotografieren

Um mit dem Chip arbeiten zu können, muss dieser zunächst aus dem Kartenkörper gelöst werden³. Dazu eignen sich rauchende Salpetersäure oder Aceton (letzteres ist weniger

---

¹GSM (*Global System for Mobile communication*)
²CSS (*Content Scrambling System*)
³Oder man bestellt gleich rohe Mifare-Classic-Chips ohne Antenne oder Kartenkörper.
5. Physikalisches Reverse Engineering

Abbildung 5.1. Mifare-Classic-Karte nach 30 Minuten in Aceton

gefahrenlich und daher zu bevorzugen). Abbildung 5.1 zeigt einen Chip, der sich nach ca. einer halben Stunde in Aceton vom Kartenmaterial getrennt hat.

Um die einzelnen Schichten, aus denen der Chip besteht, analysieren zu können, wird jetzt Schicht für Schicht mechanisch (mit einer Poliermaschine) herunterpoliert. Dabei muss darauf geachtet werden, dass der Materialabtrag immer parallel zu den Chipebenen stattfindet. Wenn der Chip schief poliert wird, erhält man Schnittbilder, die durch mehrere Schichten gehen. Um das zu verhindern, wird der Chip in ein Plastikmaterial eingebettet. Leichte Schieflagen können später korrigiert werden, indem man mehrere schief Schnittbilder nimmt und nur die Teile, die dieselbe Ebene zeigen, miteinander kombiniert.


Nachdem alle 6 Ebenen des Mifare-Classic-1k-Chips fotografiert sind, werden die Bilder der Ebenen zueinander ausgerichtet und können dann analysiert werden. Die vollständige Erfassung aller relevanten Details erforderte das Abschleifen und fotografieren von etwa einem Dutzend Chips, um Schleiffehler und schlechte Fotos ausgleichen zu können. Abbildung 5.2 auf der nächsten Seite zeigt ausschließlich ein Beispiel für jede Ebene: 1 Deckschicht (mit einigen wenigen Verbindungen), 3 Verbindungsschichten, 1 Logikschiht und 1 Transistorschicht.

5.2. Fotos auswerten

Die unteren beiden Schichten enthalten die eigentlichen n- und p-dotierten Halbleiter und damit die aktiven Komponenten des Chips. Aus Analyse dieser beiden Schichten
Abbildung 5.2. Die 6 Ebenen des Mifare-Classic-1k-Chips

Karsten Nohl hat eine Software geschrieben, welche als Eingabe ein Schichtbild und je ein Beispielbild (Template, für Beispiele siehe [sil]) für jeden Gatetyp nimmt und dann automatisch alle Instanzen dieses Types auf der gesamten Schicht identifiziert (Spiegelungen und Drehungen werden berücksichtigt). Auf diese Art entsteht eine Art Landkarte des Chips.

Mit dieser Karte kann nun auch der Bereich des Chips, der zu analysieren ist, eingeschränkt werden. Dazu sucht man nach auffälligen Bereichen mit bestimmten Kriterien, die auf eine Verschlüsselungslogik hinweisen: Lange Reihe von Flip-Flops (das ist das Zustandsshiftregister), viele XOR-Gates (kommen in normaler Kontrolllogik nur selten vor, aber sehr häufig in Kryptoalgorithmen), Bereiche am Rand des Chips, die sehr eng untereinander vernetzt sind, aber kaum Verbindungen zum Rest haben.

Um schließlich die eigentliche Verschaltung der Gates miteinander zu rekonstruieren, haben Jan Krisler und Karsten Nohl in langer Handarbeit die Leiterbahnen, die die Gates miteinander verbinden über die drei Verbindungsschichten hinweg verfolgt und aufgezeichnet. Dieser Vorgang sollte in Zukunft automatisiert werden, da das manuelle Verfolgen fehleranfällig und mühsam ist.

Das Ergebnis dieser Arbeit ist die vollständige Rekonstruktion des Zufallszahlengenerators und des Crypto-1-Schlüsselstromgenerators. Diese Bestandteile werden im nächsten Kapitel beschrieben.
Kapitel 6

The enemy knows the system.

Claude Shannon

Analyse-Ergebnisse

6.1. Pseudozufallszahlengenerator

Mifare-Classic-Karten enthalten einen PRNG (Pseudo Random Number Generator, Pseudozufallszahlengenerator) als Zufallszahlengenerator für die mutual-authentication-Phase. Das betrifft nur die Classic-Karten, nicht die Classic-Emulationen in ProX/SmartMX (siehe 2.4.2 auf Seite 16) und ebenfalls nicht Mifare Plus (neu), da diese Kartentypen mit einem echten Hardware-Zufallszahlengenerator ausgestattet sind.

Ein Zufallszahlengenerator (Pseudo- oder Hardware-) ist nötig, um die Nonce/Challenge während der Mutual-Authentication-Phase zu erzeugen. Wenn die Zufallszahlen vorhersehbar oder sogar kontrollierbar sind, bricht das viele Aspekte der mutual authentication und ermöglicht zum Beispiel Replay-Angriffe.

Der PRNG der Mifare-Classic-Karten ist ein LFSR (Linear Feedback Shift Register, Linear rückgekoppeltes Schieberegister) der Länge 16 Bit, mit der maximalen Periodenlänge von $2^{16} - 1$. Es hat die Form

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

und wird mit dem ISO-14443-Bit-Takt von $\frac{13.56\text{MHz}}{128} \approx 105,9$ kHz getaktet.

Da das Register nur $2^{16} - 1$ Zustände hat, wiederholen sich die Zustände des Registers nach

$$\frac{2^{16} - 1}{\frac{13.56\text{MHz}}{128}} \approx 0,618619 \text{ s}$$

Der PRNG wird, kurz nachdem die Karte mit genügend Strom über die Luftschnittstelle versorgt wurde, initialisiert, indem das Shift-Register auf einen festen Wert gesetzt wird (101010...). Anschließend wird das Shift-Register alle 128 Träger-Takte weitergetaktet, solange wie die Karte genügend Energie erhält. Wenn die Karte eine Zufallszahl für die mutual-authentication-Prozedur benötigt, wird das Ausgabebit des Shiftregisters für 32 Takte abgegriffen und in einem Register gespeichert, der Wert dieses Registers wird dann als Karten-Nonce im Protokoll verwendet.

$^1$Der Zustand der in dem alle Bits 0 sind ist nicht gültig.
6. Analyse-Ergebnisse

**Tabelle 6.1.** Karten-Nonces, in drei Sitzungen gesniff, mit Offset relativ zu einem willkürlich gewählten Startpunkt

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits (in Sendereihenfolge)</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>03 2E 33 19</td>
<td>1100 0000 0111 0100 1100 1100 1001 1000</td>
<td>1675</td>
</tr>
<tr>
<td>01 97 99 0C</td>
<td>1000 0000 1110 1001 1001 1001 0011 0000</td>
<td>1676</td>
</tr>
<tr>
<td>7B 90 E9 8A</td>
<td>1101 1110 0000 1001 0111 0101 1000</td>
<td>63859</td>
</tr>
</tbody>
</table>

**Beispiel** Tabelle 6.1 zeigt zwei Karten-Nonces, die unabhängig voneinander ungefähr zur gleichen Zeit nach der Aktivierung der Stromversorgung beobachtet wurden (bei 1,01736624 s und 1,01735047 s nach Aktivieren der Lesegerätestromversorgung) sowie eine dritteNonce, die etwa eine Viertel PRNG-Periode später (bei 1,16035723 s nach Aktivieren der Lesegerätestromversorgung) beobachtet wurde.

**Listing 6.1** Implementierung eines PRNG, äquivalent zum in Mifare Classic benutzten PRNG, adaptiert nach dem Beispiellisting in [Wiko8]

```c
#include <sys/types.h>
#include <stdio.h>
#define INITIAL 0xACE1

int main(void) {
    u_int16_t reg = INITIAL;
    u_int16_t bit;
    do {
        bit = (reg & 0x0001) ^
             ((reg & 0x0004) >> 2) ^
             ((reg & 0x0008) >> 3) ^
             ((reg & 0x0020) >> 5);
        reg = (reg >> 1) | (bit << 15);
        printf("%i",

    /* Alternative Ausgabeformate:
    * printf("%04X\n",reg);
    * for(int i=0; i<16; i++) printf("%i", !(reg & (1<<i)));
    * printf("\n");
    */
    } while(reg != INITIAL);
    return 0;
}
```

Eine mögliche Implementierung für einen äquivalenten LFSR in C ist in Listing 6.1 abgebildet, der Startwert ist dabei willkürlich gewählt.

6.2. Crypto-1-Chiffre
Abbildung 6.1. Übersicht über Crypto-1

Challenge | Response | Schlüsselstrom

PRNG | ID | + | + | + | + | 48-bit LFSR | f(.)

Abbildung 6.2. Detailansicht der Filterfunktion \( f(\cdot) \) von Crypto-1

fa(.) | fb(.) | fa(.) | fa(.) | fb(.) | fc(.)

48-bit LFSR


Zur Initialisierung und mutual authentication (auf der Kartenseite) wird das Zustandsshifregister mit dem geheimen 48-Bit Schlüssel initialisiert. Dann werden vom PRNG 32 Bits an 32 aufeinanderfolgenden Bittakten generiert und bitweise mit der UID XOR-verknüpft bei aktiviertem LFSR-Feedback in das Zustandshifregister geschoben (während dieser Details zu den LFSR-Taps vormals nicht explizit veröffentlicht, aber die Positionen der ersten 13 von 18 Taps können direkt aus den Ergebnissen von Abschnitt 4.3 auf Seite 44 abgeleitet werden: Jedesmal, wenn ein neues Bit in die Menge der zu kippenden UID-Bits eingeht, ist ein Feedback-Tap gefunden.)
6. Analyse-Ergebnisse


Zur Datenverschlüsselung werden alle aus- bzw. eingehenden Bits jeweils mit dem nächsten Bit des Schlüsselstroms XOR-verknüpft. Das Shiftregister wird dabei nur für die Daten- und CRC-Bits weitergetaktet, für die Paritätsbits wird ein Schlüsselstrombit verwendet, das bereits für ein Datenbit verwendet wurde.

Die Verschlüsselung findet im Schichtenmodell unterhalb der Paritätsbits und CRC statt, daher ist in verschlüsselten Datenübertragungen die CRC auf dem Funkkanal in der Regel falsch und etwa die Hälfte der Paritätsbits stimmen nicht. CRC- und Paritätsprüfung können also beim Empfänger erst nach der Entschlüsselung durchgeführt werden.
Teil III.

Angriffe

Wherein I show you how deep the rabbit hole goes.
Kapitel 7

Angriffe auf das Funk-Protokoll

7.1. Online Brute Force – Rohe Gewalt

Vor den Ereignissen aus Abschnitt II ab Seite 39 war der einzige öffentlich diskutierte Angriff auf Mifare Classic das simple Durchprobieren aller möglichen Schlüssel oder zumindest eines bestimmten Satzes von Standardschlüsseln. Grunwald ([Gru06]) hat dabei die Dauer eines Authentisierungsversuchs mit kommerzieller Standardhardware und eigener Software auf etwa 25 ms bestimmt. Da jeder der beiden Schlüssel 48 Bit lang ist, ergeben sich \(2^{48} \approx 281,5 \cdot 10^{12}\) verschiedene Möglichkeiten für jeden Schlüssel. Wenn man nur einen der beiden Schlüssel eines Sektors durch Ausprobieren in Erfahrung bringen möchte, ergeben sich damit

\[
2^{48} \text{ Schlüssel} \cdot 25 \frac{\text{ms}}{\text{Schlüssel}} = 7,036,874,417,766,4 \text{ s} \approx 81,445,306 \text{ Tage} \approx 222,985 \text{ Jahre}
\]

, um alle möglichen Schlüssel im Versuchsaufbau von Grunwald auszuprobieren (bei 365,25 Tagen pro Jahr). Falls die Wahrscheinlichkeit für alle Schlüssel gleich ist, darf im Erwartungswert damit gerechnet werden, dass der korrekte Schlüssel nach der Hälfte der Zeit, also ca. 111.493 Jahren gefunden wird.

Grunwalds Versuchsaufbau lässt noch einige Optimierungsmöglichkeiten offen: Wenn die Software, die die Schlüssel durchprobiert, „näher“ an den Leser-IC herangebracht wird, also in die Lesegerätefirmware integriert wird, fällt der Kommunikationsoverhead zwischen Lesegerät und Host-PC sowie eventuelle Latenzen auf dem Host-PC weg. Dann lassen sich Versuche so schnell ausführen, wie von der Karte erlaubt. [Mif06] rechnet beispielhaft vor, mit welcher minimalen Dauer pro Versuch dabei zu rechnen ist: 4 ms für die Selektion der Karte plus 2 ms für den eigentlichen Authentisierungsversuch gleich 6 ms pro Versuch. Damit reduziert sich die Zeit, die für alle Schlüssel gebraucht wird, auf ca. 53 516 Jahre, im Erwartungswert ca. 26 758 Jahre.

Die Offenheit der OpenPCD-Firmware (siehe Abschnitt 3.1 auf Seite 28) ermöglicht dieses Vorgehen und bereits eine leichte Modifikation der Standard-Firmware war in der Lage, Authentisierungen alle 8 ms durchzuführen. Mit weiteren Optimierungen sollte die theoretische 6 ms-Grenze problemlos erreicht werden.

Um diesen Angriff weiter zu beschleunigen, kann man versuchen, die Arbeitsgeschwindigkeit der Karte zu erhöhen. Ossmann ([Oss06]) beschreibt einen experimentellen RFID-
7. Angriffe auf das Funk-Protokoll

Abbildung 7.1. Einfacher Relay-Aufbau nach Hancke ([Han05])

Reader ohne dedizierten Reader-IC, der eine freiere Wahl der Arbeitsfrequenz ermöglicht und berichtet, dass Mifare-Karten unter Umständen mit bis zu 16 MHz getaktet werden können. Da der interne Takt der Karte aus dem Trägerfeld generiert wird und mit diesem synchron ist, erhöht sich damit auch der Takt der Karte um ca. 18%, was die Zeit pro Authentisierungsversuch auf 5 ms und den Erwartungswert für das Finden des Schlüssels auf ca. 22 676 Jahre senkt. Es ist allerdings anzumerken, dass dieser experimentelle RFID-Reader ohne einen NXP-Reader-IC aufgebaut ist und daher nicht über eine Crypto-1-Implementierung verfügt. Ob dieser Angriff auch mit einem Reader mit dem Reader-IC möglich ist, also ob sich der Reader-IC ähnlich „übertakten“ lässt, ist fraglich. Ein Angreifer müsste also eine eigene Crypto-1-Implementierung haben, also Crypto-1 kennen. Und in diesem Fall sind wesentlich bessere Angriffe möglich, siehe Abschnitt 8 ab Seite auf Seite 65.

7.2. Relay-Attacken

Eine sehr schwer zu eliminierende Klasse von Angriffen stellen Relay-Attacken dar. Dabei verwendet ein Angreifer ein PCD (in diesem Fall bekannt als leech oder mole) und einen PICC-Emulator (genannt ghost oder proxy) mit einer direkten Verbindung zueinander. Der ghost wird in die Nähe des zu attackierenden Lesegeräts (etwa eine Kasse oder U-Bahn-Eingangssperre) gebracht, während der leech an die zu attackierende Karte gehalten wird. Da die Verbindung zwischen Angreifer und Karte drahtlos ist, kann diese sich dabei noch im Besitz ihres rechtmäßigen Eigentümers befinden, etwa in einer Tasche.

Hancke ([Han05]) beschreibt eine kostengünstige, minimale Relay-Schaltung: die Signale der angegriffenen Karte bzw. des angegriffenen Lesegeräts werden lediglich demoduliert und ohne Dekodierung direkt auf einem bidirektionalen UHF-Funkkanal übertragen. Ein solcher einfacher Aufbau ist in Abbildung 7.1 dargestellt.

Hancke verwendet dazu auf der Mole-Seite ein RFID-Reader-Modul auf Basis des Philips (jetzt NXP) MF RC530, vergleichbar dem MF RC632 im OpenPCD, und macht sich dessen MFIN/MFOUT-Funktion zunutze (siehe Abschnitt 3.1.2 auf Seite 30), um einen Subcarrier-Demodulator mit minimalem Aufwand implementieren zu können. Hancke’s Proxy unterscheidet sich vom OpenPICC (Abschnitt 3.2) hauptsächlich dadurch, dass letz-
terer keinen expliziten Subcarrier-Generator hat, sondern diese Funktion vom Prozessor übernommen wird.


Kfir und Wool ([KW05]) beschreiben eine Methode, um einen Proxy und Mole (dort als Ghost und Leech bezeichnet) noch einfacher zu konstruieren: Statt selbst gebauter Hardware schlagen sie vor, NFC-Geräte zu benutzen, die bereits von Haus aus über die Fähigkeit verfügen, sowohl Lesegerät als auch Karte in einer ISO-14443-basierten Kommunikation zu sein und darüberhinaus eine einfache API anbieten, diese Funktionalität zu nutzen. Dieser Weg hat zwei Nachteile: Der Abstand zwischen Ghost und Lesegerät sowie zwischen Leech und Karte ist auf ca. 10 cm beschränkt. Zudem funktioniert dieses Vorgehen nur mit Protokollen, die auf ISO 14443-4 aufsetzen, betrifft also nicht Mifare Classic. Für das erste Problem beschreiben und berechnen sie allerdings Hardware-Erweiterungen, welche die Reichweite unter Umständen drastisch erhöhen können und die auf den Hancke-Ansatz übertragbar sind. Damit ergibt sich ein maximaler Abstand zwischen Mole/Leech und Karte von etwa 40 cm und zwischen Proxy/Ghost und Lesegerät von etwa 50 m.

7.2.1. Gegenmaßnahmen


Es gibt in der Literatur theoretische Ansätze für ein Distance Bounding auf Basis der Lichtgeschwindigkeit ([HK05, BC93]), das benötigt jedoch spezielle Hardware auf der Tag-Seite und wird zur Zeit in keinem System praktisch eingesetzt.

Wenn die Gegenmaßnahme nicht darauf abzielen soll, den Aufenthaltsort der Karte exakt einzuzgrenzen, sondern nur dazu dienen soll, das Einverständnis des Karteninhabers sicherzustellen, gibt es zwei Möglichkeiten: Wenn die Karte in einer Schutzhülle mit der Funktion eines Faradayischen Käfigs aufbewahrt wird und vom Inhaber nur zu tatsächlichen Transaktionen aus dieser Hülle geholt wird, wird ein Relay-Angriff auf das Zeitfenster 1

1In der Praxis ist er vermutlich noch geringer. Eigene Experimente mit dem Nokia 6131 NFC zeigen einen Arbeitsabstand im Bereich von 2 bis 4 cm.
7. Angriffe auf das Funk-Protokoll

eingeschränkt, in dem die Karte nicht in der Schutzhülle ist. Unter Umständen reicht statt einer Hülle auch bereits eine gewisse Menge Metall (etwa ein Blatt Alufolie), welches die Resonanzfrequenz der Antenne soweit verändert, dass die Karte nicht mehr ordnungsgemäß funktionieren kann. Hier ist aber darauf zu achten, dass der Angreifer die Frequenz seines Feldes frei wählen kann.

Weiterhin besteht zumindest prinzipiell die Möglichkeit, die Antenne der Karte nur über einen (Folien-)Taster mit dem Chip zu verbinden, so dass der Inhaber aktiv den Taster betätigen muss, um die Transaktion zu bestätigen. Hier muss eventuell noch genauer untersucht werden, wie sich dieser zusätzliche Widerstand auf das Resonanzverhalten und die Performance der Kartenantenne auswirkt.


7.3. Man-in-the-Middle-Attacken

Eine Schwäche im Mifare-Funkprotokoll wird in [Han05] erstmals indirekt erwähnt2: Crypto-1 ist eine Stromchiffre, die jedes Bit unabhängig von allen anderen Bits mit dem Schlüsselstrom verknüpft. Wenn ein Angreifer den Klartext kennt (oder gut rät), kann er den verschlüsselten Bitstrom gezielt manipulieren, wodurch beim Entschlüsseln ein manipulierter Klartext entsteht. Mifare setzt keinen MAC (Message Authentication Code) ein, sondern nur eine CRC (Cyclic Redundancy Check). Es ist bekannt, dass eine solche Kombination nur Vertraulichkeit zusichern kann, nicht aber Integrität ([BGW01]).

CRCs sind immer linear, d.h. es gilt

\[ \text{CRC}(n \oplus x) = \text{CRC}(m) \oplus \text{CRC}(x) \]

für alle Werte von \( n \) und \( x \) (wobei \( \oplus \) die XOR-Operation ist) ([SPMR06, BGW01]). Ähnliches gilt für Stromchiffren wie Crypto-1. Crypto-1 besteht aus einem Schlüsselstromgenerator \( g \), der einen Schlüsselstrom generiert, welcher mit dem Klartext XOR-verknüpft wird, um den Geheimtext zu erhalten bzw. mit dem Geheimtext XOR-verknüpft wird, um den Klartext zu erhalten:

\[
\begin{align*}
c & = g(k) \oplus m \\
m' & = g(k) \oplus c \\
& = g(k) \oplus g(k) \oplus m \\
& = m
\end{align*}
\]

Dabei sind \( m \) der Klartext, \( c \) der Geheimtext, \( m' \) der beim Empfänger wiederhergestellte Klartext und \( k \) der Schlüssel.

---

2Weiterhin beschreibt Hancke ([Han06]) FPGA-basierte Proof-of-Concept-Hardware, die in der Lage wäre, den hier beschriebenen Angriff durchzuführen, ohne aber auf Mifare Classic einzugehen.
Im Mifare-Classic-Funkprotokoll wird die CRC an die zu übertragende Nachricht $n$ angehängt und dann die Gesamtnachricht verschlüsselt:

\[
m = n \parallel \text{CRC}(n) \\
c = g(k) \oplus (n \parallel \text{CRC}(n))
\]

Ein Angreifer kann jetzt ein $c' = c \oplus y$ konstruieren, mit $y = x \parallel \text{CRC}(x)$ und erhält dann:

\[
c' = c \oplus (x \parallel \text{CRC}(x)) \\
= (g(k) \oplus (n \parallel \text{CRC}(n))) \oplus (x \parallel \text{CRC}(x)) \\
= g(k) \oplus ((n \oplus x) \parallel (\text{CRC}(n) \oplus \text{CRC}(x))) \\
= g(k) \oplus ((n \oplus x) \parallel (\text{CRC}(n \oplus x)))
\]

welches beim Entschlüsseln den Klartext $m''$ mit korrekter CRC ergibt:

\[
m'' = g(k) \oplus c' \\
= g(k) \oplus g(k) \oplus ((n \oplus x) \parallel (\text{CRC}(n \oplus x))) \\
= (n \oplus x) \parallel \text{CRC}(n \oplus x)
\]

Die Nachricht, die nach dem Entschlüsseln und Überprüfen der CRC empfangen wird, ist also $n' = n \oplus x$.


7. Angriffe auf das Funk-Protokoll

Tabelle 7.1. Häufige Kommandosequenzen und die zu beobachtenden Längen der übertragenen Daten, teilweise nach de Koning Gans et al. ([KGHG08])

<table>
<thead>
<tr>
<th>Kommando(sequenz)</th>
<th>Beobachtbare Längen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sektoren lesen</td>
<td>Lesegerät: 4 Byte</td>
</tr>
<tr>
<td></td>
<td>Karte: 18 Byte</td>
</tr>
<tr>
<td>Sektoren schreiben</td>
<td>Lesegerät: 4 Byte</td>
</tr>
<tr>
<td></td>
<td>Karte: 4 Bit</td>
</tr>
<tr>
<td></td>
<td>Lesegerät: 18 Bytes</td>
</tr>
<tr>
<td></td>
<td>Karte: 4 Bit</td>
</tr>
<tr>
<td>Aufwerten/Abbuchen</td>
<td>Lesegerät: 4 Byte</td>
</tr>
<tr>
<td></td>
<td>Karte: 4 Bit</td>
</tr>
<tr>
<td></td>
<td>Lesegerät: 6 Byte</td>
</tr>
<tr>
<td>Transfer</td>
<td>Lesegerät: 4 Byte</td>
</tr>
<tr>
<td></td>
<td>Karte: 4 Bit</td>
</tr>
</tbody>
</table>

Aufladen von 5 € führt beispielweise dazu, dass ein Aufwertkommando mit dem Argument 00 00 01 F4h (plus CRC) gesendet wird. Um daraus ein Aufladekommando für 15,24 € zu machen, muss der Angreifer nur ein Bit kippen: 00 00 05 F4h = 00 00 01 F4h ⊕ 00 00 04 00h.


7.4. Replay-Attacken

Der schwache und beeinflussbare Pseudozufallszahlengenerator der Mifare-Classic-Karten erlaubt Replay-Angriffe gegen die Karte. Ein Angreifer, der eine Authentisierung mit anschließender Kommunikation abgehört hat, kann aus der Karten-Nonce berechnen, an welchem Zeitpunkt nach dem Aktivieren des RF-Feldes er die Authentisierung beginnen muss, um die gleiche Nonce von der Karte zu erhalten. Wenn sich die Karte in seinem Besitz befindet, reicht es sogar, wenn er das Timing nicht in jedem Fall exakt einhalten kann, sondern nur mit nicht vernachlässigbarer Wahrscheinlichkeit. Wenn der Angreifer es geschafft hat, die Karte dazu zu bringen, die gleiche Nonce wieder zu verwenden, kann er die PCD-Seite der aufgezeichneten Kommunikation 1:1, oder verändert, abspielen.
Der erste Fall, die aufgezeichnete Kommunikation exakt wiederabzuspielen, ist für drei Szenarien interessant:

- Wenn es sich bei der Aufzeichnung um ein Aufwertkommando handelt („Addiere 5 € zum aktuellen Kartenwert”), kann er den Kartenwert beliebig weit erhöhen.

- Wenn es ein Schreibkommando mit einem festen Kartenwert ist (oder einer anderen abbuchbaren Ressource, etwa verbleibende U-Bahn-Benutzungen), kann er die Karte benutzen (wobei der Wert oder die Ressource reduziert wird) und anschließend wieder auf den ursprünglichen Wert wiederherstellen.

- Wenn es sich um ein Lesekomando handelt, kann der Angreifer Rückschlüsse auf Veränderungen der auf der Karte gespeicherten Daten ziehen. Das ist hilfreich beim Reverse Engineering (siehe auch Abschnitt 7.5) oder kann ein Privacy-Problem sein („Wurde diese U-Bahn-Karte seit dem letzten Lesen benutzt?").

Der zweite Fall, die Aufzeichnung vor dem Wiederabspielen zu verändern, führt direkt zu allen Möglichkeiten aus Abschnitt 7.3 ab Seite 60.

### 7.5. Keystream-Recovery

Einen interessanten Angriff auf Mifare Classic, ebenfalls basierend auf dem schwachen PRNG, zeigen de Koning Gans et al. ([KGHG08]). Sie verwenden die Proxmark-III-Hardware (sowohl als Reader-Emulator als auch als Zwei-Wege-Sniffer), um wiederholt identische Karten-Nonces zu erzeugen, was ultimativ zu identischen Keystreams führt. Das doppelte Benutzen desselben Schlüsselstroms in einer Stromchiffre für mehrere unabhängige Nachrichten führt zur Kompromittierung aller Nachrichten, wenn der Klartext einer der Nachrichten bekannt ist (wie Borisov et al. ([BGW01]) bereits gezeigt haben).

Wenn der Angreifer eine einzige Transaktion mitgeschnitten hat und sich darin ein Lesekommando befindet (siehe Tabelle 7.1 auf der vorherigen Seite), kann er diese Transaktion wieder abspielen (Abschnitt 7.4), aber dabei den Zielblock innerhalb des Sektors frei wählen (mit den Techniken aus Abschnitt 7.3).

Bei Mifare Classic ist der erste Block des ersten Sektors unbeschreibbar mit der UID (und einigen herstellerspezifischen Daten) gefüllt. Der Inhalt der ersten 5 Bytes dieses Blocks ist dem Angreifer bekannt, da die UID als Teil der Antikollisionsprozedur unverschlüsselt übertragen wird. Die restlichen 11 Bytes sindopaque, unbekannte Daten, wovon die ersten 5 Bytes aber bei den meisten Karten gleich und daher bekannt sind.

De Koning Gans et al. benutzen an dieser Stelle einen Trick, der auf die besondere Art der Zugriffsberechtigungen des Sektora-Trailers (Tabelle 2.3 auf Seite 22) zurückgeht: Schlüssel A ist niemals lesbar, der Klartext der ersten 6 Bytes des Sektora-Trailers ist beim Lesen also fest 00 00 00 00 00 00h. Schlüssel B ist nicht lesbar, wenn er als Schlüssel benutzt wird (was häufig zu erwarten ist), der Klartext ist in dem Fall auch für die letzten 6 Bytes des Sektor-Trailers fest. Für den ersten Sektor kann festgestellt werden, ob Schlüssel B lesbar ist, indem man die Zugriffsberechtigungen ausliest (was durch die geratenen ersten 5 Bytes des opaquen Datenteils im Herstellerblock ermöglicht wird). Für die restlichen Sektoren fehlt der Vorteil des bekannten Klartextes im ersten Block, so dass dort mit Garantie nur die ersten 6 Bytes
7. Angriffe auf das Funk-Protokoll

lesbar werden sowie in aller Regel auch die letzten 6 Bytes\(^3\), mit einer Lücke von 4 Bytes in der Mitte.

Diese Lücke kann unter Umständen durch intelligentes Raten des Klartextes aufgefüllt werden: Wenn einer der Blöcke im Sektor ein Wertblock ist (Abbildung 2.8 auf Seite 25), führt die Kenntnis der ersten 6 Bytes (über Schlüssel A im Sektor-Trailer) zur Kenntnis des exakten Werts des Blocks, was wiederum zur Kenntnis der ersten zwölf Byte führt (über die starre Wiederholungsstruktur des Blockformats). Komplette Kenntnis des Inhalts eines beliebigen Blocks führt naturgemäß in jedem Fall trivial zu genügend Schlüsselstrom, um alle Blöcke im selben Sektor zu lesen.

Es ist klar, dass diese Attacke nur sektorweise\(^4\) garantiert funktioniert: Wenn unterschiedliche Sektoren mit unterschiedlichen Schlüsseln gesichert sind, wird eine gesniffte Transaktion pro Sektor benötigt, um den Angriff zu bootstrappen. Wenn mehrere Sektoren gleiche Schlüssel verwenden, können hingegen bekannte Klartexte aus mehreren Sektoren kombiniert werden, um eine ausreichende Menge Schlüsselstrom zu erzeugen.

Wenn der Angreifer genügend bekannten Schlüsselstrom gesammelt hat, kann er nicht nur alle Blöcke in den betroffenen Sektoren lesen, sondern auch ändern, da der Schlüsselstrom von beiden Seiten (Karte und Lesegerät) gleichermaßen verwendet wird. Das alles ist ohne Kenntnis des Algorithmus oder von Algorithmendetails möglich.

7.5.1. Gegenmaßnahmen

Der Angriff wird durch einen echten Zufallszahlengenerator auf der Karte (wie für Mifare Plus (neu) angekündigt) stark erschwert bis unmöglich gemacht.

Verschlüsselung auf Applikationsebene, also zum Beispiel indem die Blockinhalte AES- verschlüsselt werden, kann einige Aspekte der Attacke unmöglich machen, bedeutet aber starke Eingriffe in die jeweilige Anwendung (beispielsweise können Wertblöcke nicht mehr verwendet werden). Außerdem hilft diese zusätzliche Verschlüsselung nicht gegen einen Angreifer, der den Blockinhalt (und damit ggf. das Guthaben) nach Benutzung wiederherstellt. Dagegen würde nur die konstante Prüfung des Karteninhalts gegen eine Online-Datenbank helfen, was das ganze Konzept, den Wert auf der Karte zu speichern ad absurdum führt.

\(^{3}\)Eine einfache Bestätigung, dass Schlüssel B nicht lesbar ist, besteht darin, dass er vom echten Lesegerät zur Authentisierung verwendet wird.

\(^{4}\)Die Sektornummer ergibt sich implizit aus der Blocknummer im AUTH1A- oder AUTH1B-Kommando, welches im Klartext übertragen wird.
Angriffe auf die Crypto-1-Chiffre

8.1. Schwächen in der Initialisierung


Da die UID und Nonce aber XOR-verknüpft werden und beide Werte 32 Bit lang sind, kann für jede Kombination von \((\text{UID}_x, \text{Nonce}_x)\) und \((\text{UID}_y, \text{Nonce}_y)\), wenn jeweils drei der Werte gegeben sind, der restliche Wert so gesetzt werden, dass der Initialzustand, der von beiden Paaren erzeugt wird, der gleiche ist.

Ein ähnliches Problem besteht im Zusammenhang von UID und Schlüssel: Da das Zustandsregister nur 48 Bit breit ist, der Schlüssel aber bereits 48 Bit hat, die UID 32 Bit und der Initialzustand des Registers sowohl von der UID als auch vom Schlüssel abhängt, muss es (verschiedene) Kombinationen von UID und Schlüssel geben, die den gleichen Initialzustand erzeugen. Bei gegebenem \((\text{UID}_x, \text{Schlüssel}_x)\) und \(\text{UID}_y\) ist es möglich, Schlüssel\(_y\) auszurechnen, so dass \((\text{UID}_x, \text{Schlüssel}_x)\) und \((\text{UID}_y, \text{Schlüssel}_y)\) den gleichen Initialzustand erzeugen. Die Verknüpfung ist hier aber nicht bitweises XOR, sondern wird durch die Rückkoppelung des Schieberegisters kompliziert. Die korrekte Verknüpfung kann jedoch (ohne Kenntnis des Algorithmus) iterativ bestimmt werden (siehe Abschnitt 4.3 auf Seite 44).

Durch diese einfache Verknüpfung von UID, Karten-Nonce und Schlüssel kann ein Angreifer, der einen MITM-Angriff (siehe Abschnitt 7.3 auf Seite 60) durchführt, die UID der Karte – ohne Kenntnis des Schlüssels oder des Algorithmus – fälschen, was je nach Einsatzszenario eine privilege escalation bewirkt oder zumindest etwaige Protokollaufzeichnungen mit der Karten-UID nutzlos werden lässt.
8. Angriffe auf die Crypto-1-Chiffre

8.2. Schwächen in der Schlüsselstromgenerierung

Fehlende Nichtlinearität


Schneller ist der Angriff aus Abschnitt 8.3; dieser kommt außerdem ohne Vorberechnung einer Tabelle aus.

Bias in der Filterfunktion

Nohl weist in [Noh08] auf einen Bias in der Filterfunktion von Crypto-1 hin. Für einige der Eingänge der Filterteilfunktionen \( f_a, f_b \) und \( f_c \) gilt: Wenn man den Eingang konstant (auf 0 bzw. 1) hält und alle Kombinationen für die anderen Eingänge durchgeht, hat der Ausgang in 5 von 8 bzw. 10 von 16 Fällen den gleichen Wert.

Durch gezielte Wahl der Karten-Challenge (bei Verwendung eines PICC-Emulators) und Beobachtung des ersten generierten Schlüsselstrombits (was durch die Zeitabhängigkeit der Lesegerät-Challenge ermöglicht wird), können dann Rückschlüsse auf die Eingänge des Filterblocks gezogen werden. Das erlaubt es, bis zu 32 Bit des Schlüssels in Erfahrung zu bringen, wofür allerdings sehr viele Challenge-Response-Paare nötig sind. Mit nur wenigen Challenge-Response-Paaren können aber immerhin bis zu 12 Bit des Schlüssels entdeckt werden, was den Suchraum für einen Brute-Force-Angriff auf 36 Bit senkt.

8.3. Algebraischer Angriff


Ermöglicht wird das vor allem durch die fehlende Nichtlinearität bei der Schlüsselstromgenerierung: Zwischen Ein- und Ausgabevariablen wird nur wenig Komplexität aufgebaut, was zu einfachen Gleichungen im Gleichungssystem führt.

Teil IV.

Abschluss
Zusammenfassung und Ausblick


Weiterhin hat die Arbeit gezeigt, dass Hardware-Reverse-Engineering ein ernstzunehmender Faktor ist und auch von Forschern mit beschränkten finanziellen Mitteln durchgeführt werden kann. Dies sollte bei allen Hardware-Designs berücksichtigt werden und Designs, die auf die Geheimhaltung der Hardwarefunktionalität angewiesen sind, sollten vermieden werden.

Ausblick

Alle beschriebenen Angriffe benötigen die Kooperation eines berechtigten Lesegeräts, entweder indirekt (durch Abhören der Kommunikation einer Karte mit dem Lesegerät) oder direkt (durch Verwenden eines Kartenemulators, der gewählte Challenges an das Lesegerät sendet). Das macht den Angriff weniger praktikabel, wenn pro Karte ein anderer Schlüssel verwendet wird und der pro Angriff erzielte Gewinn extrem niedrig ist (etwa wenn der Angriff nur auf die Privatsphäre des Karteninhabers, zum Beispiel zu Marktforschungszwecken, abzielt).

Es ist zu untersuchen, ob es nicht eventuell auch auf den Karten Implementierungs- oder Designfehler gibt, die einen direkten Angriff auf eine Karte ermöglichen. Eine solche Angriffsklasse wären zum Beispiel Seitenkanalattacken, die den Stromverbrauch oder das Timing während der Authentisierungsphase auswerten.

Listing A.1: crypto1.h – API und Globale Definitionen

```c
/*
 * Philips/NXP Mifare Crypto-1 implementation v1.0
 * By Karsten Nohl, Henryk Plötz, Sean O’Neil
 */
#ifndef CRYPTO1_H_
#define CRYPTO1_H_
#ifdef HAVE_SYS_TYPES_H
#include <sys/types.h>
#else
typedef unsigned int size_t;
#endif
#ifdef HAVE_STDINT_H
#include <stdint.h>
#else
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
#endif

/*
 * This type holds data bytes with associated parity bits.
 * The data is in the low byte while the associated parity bit
 * is in the least-significant bit of the high byte.
 */
typedef uint16_t parity_data_t;

struct _crypto1_state;

struct _crypto1_ops {
  void(*init)(struct _crypto1_state *state, const uint64_t key);
};
```

A. Crypto-1-Beispielimplementierung

void(*mutual_1)(struct _crypto1_state *state, const uint32_t uid, const uint32_t card_challenge);
void(*mutual_2_reader)(struct _crypto1_state *state, parity_data_t *reader_response);
int (*mutual_3_reader)(struct _crypto1_state *state, const parity_data_t *card_response);
int (*mutual_2_card)(struct _crypto1_state *state, const parity_data_t *reader_response);
void(*mutual_3_card)(struct _crypto1_state *state, parity_data_t *card_response);
void(*transcrypt_bits)(struct _crypto1_state *state, parity_data_t *data, size_t bytes, size_t bits);

typedef struct _crypto1_state {
    uint64_t lfsr; /* The 48 bit LFSR for the main cipher state and keystream generation */
    uint16_t prng; /* The 16 bit LFSR for the card PRNG state, also used during authentication. */
    uint8_t is_card; /* Boolean whether this instance should perform authentication in card mode. */
    const struct _crypto1_ops *ops;
} crypto1_state;

enum crypto1_cipher_implementation {
    CRYPTO1_IMPLEMENTATION_CLEAN,
    CRYPTO1_IMPLEMENTATION_OPTIMIZED
};

int crypto1_new(crypto1_state *state, uint8_t is_card, enum crypto1_cipher_implementation implementation);
void crypto1_init(crypto1_state *state, uint64_t key);
void crypto1_mutual_1(crypto1_state *state, uint32_t uid, uint32_t card_challenge);
int crypto1_mutual_2(crypto1_state *state, parity_data_t *reader_response);
int crypto1_mutual_3(crypto1_state *state, parity_data_t *card_response);
void crypto1_transcrypt(crypto1_state *state, parity_data_t *data, size_t length);
void crypto1_transcrypt_bits(crypto1_state *state, parity_data_t *data, size_t bytes, size_t bits);

int _crypto1_new_opt(crypto1_state *state);
int _crypto1_new_clean(crypto1_state *state);

/* Reverse the bit order in the 8 bit value x */
define rev8(x) (((x)>>7)&1) ^(((x)>>6)&1)<<1)^
    (((x)>>5)&1)<<2)^(((x)>>4)&1)<<3)^
    (((x)>>3)&1)<<4)^(((x)>>2)&1)<<5)^
    (((x)>>1)&1)<<6)^((x)&1)<<7)
/* Reverse the bit order in the 16 bit value x */
define rev16(x) (rev8(x)>>8) << 8)
/* Reverse the bit order in the 32 bit value x */
define rev32(x) (rev16(x)>>16) <<16))
/* Return the nth bit from x */

#define bit(x,n) (((x)>>>(n))&1)

/* Convert 4 array entries (a[0], a[1], a[2] and a[3]) into a 32 bit integer, */
/* where a[0] is the MSByte and a[3] is the LSByte */
#define ARRAY_TO_UINT32(a) (((uint32_t)((a)[0]&0xff) << 24) | \
  ((uint32_t)((a)[1]&0xff)<< 16) | \
  ((uint32_t)((a)[2]&0xff)<< 8) | \
  ((uint32_t)((a)[3]&0xff) << 0) )

/* Copy an uint32_t i into 4 array entries (a[0], a[1], a[2] and a[3]), where */
/* a[0] is the MSByte */
#define UINT32_TO_ARRAY(i, a) do{ \
  (a)[0] = ((i)>>24) & 0xff; \
  (a)[1] = ((i)>>16) & 0xff; \ 
  (a)[2] = ((i)>> 8) & 0xff; \ 
  (a)[3] = ((i)>> 0) & 0xff; \ 
}while(0);

/* Calculate the odd parity bit for one byte of input */
#define ODD_PARITY(i) (( (i) ^ (i)>>1 ^ (i)>>2 ^ (i)>>3 ^ \
  (i)>>4 ^ (i)>>5 ^ (i)>>6 ^ (i)>>7 ^ 1) & 0x01)

/* Like UINT32_TO_ARRAY, but put the correct parity in the 8th bit of each array */
/* entry (thus the array entries must at least be 16 bit wide) */
#define UINT32_TO_ARRAY_WITH_PARITY(i, a) do{ \
  (a)[0] = ((i)>>24) & 0xff; (a)[0] |= ODD_PARITY((a)[0])<<8; \ 
  (a)[1] = ((i)>>16) & 0xff; (a)[1] |= ODD_PARITY((a)[1])<<8; \ 
  (a)[2] = ((i)>> 8) & 0xff; (a)[2] |= ODD_PARITY((a)[2])<<8; \ 
  (a)[3] = ((i)>> 0) & 0xff; (a)[3] |= ODD_PARITY((a)[3])<<8; \ 
}while(0);

#endif /*CRYPTO1_H*/
#include <string.h>

#include "crypto1.h"

/*
* Create a new cipher instance of either card or reader side
*/
int crypto1_new(crypto1_state *state, uint8_t is_card, enum crypto1_cipher_implementation implementation)
{
    memset(state, 0, sizeof(*state));
    state->is_card = is_card;

    switch(implementation) {
    case CRYPTO1_IMPLEMENTATION_CLEAN:
        return _crypto1_new_clean(state);
    case CRYPTO1_IMPLEMENTATION_OPTIMIZED:
        #ifdef HAVE_OPTIMIZED_IMPLEMENTATION
            return _crypto1_new_opt(state);
        #else
            return 0;
        #endif
    }
    return 0;
}

/* Initialize a cipher instance with secret key */
void crypto1_init(crypto1_state *state, uint64_t key)
{
    state->ops->init(state, key);
}

/*
* First stage of mutual authentication given a card's UID.
* card_challenge is the card nonce as an integer
*/
void crypto1 Mutual_1(crypto1_state *state, uint32_t uid, uint32_t card_challenge)
{
    state->ops->mutual_1(state, uid, card_challenge);
}
* Second stage of mutual authentication.
* If this is the reader side, then the first 4 bytes of reader_response must
* be preloaded with the reader nonce (and parity) and all 8 bytes will be
* computed to be the correct reader response to the card challenge.
* If this is the card side, then the response to the card challenge will be
* checked.
* /
int crypto1_mutual_2(crypto1_state *state, parity_data_t *reader_response)
{
  if(state->is_card) {
    return state->ops->mutual_2_card(state, reader_response);
  } else {
    state->ops->mutual_2_reader(state, reader_response);
    return 1;
  }
}

* Third stage of mutual authentication.
* If this is the reader side, then the card response to the reader
* challenge will be checked.
* If this is the card side, then the card response to the reader
* challenge will be computed.
* /
int crypto1_mutual_3(crypto1_state *state, parity_data_t *card_response)
{
  if(state->is_card) {
    state->ops->mutual_3_card(state, card_response);
    return 1;
  } else {
    return state->ops->mutual_3_reader(state, card_response);
  }
}

* Perform the Crypto-1 encryption or decryption operation on 'length' bytes
* of data with associated parity bits.
* /
void crypto1_transcrypt(crypto1_state *state, parity_data_t *data, size_t length)
{
  crypto1_transcrypt_bits(state, data, length, 0);
}

* Perform the Crypto-1 encryption or decryption operation on 'bytes' bytes
* of data with associated parity bits.
* The additional parameter 'bits' allows processing incomplete bytes after the
* last byte. That is, if bits > 0 then data should contain (bytes+1) bytes where
* the last byte is incomplete.
* /
void crypto1_transcrypt_bits(crypto1_state *state, parity_data_t *data, size_t bytes, size_t bits)
{
  state->ops->transcrypt_bits(state, data, bytes, bits);
#include "crypto1.h"

// IMPLEMENTATION section ====================

/* Clock the prng register by n steps and return the new state, don’t */
* update the register.
* Note: returns a 32 bit value, even if the register is only 16 bit wide.
* This return value is only valid, when the register was clocked at least
* 16 times. */
static uint32_t prng_next(const crypto1_state * const state, const size_t n)
{
    uint32_t prng = state->prng;
    for (int i = 0; i < n; i++)
        prng = (prng<<1) | ( ((prng>>15)^(prng>>13)^(prng>>12)^(prng>>10)) & 1 );
    return rev32(prng);
}

/* These macros are linearized boolean tables for the output filter functions.
* E.g. fa(0,1,0,1) is (mf2_f4a >> 0x5)&1 */
#define i4(x,a,b,c,d) (((x)>>>(a)) & 1)<<0 |
                     (((x)>>>(b)) & 1)<<1 |
                     (((x)>>>(c)) & 1)<<2 |
                     (((x)>>>(d)) & 1)<<3

/* Return one bit of non-linear filter function output for 48 bits of */
* state input */
static uint32_t mf20 (const uint64_t x)
{
    const uint32_t d = 2; // number of cycles between when key stream is produced
    const uint32_t e = 3; /* and when key stream is used. */

    ...
const uint32_t i5 = ((mf2_f4b >> i4 (x, 7+d, 9+d,11+d,13+d)) & 1)<<0
| ((mf2_f4a >> i4 (x,15+d,17+d,19+d,21+d)) & 1)<<1
| ((mf2_f4a >> i4 (x,23+d,25+d,27+d,29+d)) & 1)<<2
| ((mf2_f4b >> i4 (x,31+d,33+d,35+d,37+d)) & 1)<<3
| ((mf2_f4a >> i4 (x,39+d,41+d,43+d,45+d)) & 1)<<4;
return (mf2_f5c >> i5) & 1;
}

static uint8_t mifare_update (struct _crypto1_state * const state,
const uint8_t injection, const uint8_t feedback)
{
const uint64_t x = state->lfsr;
const uint8_t ks = mf20(state->lfsr);

state->lfsr = (x >> 1) |
| ((x >> 0) ^ (x >> 5)
| (x >> 9) ^ (x >> 10) ^ (x >> 12) ^ (x >> 14)
| (x >> 15) ^ (x >> 17) ^ (x >> 19) ^ (x >> 24)
| (x >> 25) ^ (x >> 27) ^ (x >> 29) ^ (x >> 35)
| (x >> 39) ^ (x >> 41) ^ (x >> 42) ^ (x >> 43)
| injection ^ (feedback?ks:0) ) & 1) << 47);
return ks;
}

static uint8_t mifare_update_byte (struct _crypto1_state * const state,
const uint8_t injection, const uint8_t feedback)
{
uint8_t ret = 0;
for(int i = 0; i < 8; i++)
ret |= mifare_update(state, bit(injection, i), feedback) << i;
return ret;
}

static uint32_t mifare_update_word (struct _crypto1_state * const state,
const uint32_t injection, const uint8_t feedback)
{
uint32_t ret = 0;
for(int i = 3; i >= 0; i--)}
A. Crypto-1-Beispielimplementierung

```c
ret |= (uint32_t)mifare_update_byte(state, (injection >> (i*8)) & 0xff, feedback) << (i*8);
return ret;
}

/* API section =============================== */
/* Initialize the LFSR with the key */
static void crypto1_clean_init(struct _crypto1_state * const state, const uint64_t key)
{
    state->lfsr = 0;
    state->prng = 0;

    /* Shift in keybytes in reverse order */
    uint64_t k = key;
    for(int i=0; i<6; i++) {
        state->lfsr <<= 8;
        state->lfsr |= (k & 0xff);
        k >>= 8;
    }
}

/* Shift UID xor card_nonce into the LFSR without active cipher stream */
static void crypto1_clean_mutual_1(struct _crypto1_state * const state, const uint32_t uid, const uint32_t card_challenge)
{
    const uint32_t IV = uid ^ card_challenge;

    /* Go through the IV bytes in MSByte first, LSBit first order */
    mifare_update_word(state, IV, 0);
    state->prng = card_challenge; /* Load the card’s PRNG state into our PRNG */
    return;
}

/* Shift in the reader nonce to generate the reader challenge, */
/* then generate the reader response */
static void crypto1_clean_mutual_2_reader(struct _crypto1_state * const state, parity_data_t * const reader_response)
{
    /* Unencrypted reader nonce */
    const uint32_t reader_nonce = ARRAY_TO_UINT32(reader_response);

    /* Feed the reader nonce into the state and simultaneously encrypt it */
    for(int i = 3; i >= 0; i--) { /* Same as in mifare_update_word, but with added parity */
        reader_response[3-i] = reader_response[3-i] ^ mifare_update_byte(state, (reader_nonce >> (i*8)) & 0xff, 0);
        reader_response[3-i] ^= mf20(state->lfsr) << 8;
    }
}

/* Unencrypted reader response */
```
const uint32_t RR = prng_next(state, 64);
UINT32_TO_ARRAY_WITH_PARITY(RR, reader_response+4);

/* Encrypt the reader response */
crypto1_transcrypt(state, reader_response+4, 4);
}

/* Generate the expected card response and compare it to the actual */
static int crypto1_clean_mutual_3_reader(struct _crypto1_state * const state,
const parity_data_t * const card_response)
{
const uint32_t TR_is = ARRAY_TO_UINT32(card_response);
const uint32_t TR_should = prng_next(state, 96) ^ mifare_update_word(state, 0, 0);
return TR_is == TR_should;
}

/* Shift in the reader challenge into the state, generate expected reader * response and compare it to actual reader response. */
static int crypto1_clean_mutual_2_card(struct _crypto1_state * const state,
const parity_data_t * const reader_response)
{
/* Reader challenge/Encrypted reader nonce */
const uint32_t RC = ARRAY_TO_UINT32(reader_response);
/* Encrypted reader response */
const uint32_t RR_is = ARRAY_TO_UINT32(reader_response+4);

/* Shift in reader challenge */
const uint32_t keystream = mifare_update_word(state, RC, 1);

#ifdef PRINT_CORRECT_READER_CHALLENGE
printf("%016LX
", RC ^ keystream);
#else
(void)keystream;
#endif

/* Generate expected reader response */
const uint32_t RR_should = prng_next(state, 96) ^ mifare_update_word(state, 0, 0);
return RR_should == RR_is;
}

/* Output the card response */
static void crypto1_clean_mutual_3_card(struct _crypto1_state * const state,
parity_data_t * const card_response)
{
/* Unencrypted tag response */
const uint32_t TR = prng_next(state, 96);
UINT32_TO_ARRAY_WITH_PARITY(TR, card_response);

/* Encrypt the response */
crypto1_transcrypt(state, card_response, 4);
}
/* Encrypt or decrypt a number of bytes */
static void crypto1_clean_transcrypt_bits(struct _crypto1_state *const state,
    parity_data_t *const data, const size_t bytes, const size_t bits)
{
    for(int i = 0; i < bytes; i++) {
        data[i] = data[i] ^ mifare_update_byte(state, 0, 0);
        data[i] = data[i] ^ (mf20(state->lfsr) << 8);
    }
    for(int i = 0; i < bits; i++) {
        data[bytes] ^= mifare_update(state, 0, 0) << i;
    }
}

static const struct _crypto1_ops crypto1_clean_ops = {
    crypto1_clean_init,
    crypto1_clean_mutual_1,
    crypto1_clean_mutual_2_reader,
    crypto1_clean_mutual_3_reader,
    crypto1_clean_mutual_2_card,
    crypto1_clean_mutual_3_card,
    crypto1_clean_transcrypt_bits
};

int _crypto1_new_clean(crypto1_state *const state)
{
    state->ops = &crypto1_clean_ops;
    return 1;
}
Listing A.4: tests.c – Testcases und Nutzungsbeispiel

/*
* Philips/NXP Mifare Crypto-1 implementation v1.0
* By Karsten Nohl, Henryk Plötz, Sean O’Neil
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "crypto1.h"

#define ARRAY_LENGTH(x) (sizeof(x)/sizeof(x[0]))
#ifndef DEBUG_PRINTS
#define MAX_PLAINTEXT_CIPHERTEXT_PAIRS 5
#define MAX_PLAINTEXT_LEN 18
#else
*/ format is either "xx xx xx xx..."
* or "xx b? xx b? xx b?...
* where xx is the byte in hex, b is the parity bit and ? may be anything or
* not there at all
*/
typedef char* printable_parity_data_t;
#endif
#define FLAGS_NO_CHECK_PARITY_AUTH 1

struct plaintext_ciphertext_pair {
    printable_parity_data_t plaintext;
    printable_parity_data_t ciphertext;
    size_t num_bits;
};

struct testcase {
    uint64_t key;
    uint32_t uid;
    uint32_t card_challenge;
    uint32_t reader_challenge;
    printable_parity_data_t reader_response;
    printable_parity_data_t card_response;
    int data_pairs;
    struct plaintext_ciphertext_pair data[MAX_PLAINTEXT_CIPHERTEXT_PAIRS];
};

#define KEY_1 0xfffffffffffffffULL
#define KEY_2 0xa0a1a2a3a4a5ULL
#define UID_1 0xB479F7D7
#define UID_2 0x8CBA5DD3

/* These test cases come from sniffed transactions with known key and
* contents between an OpenPCD and a Mifare Classic 1k card */
const struct testcase testcases[] = {

A. Crypto-1-Beispielimplementierung

```c
/* key, uid, card chal, reader chal,
    reader resp, card resp, number of test ciphertexts,
    plaintext/ciphertext pairs ... */

{KEY_1, UID_1, 0xF3FBAEED, 0x07C9A995,
  "7C 1! 74 1 07 1! EB 1 0F 0! 7B 1 DS 0 1B 0!",
  "3D 1! 0E 1! A0 0! E2 1!", 2, {
  {"30 1 00 1 02 0 a8 0", "65 0! 8D 0! 65 1 1F 0"},
  {"B4 1 79 0 F7 0 D7 1!", "52 0 F6 1 46 0 35 1"}, }
},

{KEY_1, UID_1, 0x2D4DAAC5, 0x68368F0C,
  "ED 1 73 1! 6B 0 02 1! 88 1 42 1 5B 0 A4 1!",
  "A2 1! D4 0! 3C 0! C3 1!", 2, {
  {"30 1 00 1 02 0 a8 0", "5B 0 6F 1 96 1 CF 1"},
  {"B4 1 79 0 F7 0 D7 1!", "BB 1 FD 1! 82 1 D2 0!"}, }
},

{KEY_1, UID_2, 0x9347B9F4, 0x3BA73C6D,
  "E5 1! 0A 1 5B 0 84 1 44 1 E5 1! C1 0 0C 1!",
  "A7 0 A2 2! DA 0 ED 0!", 4, {
  {"A0 1 01 0 d6 0 a0 1", "E3 0 B6 0 0E 1! A5 1"},
  {"0a", "00", 4},
  {"00 1 00 1 00 1 00 1 00 1 00 1 00 1 00 1 00 1 00 1
   00 1 37 0 49 0", "C2 0 E1 1 E4 1 22 0! 99 1 78 0! 6B 0 A1 1! D2 1 C8 1!
   62 1! 14 1 0A 1 BA 0 DD 1 AE 0 00 0! 59 0!*},
  {"0a", "0C", 4}, }
},

{KEY_2, UID_2, 0x0DF547C9, 0x55414992,
  "85 0 1E 1 29 1! 49 0 BF 0 44 1 5B 1! EB 1!",
  "A5 0! 86 1! F4 0 37 1!", 2, {
  {"30 1 04 0 26 0 ee 1", "86 1! E0 1! 1B 0! 9E 0"},
  {"00 1 00 1 00 1 00 1 00 1 00 1 00 1 00 1 00 1 00 1
   00 1 37 0 49 0", "A3 1 5B 1! F2 0 F9 1 00 0! A9 0! 5F 0! A5 1 1C 0 95
   0! E7 0! OD 0 19 0 25 1! F6 0! E1 1 51 0 79 0!*}, }
},

static int get_hex_value(char c)
{
  return ( (c>='a'&&c<='f') ? (10+c-'a') : ( (c>='A'&&c<='F') ? (10+c-'A') : (c-'0') ) ) & 0xf;
}

static int convert_printable_parity_data(const printable_parity_data_t in, parity_data_t *out, size_t *outlen)
{
  size_t l = strlen(in);
  int outpos = 0;
  uint16_t byte = 0;
  enum { FIRST_NIBBLE, SECOND_NIBBLE, PARITY} next = FIRST_NIBBLE;
  for(int inpos = 0; inpos<l; inpos++) {
    const char c = in(inpos);
    if(outpos == *outlen) {
      for(;inpos<l; inpos++)
        if(isxdigit(in(inpos)))
```
return 0;
return 1;
}

switch(next) {
case FIRST_NIBBLE:
  if(isxdigit(c)) {
    byte = get_hex_value(c);
    next = SECOND_NIBBLE;
  }
  break;
case SECOND_NIBBLE:
  if(isxdigit(c)) {
    byte = (byte << 4) | get_hex_value(c);
    next = PARITY;
  } else {
    out[outpos++] = byte;
    next = FIRST_NIBBLE;
  }
  break;
case PARITY:
  if(isxdigit(c)) {
    if(!isxdigit(in[inpos+1])) {
      byte = byte | (get_hex_value(c) << 8);
      out[outpos++] = byte;
      next = FIRST_NIBBLE;
    } else {
      out[outpos++] = byte;
      byte = get_hex_value(c);
      next = SECOND_NIBBLE;
    }
  }
  break;
if(next != FIRST_NIBBLE) {
  if(outpos == *outlen)
    return 1;
  else
    out[outpos++] = byte;
}
*outlen = outpos;
return 1;
}

static int compare_parity_data(parity_data_t *a, parity_data_t *b, size_t len,
    size_t num_bits, unsigned int *warnings)
{
  char warned = 0;
  for(int i=0; i<len; i++) {
    if(i==len-1 && num_bits > 0) {
      parity_data_t c = a[i], d = b[i];
      c &= !(~0)<<num_bits);
    }
d &= ~((~0)<<num_bits);
if(c != d)
    return 0;
} else {
    if((a[i] & 0xff) != (b[i] & 0xff)) {
        return 0;
    } else if(a[i] != b[i]) {
        if(!warned) {
            printf(" + Warning: Parity mismatch at offset %i\n", i);
            if(warnings)
                (*warnings)++;
        }
        warned = 1;
    }
}
return 1;

static unsigned int do_test(crypto1_state *state, const struct testcase *
testcase, unsigned int *warnings, const int flags)
{
    parity_data_t reader_response[8], reader_response_backup[8];
    parity_data_t card_response[4], card_response_backup[8];
    parity_data_t plaintext[MAX_PLAINTEXT_LEN], ciphertext[MAX_PLAINTEXT_LEN];
    size_t l = ARRAY_LENGTH(reader_response);
    if(!convert_printable_parity_data(testcase->reader_response,
                   (parity_data_t *)&reader_response, &l)) {
        printf(" > Internal error 1\n");
        return 1;
    }
    l = ARRAY_LENGTH(card_response);
    if(!convert_printable_parity_data(testcase->card_response,
                   (parity_data_t *)&card_response, &l)) {
        printf(" > Internal error 2\n");
        return 1;
    }
    memcpy(reader_response_backup, reader_response, sizeof(reader_response));
    memcpy(card_response_backup, card_response, sizeof(card_response));
    cryptol_init(state, testcase->key);
    crypto1_mutual_1(state, testcase->uid, testcase->card_challenge);
    if(!state->is_card) {
        // Preload the reader nonce into the reader response */
        UINT32_TO_ARRAY_WITH_PARITY(testcase->reader_challenge, reader_response);
    }
    if(!crypto1_mutual_2(state, reader_response)) {
        printf(" > Mutual 2 failed\n");
        return 1;
    }
if(!crypto1_mutual_3(state, card_response)) {
    printf("> Mutual 3 failed\n");
    return 1;
}

if(!(flags & FLAGS_NO_CHECK_PARITY_AUTH)) {
    if(!compare_parity_data(reader_response, reader_response_backup, ARRAY_LENGTH(reader_response), 0, warnings)) {
        printf("> Compare reader response failed\n");
        return 1;
    }
    if(!compare_parity_data(card_response, card_response_backup, ARRAY_LENGTH(card_response), 0, warnings)) {
        printf("> Compare card response failed\n");
        return 1;
    }
    for(int i=0; i<testcase->data_pairs; i++) {
        size_t pl = ARRAY_LENGTH(plaintext), cl = ARRAY_LENGTH(ciphertext);
        if(!convert_printable_parity_data(testcase->data[i].plaintext, (parity_data_t*)&plaintext, &pl)) {
            printf("> Internal error 3\n");
            return 1;
        }
        if(!convert_printable_parity_data(testcase->data[i].ciphertext, (parity_data_t*)&ciphertext, &cl)) {
            printf("> Internal error 4\n");
            return 1;
        }
        if(testcase->data[i].num_bits == 0)
            crypto1_transcrypt(state, ciphertext, cl);
        else
            crypto1_transcrypt_bits(state, ciphertext, cl-1, testcase->data[i].num_bits);
    }
    #ifdef DEBUG_PRINTS
    for(int j=0; j<cl; j++) {
        printf("%02X", ciphertext[j] & 0xff);
    }
    printf("\n");
    #endif
    if(!compare_parity_data(ciphertext, plaintext, pl, testcase->data[i].num_bits, warnings)) {
        printf("> Compare plaintext %i failed: Is ", i);
        for(int j=0; j<cl; j++) {
            printf("%02X", ciphertext[j] & 0xff);
        }
        printf(" should be ");
        for(int j=0; j<cl; j++) {
            printf("%02X", plaintext[j] & 0xff);
        }
        printf("\n");
    }
}
return 1;
}

return 0;
}

static unsigned int do_tests(crypto1_state *state, const char *mode, const int flags)
{
    unsigned int result = 0;
    unsigned int warnings = 0;
    for(int i = 0; i< ARRAY_LENGTH(testcases); i++) {
        int ret = do_test(state, &testcases[i], &warnings, flags);
        if(ret > 0) {
            printf("+ Testcase %i failed in %s mode\n", i, mode);
        } else {
            if(warnings > 0)
                printf("+ Testcase %i produced %i warnings in %s mode\n", i, warnings, mode);
        }
        result += ret;
        warnings = 0;
    }
    return result;
}

static unsigned int test_one_implementation(enum crypto1_cipher_implementation implementation, const int flags)
{
    crypto1_state state;
    int result = 0;
    if(crypto1_new(&state, 0, implementation)) {
        result += do_tests(&state, "reader", flags);
    } else {
        printf("+ failed to initialize implementation in reader mode\n");
        result += 1;
    }
    if(crypto1_new(&state, 1, implementation)) {
        result += do_tests(&state, "card", flags);
    } else {
        printf("+ failed to initialize implementation in card mode\n");
        result += 1;
    }
    return result;
}

int main(void)
{
    unsigned int errors = 0;
    printf("Testing clean implementation\n");
}
errors += test_one_implementation(CRYPTO1_IMPLEMENTATION_CLEAN, 0);

/* The optimized implementation doesn't correctly handle parity bits during authentication, simply silence those warnings (and don't use this implementation for any real hardware) */
printf("Testing optimized implementation\n");
errors += test_one_implementation(CRYPTO1_IMPLEMENTATION_OPTIMIZED,
    FLAGS_NO_CHECK_PARITY_AUTH);

printf("------\n");
if(errors > 0) {
    printf("%u errors during test run\n", errors);
    return 1;
} else {
    printf("0 errors during test run\n");
    return 0;
}
Abbildungsverzeichnis

2.1. Visualisierung der Modulation und Kodierung in PCD→PICC- und PICC→PCD-Richtung, nach ISO 14443-2 Abbildung 1 ........................ 8
2.2. FDT zwischen Ende der PCD-Kommunikation und Beginn der PICC- Kommunikation, nach ISO 14443-3 Abbildung 1 ......................... 10
2.3. Zustandsmaschine für Tags nach ISO 14443-3 Typ A, vereinfacht nach ISO 14443-3 Abbildung 6 ....................................................... 11
2.4. ANTICOLLISION- und SELECT-Kommandos .......................... 12
2.5. Mifare-Classic-Speicherlayout ............................................. 20
2.6. Format der Zugriffsbedingungen im Mifare Classic Sektor-Trailer .... 20
2.7. Kommunikation zwischen Host, Leser-IC und Karte bei der Authentisierung ................................................................. 24
2.8. Wertblock-Format auf Mifare-Classic-Karten .......................... 25
3.1. OpenPCD-Platine ................................................................. 28
3.2. Vereinfachter Übersichtsschaltplan für OpenPCD ....................... 29
3.3. OpenPICC-Platine v0.2 ......................................................... 31
3.4. Vereinfachter Übersichtsschaltplan für OpenPICC v0.2 ohne Modifikationen ................................................................. 31
3.5. Genereller Empfangs- und Sendeablauf, am Beispiel von REQA und ATQA nach ISO 14443 Typ A ........................................ 34
3.6. OpenPICC-Platine v0.2 mit Modifikationen ............................. 35
4.1. Versuchsaufbau, um die Kommunikation des OpenPCD mit einer Karte bidirektional mitzuschneiden ................................. 42
5.1. Mifare-Classic-Karte nach 30 Minuten in Aceton ......................... 48
5.2. Die 6 Ebenen des Mifare-Classic-1k-Chips ............................. 49
6.1. Übersicht über Crypto-1 ......................................................... 53
6.2. Detailansicht der Filterfunktion \( f(\cdot) \) von Crypto-1 .......... 53
7.1. Einfacher Relay-Aufbau nach Hancke ................................. 58
Tabellenverzeichnis

2.1. Kodierungen für beide Kommunikationsrichtungen, jeweils bezogen auf eine Bitlänge ................................................................. 9
2.2. Übersicht über die verschiedenen Kartentypen der Mifare-Familie .................. 19
2.3. Bedeutung der Zugriffsbedingungen für den Sektor-Trailer .................... 22
2.4. Bedeutung der Zugriffsbedingungen für normale Blöcke ....................... 23

4.1. Erste bidirektionale Sniffing-Ergebnisse einer Mifare-Classic-Sitzung, in der ein Sektor gelesen wird ........................................ 43
4.2. Erste vier aufgezeichnete Challenge-Response-Paare ......................... 44
4.3. Verlauf des Experiments, die zu den Schlüsselbits zugehörigen UID-Bits zu finden ................................................................. 46

6.1. Karten-Nonces, in drei Sitzungen gesnifft, mit Offset relativ zu einem willkürlich gewählten Startpunkt ........................... 52
7.1. Häufige Kommandosequenzen und die zu beobachtenden Längen der übertragenen Daten ......................................................... 62
Verzeichnis der Listings

2.1. Implementierung von CRC_A .................................................. 12

6.1. Implementierung eines PRNG, äquivalent zum in Mifare Classic benutzten
     PRNG .................................................................................. 52

A.1. crypto1.h – API und Globale Definitionen ................................. 71
A.2. crypto1.c – API ......................................................................... 74
A.3. crypto1_clean.c – Unoptimierte, direkte C-Implementierung .......... 76
A.4. tests.c – Testcases und Nutzungsbeispiel .................................. 81
Verzeichnis der Algorithmen

2.1. Bitorientiertes Antikollisionsverfahren nach ISO 14443-3 Typ A . . . . . . . 15
4.1. Algorithmus, um die zu den Schlüsselbits zugehörigen UID-Bits zu finden . 45
Literaturverzeichnis


[BGS+05] BONO, Steve ; GREEN, Matthew ; STUBBLEFIELD, Adam ; JUELS, Ari ; RUBIN, Avi ; SZYDLO, Michael: Security analysis of a cryptographically-enabled RFID device. In: 14th USENIX Security Symposium, USENIX, 2005


Literaturverzeichnis


\%20Geheime\_code\_van\_ov-\kaart\_ligt\_op\_straat. Zeitungsartikel


[KNPo8] KRISSSLER, Jan ; NOHL, Karsten ; PLOTZ, Henryk: Chiptease. In: c't magazin für computer technik (2008), Nr. 8, S. 80–85


Literaturverzeichnis


Alle URLs wurden zuletzt am 18.08.2008 geprüft.