Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät II
Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin

Mifare Classic – Eine Analyse der Implementierung

Henryk Plötz

Betreuer: Prof. Dr. Ernst-G. Giessmann

Diese Arbeit enthält die Ergebnisse und wesentlichen Bestandteile der Diplomarbeit des Autors (eingereicht am 18. August 2008). Sie wurde jedoch an einigen Stellen leicht erweitert, korrigiert und durch Anhang A ergänzt.

Letzte inhaltliche Änderung: 14. Oktober 2008

Inhaltsverzeichnis

Vorwort v

I. Einführung 1

1. Einleitung 3

2. RFID 5
- 2.1. Grundsätzliche Funktionsweise . . . 5
- 2.2. Frequenzbereiche . . . 6
  - 2.2.1. 13.56MHz . . . 6
- 2.3. ISO 14443 Typ A im Detail . . . 7
  - 2.3.1. Part 1 . . . 7
  - 2.3.2. Part 2 . . . 8
  - 2.3.3. Part 3 . . . 8
  - 2.3.4. Part 4 . . . 14
- 2.4. Mifare . . . 14
  - 2.4.1. Geschichte . . . 15
  - 2.4.2. Varianten . . . 16
  - 2.4.3. Mifare-Classic-Zugriffsbedingungen . . . 18
  - 2.4.4. Mifare-Classic-Kommandoablauf . . . 21

3. Verwendete Hardware 27
- 3.1. OpenPCD . . . 28
  - 3.1.1. RC632 . . . 29
  - 3.1.2. MFIN/MFOUT . . . 30
- 3.2. OpenPICC . . . 30
  - 3.2.1. Analogteil . . . 32
  - 3.2.2. Digitalteil . . . 32
  - 3.2.3. Empfangs- und Sendeablauf . . . 34
  - 3.2.4. Probleme und Modifikationen . . . 35

II. Analyse 37

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 fotographieren . . . 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.

Nach der Vorstellung der ersten Zwischenergebnisse gab es große Resonanz, sowohl in der Mainstream-Presse als auch bei den Herstellern und Benutzern der betroffenen Kartensysteme. Die Aufdeckung der Schwächen des Systems hat direkt dazu geführt, dass geplante und existierende Systeme überdacht oder erweitert wurden und indirekt die Ankündigung und Details eines neuen Kartentyps beschleunigt und verändert.

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 löschbarer, 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

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

Kapitel 1

> A good traveler has no fixed plans, and is not intent on arriving.
>
> LAO TZU

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 Verworrenheit' 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

Kapitel 2 erklärt die zugrundeliegende Technologie und die verwendeten Kommunikationsprotokolle. Kapitel 3 wird besonders hilfreiche Hardware, deren Aufbau und Funktion beschreiben. Kapitel 4 befasst sich mit meinen Experimenten mit der Hardware und dem Funkprotokoll. Kapitel 5 gibt eine Übersicht über die Reverse-Engineering-Technik die von Jan Krissler und Karsten Nohl verwendet wurde. Kapitel 6 fasst die Ergebnisse der vorhergehenden Kapitel zusammen und beschreibt die Funktionsweise der untersuchten Karten. Kapitel 7 und 8 leiten dann daraus verschiedene Angriffe gegen die Sicherheit des Systems ab.

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 ([Ver08])

Nach meiner Kenntnis wurden die Hardware-Reverse-Engineering-Techniken aus Kapitel 5 bisher bei keiner veröffentlichten Analyse eines kryptographischen Systems eingesetzt. Öffentlich dokumentierte Analysen von proprietären Kryptosystemen beziehen sich in der Regel auf ein Reverse Engineering von Softwarekomponenten oder eine Black-Box-Analyse. Darauf wird in Kapitel 5 auf Seite 47 kurz eingegangen werden.

> 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.

Auch in dieser engeren Definition von *RFID* gibt es zwei Enden eines Spektrums mit teils fließenden Übergängen: Von sehr ‚dummen' Tags, die nur eine Identifikation im Wert von wenigen Bits übermitteln können bis hin zu voll ausgewachsenen Mini-Computern mit erweiterten kryptographischen Funktionen. Manche Definitionen des RFID-Begriffs beschränken sich ausdrücklich auf den ersteren Teil dieses Spektrums und unterscheiden RFID-Tags explizit von „kontaktlosen Smartcards" (evt. noch mit einer Kategorie „kontaktlose Speicherkarten" dazwischen). Da sich diese Arbeit hauptsächlich mit eben diesen kontaktlosen Karten beschäftigt, soll hier ‚RFID' so definiert sein, dass es mindestens kontaktlose Speicherkarten und Smartcards umfasst.

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 eingegrenzte 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.

2.2. 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*).

ISO 14443 ist ein Lastmodulationssystem, bei dem das Lesegerät ein relativ starkes elektromagnetisches Feld mit einer Frequenz von 13,56 MHz (± 7 kHz) erzeugt, an welches die Karte induktiv angekoppelt wird. Durch die Kopplung wird in der Antenne der Karte eine Wechselspannung induziert, welche dann mit einem Gleichrichter und einem Kondensator als Kurzzeitenergiespeicher die Spannungsversorgung für die Elektronik der Karte bereitstellen kann. Kommandos in PCD→PICC-Richtung werden auf dieses vom Lesegerät erzeugte Feld aufmoduliert, Antworten in PICC→PCD-Richtung werden von der Karte durch Lastmodulation gesendet, d.h. durch daten- und taktgesteuertes Ein- und Ausschalten einer Last (etwa eines Widerstands) an der Empfangsantenne der Karte, wodurch sich ebenfalls eine Modulation des Leserfeldes ergibt.

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 ggbf. 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 empfängen (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 (abgeschwächtes) 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

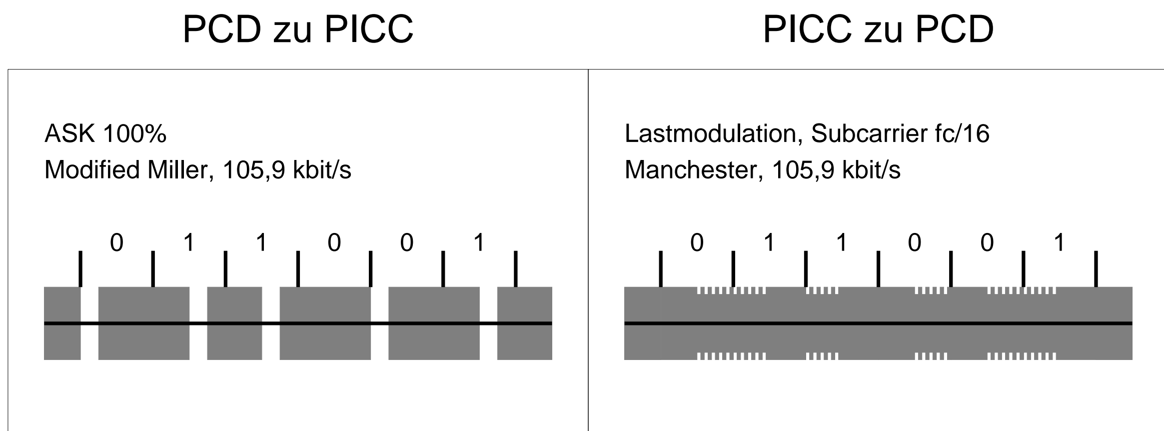
Mifare-Classic-Karten sind kompatibel zu ISO 14443 Typ A Part 1 bis 3. Mifare DESfire, SmartMX und ProX sind kompatibel zu ISO 14443 Typ A Part 1 bis 4 (siehe 2.4.2 ab Seite 16).

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)

Abbildung 2.1. Visualisierung der Modulation und Kodierung in PCD→PICC- und PICC→PCD-Richtung, nach ISO 14443-2 Abbildung 1



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_{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:

$$\begin{aligned} f_c &= 13,56\,\text{MHz} \pm 7\,\text{kHz} \\ t_{\text{Bit}} &= \frac{128}{f_c} \approx 9,44\,\mu\text{s} \\ f_s &= \frac{f_c}{16} \approx 847,5\,\text{kHz} \end{aligned}$$

Abbildung 2.1 zeigt Beispiele für die Kommunikation von Lesegerät zur Karte und von Karte zum Lesegerät, Tabelle 2.1 auf der nächsten Seite fasst die Kommunikationskodierung für SOF (*Start of Frame*, Beginn des Datenrahmens), 0-Bit, 1-Bit und EOF (*End of Frame*, Ende des Datenrahmens) zusammen.

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: $\text{FDT} = (n * 128 + 84)/f_c$,

Tabelle 2.1. Kodierungen für beide Kommunikationsrichtungen, jeweils bezogen auf eine Bitlänge

Symbol	Kodierung in ...-Richtung PCD→PICC	PICC→PCD
SOF	Mod. am Anfang	Mod. während der ersten Hälfte
0	nach 1-Bit: keine Modulation sonst: Mod. am Anfang	Mod. während der zweiten Hälfte
1	Modulation nach der Hälfte	Mod. während der ersten Hälfte
EOF	ein 0-Bit, dann eine Bitdauer lang keine Mod.	keine Modulation

wenn das letzte Datenbit des PCD-Frames ein 1-Bit ist, und FDT $= (n * 128 + 20)/f_c$, 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 $(n * 128)$ 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:

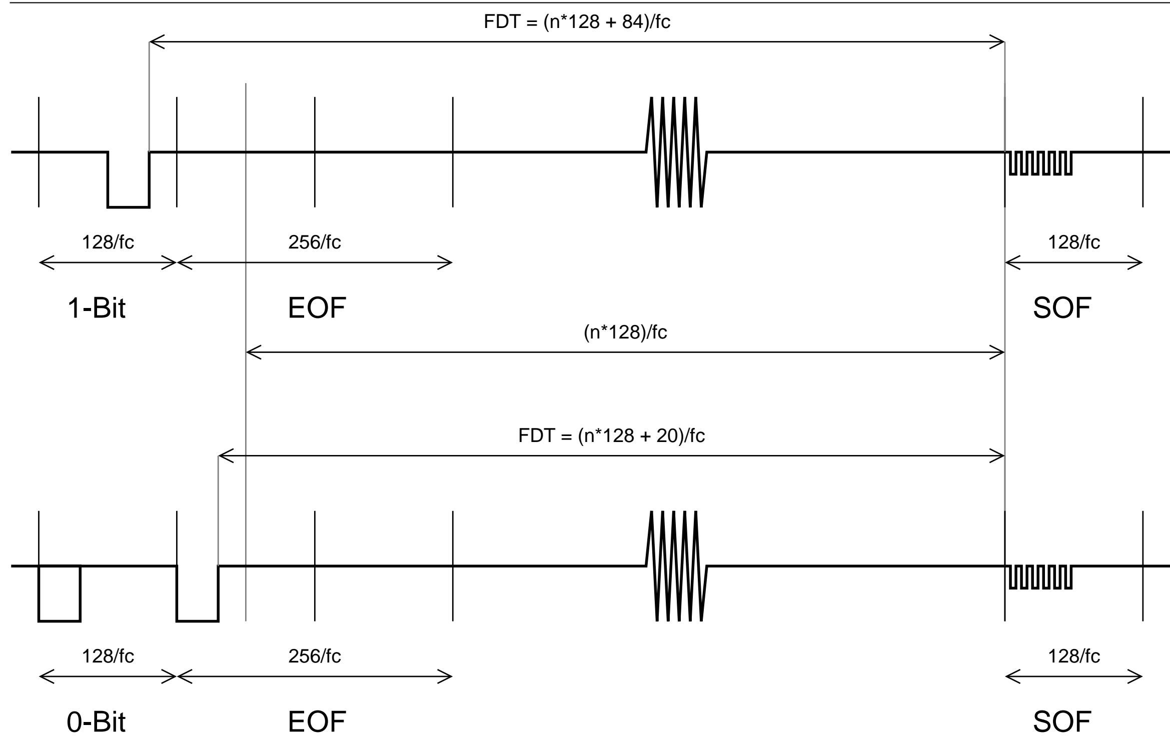
Short Frame wird für die Initialisierung benutzt. Es besteht aus SOF, 7 Bit Daten und EOF. Das MSB (*Most Significant Bit*, Höchstwertiges Bit) wird zuerst übertragen. Es werden keine Paritätsbits oder CRC hinzugefügt.

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 `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-Frames[2].

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)

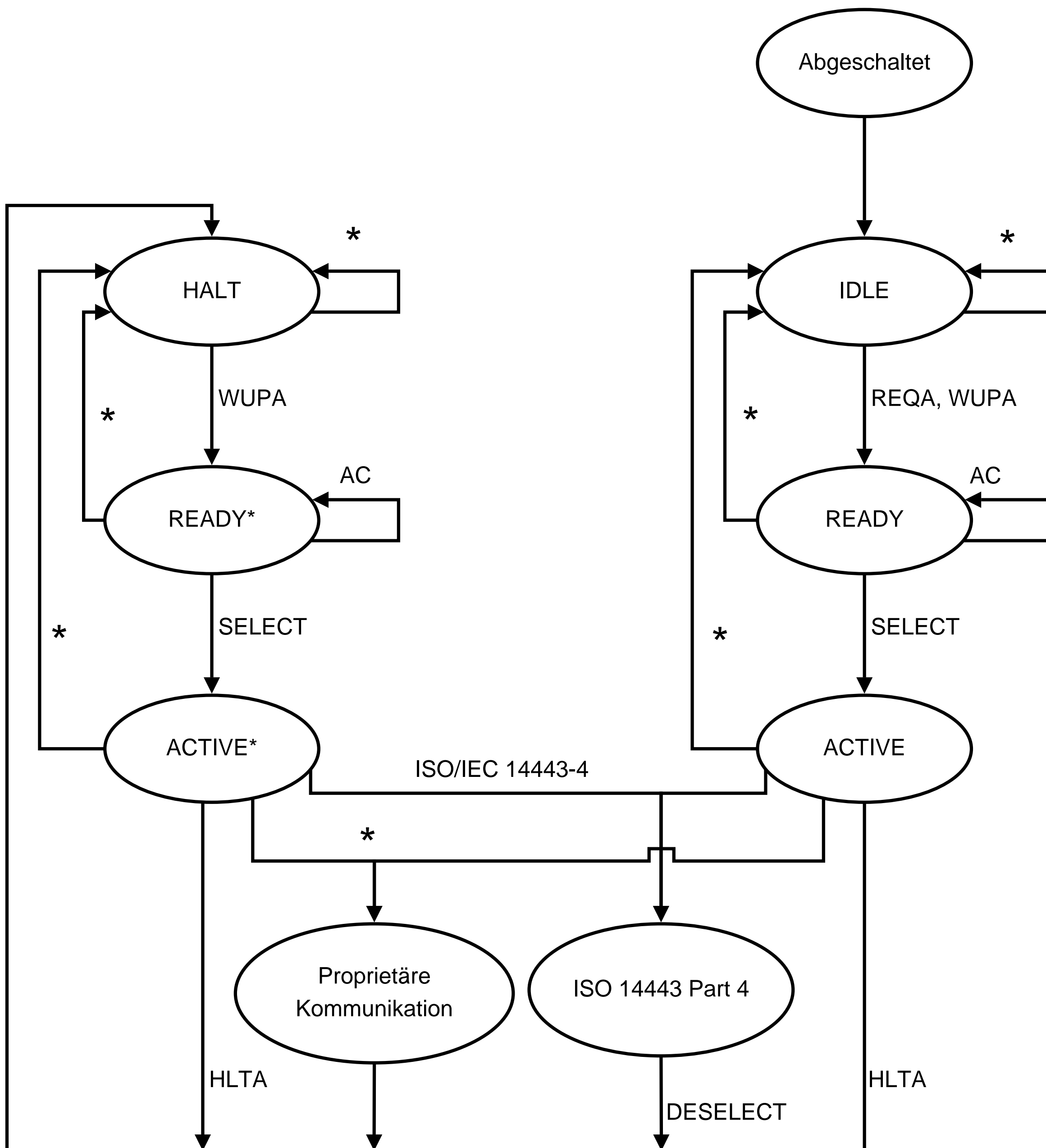
HLTA, DESELECT HTLA (*Halt, Type A*, Abschalten, Typ A) bzw. Deselektionskommando bei der PICC empfangen

⋆ 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)

Die Kommandos REQA und WUPA werden in *short frames* übertragen. Die Antikollisionskommandos werden in bitorientierten Antikollisionsframes übertragen. Alle anderen Kommandos werden in Standardframes übertragen.

[2]Während 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



Listing 2.1 Implementierung von `CRC_A`

```
#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>
<tr><td></td><td colspan="4">Byte</td></tr>
<tr><td></td><td>1</td><td>2</td><td>3 4 5 6 7</td><td>8 9</td></tr>
<tr><td>ANTICOLLISION</td><td>SEL</td><td>NVB</td><td>(0-39 Bit UID || BCC)</td><td></td></tr>
<tr><td>SELECT</td><td>SEL</td><td>NVB</td><td>40 Bit UID || BCC</td><td>CRC_A</td></tr>
</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

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.

Als letztes Kommando definiert ISO 14443-3 noch HLTA: `50 00`+CRC. Dieses Kommando hält eine selektierte Karte an und überführt sie in den HALT-Zustand. In Verbindung mit REQA (statt WUPA) für die Antikollision kann es verwendet werden, um Karten soweit zu deaktivieren, dass sie nicht mehr am Antikollisionsverfahren teilnehmen. Außerdem findet es Anwendung im proprietären Mifare-Classic-Protokoll, um Karten zu deaktivieren, nachdem sie selektiert wurden, damit mehrere Mifare-Classic-Karten im Feld nacheinander behandelt werden können.

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, mehrerer 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 einprogrammiert 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.

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 Kaskadierungsebenen 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 0 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 (implementierungsabhängig). 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[4]. 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

Part 4 von ISO 14443 beschreibt ein blockorientes Datenübertragungsprotokoll auf der Grundlage von ISO 14443-3, welches den Austausch von APDUs[5] nach ISO 7816-4 ([ISOd]) gestattet. Das Blockformat ist angelehnt an ISO 7816-3 T=1. Es werden außerdem die Kommandos RATS (*Request for Answer to Select*, Anfrage nach Selektionsantwort), welches mit einer ATS (*Answer To Select*, Selektionsantwort) beantwortet wird und PPS (*Protocol and Parameter Selection*, Protokoll- und Parameterauswahl) definiert. Das ATS-Frame nimmt dabei die Rolle des ATR in ISO 7816-3 ein und enthält auch einen zu diesem kompatiblen Block mit historischen Bytes.

2.4. Mifare

Mifare (M*ikron* 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 hoben sich durch zwei Besonderheiten hervor: eine hohe Datenübertragungsrate von ≈106 kbit/s

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

[5] APDU (*Application protocol data unit*, Dateneinheit auf Applikationsprotokollebene)

Algorithmus 2.1 Bitorientiertes Antikollisionsverfahren nach ISO 14443-3 Typ A

```
function DoOneCascadeLevel(SEL)
    n ← 0                                    // Die Anzahl der gültigen UID-Bits (als Integer)
    UID ← ∅                                  // UID als Bitstring
    repeat
        Sende ANTICOLLISION(SEL, n, UID)
        r ← empfangene UID-Bits
        if Kollision detektiert then
            c ← Bitposition der ersten Kollision
            UID ← UID || r_0 || ... || r_{c−1} || 0          // Alternativ || 1
        end if
        n ← Länge(UID)
    until n = 40
    Sende SELECT(SEL, n, UID)
    SAK ← Empfangenes SAK
    return UID, SAK
end function
function SelectFirst
    SEL ← 93h
    UID ← ∅
    repeat
        u, SAK ← DoOneCascadeLevel(SEL)
        if SAK zeigt an, dass die UID nicht vollständig ist then
            UID ← UID || u_8 ... u_39                        // u_0 ... u_7 = 88h
            SEL ← SEL + 2
        else
            UID ← UID || u_0 ... 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

1995 wurde die Mikron GmbH – und mit ihr die Mifare-Technologie – von Phillips Semiconductors aufgekauft. Phillips Semiconductors wiederum wurde 2006 von Phillips als eigenständige Firma abgetrennt und existiert seitdem als NXP Semiconductors. Im Laufe dieser Zeit, beginnend etwa 1997, wurde die Mifare-Technologie weiterentwickelt und zu einer ganzen Produktreihe ausgebaut, wobei die meisten der späteren Produkte nur noch

die Marketingbezeichnung „Mifare“ tragen und technisch (und sicherheitstechnisch) nichts mehr mit den ursprünglichen Chips zu tun haben[6]. Die ursprüngliche Verschlüsselungstechnik findet sich in dieser Produktreihe unter der Bezeichnung „Mifare Classic“ bzw. als Emulation in einigen der Prozessorchips. Die Mifare-Technik wurde an einige andere Firmen lizensiert: Infineon (vormals Siemens, [scn94]), Atmel und Hitachi.

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

2.4.2. Varianten

Die Mifare-Produktreihe umfasst eine große Anzahl verschiedener Chips mit unterschiedlichen Verschlüsselungstechniken und Speichergrößen. Allen gemeinsam ist, dass sie mindestens ISO 14443-3 Typ A als Kommunikationsprotokoll implementieren.

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[7] 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)

[6] Bis auf das Funkinterface natürlich.

[7] Dieser 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 herstellerspezisches 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 ProX und **Mifare SmartMX** sind frei programmierbare Mikroprozessorkarten, die nicht an eine starr vorgegebene Funktionalität gebunden sind – im Gegensatz zur gesamten Mifare-Classic-Reihe, die über eine fest verdrahtete Zustandsmaschine verfügt. Stattdessen müssen diese Karten mit einem Betriebssystem ausgestattet werden und funktionieren dann zum Beispiel als Java-Karte (wenn das Betriebssystem JCOP o.ä. ist). Außerdem gibt es Varianten mit kryptographischen Coprozessoren für (u.a.) RSA, 3DES oder AES, so dass die Karten dieser Reihen alle Funktionen haben können, die von konventionellen kontaktbehafteten Karten bekannt sind. Optional gibt es beide Kartentypen auch mit einem zusätzlichen kontaktbehafteten Interface als Dual-Interface-Karte. Karten dieses Typs (ohne Kontaktinterface) finden ihren Einsatz beispielsweise in elektronischen Reisepässen.

Mifare DESfire ist ebenfalls eine Mikroprozessorkarte, allerdings mit fest vorgegebenem Betriebssystem im ROM. Damit sind diese Karten in ihrer Flexibilität eingeschränkt, dafür aber kostengünstiger herzustellen und einfacher anzuwenden (da sich der Benutzer keine Gedanken mehr über die Wahl/Programmierung des Betriebssystems machen muss). Diese Karten sind damit als geringfügig teurere, aber sicherere und flexible Alternative zu Mifare Classic platziert. Namensgebendes Hauptmerkmal ist die Unterstützung von Triple DES (und damit auch Single DES, wenn man beide 3DES-Halbschlüssel gleich setzt) als Verschlüsselungsfunktion. DESfire-Karten unterstützen weiterhin ein ISO-7816-4-artiges Dateisystem, welches den Speicher (4 Kilobyte EEPROM) in verschiedene (bis zu 28) Applikationen mit einzelnen Dateien aufteilen kann.

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.

Mifare Plus ist ein doppelt belegter Name: Beginnend ca. 1996-1997 ([Fin06, S. 340], [Ber96]) gab es unter diesem Namen die Entwicklung eines Dual-Interface-Chips für die Kombination von kontaktloser Kommunikation nach dem Protokoll, das heute ISO 14443-A heisst, sowie ISO 7816. Dieses Produkt verfügte über 8 Kilobyte gemeinsames EEPROM und zwei fast völlig getrennte Interface-Einheiten, die beide auf diesen Speicher zugreifen konnten. Das war nötig, da mit der damaligen Technik noch keine vollständige Mikroprozessorkarte nur über das Funk-Interface mit Energie versorgt werden konnte. Stattdessen war im Funkbetrieb nur die wesentlich simplere Mifare-Classic-Zustandsmaschine aktiv.

Im März 2008 hat NXP einen neuen Chip angekündigt ([NXP08b]), der ebenfalls "Mifare Plus" heisst und einen Zwischenschritt zwischen Mifare Classic und Mifare DESfire EV1 darstellt. Dieses neu vorgestellte Mifare Plus soll (optional) zufällige UIDs, Crypto-1-Verschlüsselung (permanent abschaltbar) und 128 Bit AES-Verschlüsselung enthalten und die Migration von Mifare Classic auf Karten mit dem AES-Algorithmus erleichtern: Eine Installation kann solange mit Crypto-1 betrieben werden, bis alle Lesegeräte/Backend-Systeme AES-fähig sind und alle Karten gegen Mifare-Plus-Karten ausgetauscht wurden. Dann kann der gesamte Betrieb mit einem Schlag auf AES umgestellt und Crypto-1 dauerhaft abgeschaltet werden.

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 ACL_n-Feldern zu erkennen und bestimmte Arten von Angriffen[8] 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 ACL_n-Feld: $C1_i$, $C2_i$ und $C3_i$ (sowie zusätzlich die invertierten Bits $\overline{C1_i}$, $\overline{C2_i}$, $\overline{C3_i}$). 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 ACL_n-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.

[8] Ein 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]

Tabelle 2.2. Übersicht über die verschiedenen Kartentypen der Mifare-Familie

		EEPROM		Verschlüsselung				Kommunikation		
		Brutto	Netto	CRYPTO-1	3DES	AES	Sonstige	Funk[a]	Kont.[b]	Sonst.
Classic	Classic 1k	1024 B	752 B	√	–	–	–	√	–	–
Classic	Classic 4k	4096 B	3440 B	√	–	–	–	√	–	–
Classic	Mini	320 B	224 B	√	–	–	–	√	–	–
Classic	Light[c]	48 B	24 B	√	–	–	–	√	–	–
Classic	Plus (alt)[c]	8 kB		√	–	–	–	√	√	–
	Ultralight	64 B	48 B [d]	–	–	–	–	√	–	–
	ProX	4 kB, 8 kB oder 16 kB		(√)[e]	(√)[f]	–	RSA[f], EEC[f]	√, T=CL	(√)[f]	–
	SmartMX	bis 72 kB		(√)[e]	(√)[f]	(√)[f]	RSA[f], EEC[f]	√, T=CL	(√)[f]	USB[f]
	DESfire	4 kB		–	√	–	–	√, T=CL	–	–
	DESfire EV1	2 kB, 4 kB oder 8 kB		–	√	√	–	√, T=CL	–	–
	Plus (neu)			√	–	√	–	√	–	–

[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

Abbildung 2.5. Mifare-Classic-Speicherlayout

Sektor	Block	Byte innerhalb des Blocks: 0 … 4	5	6 … 9	10 … 15	
0	0	UID / BCC[a]	Herstellerspez. Daten			Manufacturer-Block
	1					
	2					
	3	Schlüssel A_0		ACL_0 /	Schlüssel B_0	Sektor-Trailer 0
1	0					
	1					
	2					
	3	Schlüssel A_1		ACL_1 /	Schlüssel B_1	Sektor-Trailer 1
2	0					
	1					
	2					
	3	Schlüssel A_2		ACL_2 /	Schlüssel B_2	Sektor-Trailer 2
		⋮				

[a] Prüfsumme, byteweises XOR der vier Bytes aus UID

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

Bit	7	6	5	4	3	2	1	0
Byte 6	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
Byte 7	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
Byte 8	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$

Da es den Grund für getrennte Schlüssel ad absurdum führen würde, wenn man mit Schlüssel A auf Schlüssel B lesend 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.

3. Den Schlüsseln A und B können in einer Schlüsselhierarchie unterschiedliche Rechte zugeordnet werden.

Die Kombination aus dem ersten und dem dritten Punkt ergibt bespielsweise 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 aus dem zweiten und dritten Punkt ermöglicht wiederaufladbare Wertkarten etwa zur Benutzung an Verkaufsautomaten oder Fahrkartenentwertern, wobei die Verkaufsautomaten/Entwerter nur im Besitz des Schlüssels A sein brauchen, welcher ausschließlich zum Dekrementieren des gespeicherten Werts berechtigt. Ein Angreifer, der den entsprechenden Automaten stiehlt, wäre dann nicht in der Lage mithilfe der Schlüssel aus dem Automaten Karten aufzuwerten.

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-spezifizisches 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.

Im Lesegerät ist dazu ein Chip von NXP enthalten – wie etwa der MF RC632 im OpenPCD, siehe Abschnitt 3.1.1 auf Seite 29 – welcher die proprietäre Crypto-1-Verschlüsselung und -Authentisierung implementiert. Das Lesegerät sendet – ggbf. veranlasst durch ein Hintergrundsystem – Kommandos an diesen Chip, welcher dann die Authentisierung abwickelt. Das *Challenge-Response*-Protokoll erfordert den Austausch von vier Nachrichten zwischen Lesegerät und Mifare-Classic-Karte, je zwei pro Richtung. Das Lesegerät sendet

Tabelle 2.3. Bedeutung der Zugriffsbedingungen für den Sektor-Trailer

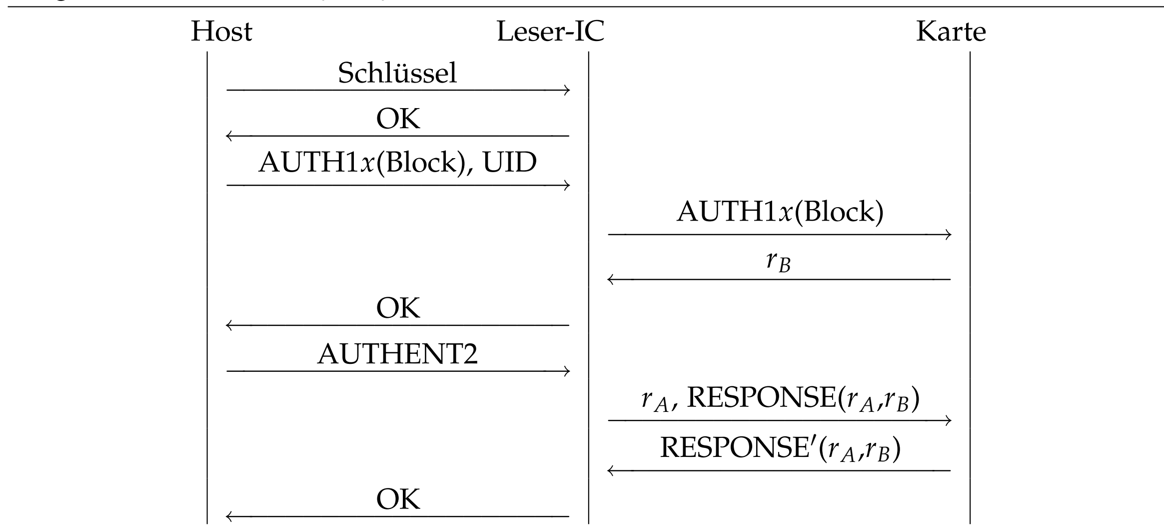
Zugriffsbits			Zugriff auf						
			Schlüssel A		Zugriffbits		Schlüssel B		
C1	C2	C3	lesen	schreiben	lesen	schreiben	lesen	schreiben	
0	0	0	verboten	Schl. A	Schl. A	verboten	Schl. A	Schl. A	Schlüssel B lesbar
0	0	1	verboten	Schl. A	Schl. A	Schl. A	Schl. A	Schl. A	Schl. B lesbar, Auslieferzustand
0	1	0	verboten	verboten	Schl. A	verboten	Schl. A	verboten	Schlüssel B lesbar
0	1	1	verboten	Schl. B	Schl. A oder B	Schl. B	verboten	Schl. B	
1	0	0	verboten	Schl. B	Schl. A oder B	verboten	verboten	Schl. B	
1	0	1	verboten	verboten	Schl. A oder B	Schl. B	verboten	verboten	
1	1	0	verboten	verboten	Schl. A oder B	verboten	verboten	verboten	
1	1	1	verboten	verboten	Schl. A oder B	verboten	verboten	verboten	

Tabelle 2.4. Bedeutung der Zugriffsbedingungen für normale Blöcke

Zugriffsbits			Zugriff auf				
C1	C2	C3	Lesen	Schreiben	Aufwerten	Abbuchen, Transfer, Restore	
0	0	0	Schl. A oder B[a]	Schl. A oder B[a]	Schl. A oder B[a]	Schl. A oder B[a]	Auslieferzustand
0	0	1	Schl. A oder B[a]	verboten	verboten	Schl. A oder B[a]	Wertblock
0	1	0	Schl. A oder B[a]	verboten	verboten	verboten	Datenblock
0	1	1	Schlüssel B[a]	Schlüssel B[a]	verboten	verboten	Datenblock
1	0	0	Schl. A oder B[a]	Schlüssel B[a]	verboten	verboten	Datenblock
1	0	1	Schlüssel B[a]	verboten	verboten	verboten	Datenblock
1	1	0	Schl. A oder B[a]	Schlüssel B[a]	Schlüssel B[a]	Schl. A oder B[a]	Wertblock
1	1	1	verboten	verboten	verboten	verboten	Datenblock

[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 $x \in \{A, B\}$



zuerst 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-1-Schaltkreisen, der als Sitzungsschlüssel 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[9], 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 Erfolgsfall antwortet die Karte mit einer zufälligen *nonce* und erwartet, dass sich das Lesegerät zuerst authentisiert[10].

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([ISOe]) 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 löschbarer, 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.

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

Byte-Nummer	0 …	4 …	8 …	12	13	14	15
Inhalt	Wert	$\overline{\text{Wert}}$	Wert	Addr	$\overline{\text{Addr}}$	Addr	$\overline{\text{Addr}}$

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

Für die Verwendung als elektronische Geldbörse oder in bestimmten Arten von Fahrgeldsystemen enthält Mifare Classic das besondere Konzept des Wertblocks. Auf diese Blöcke kann nicht nur mit dem normalen Lese- (und ggbf. Schreib-)Kommando zugegriffen werden, sondern auch mit 4 speziellen Wertblock-Kommandos: *Increment* (also Erhöhen, Aufwerten), *Decrement* (also Erniedrigen, Abbuchen), *Transfer* (Übertragen) und *Restore* (Wiederherstellen). Die letzten drei Kommandos sind in einer Gruppe und immer gleichzeitig erlaubt/verboten. Das Aufwertekommando kann selektiv für Schlüssel A und B oder nur für Schlüssel B erlaubt werden. Für jeden Schlüssel, für den das Aufwerte-Kommando erlaubt ist, müssen auch die *Transfer*- und *Restore*-Kommandos erlaubt sein.

Wertblöcke müssen ein bestimmtes Format einhalten, welches in Abbildung 2.8 gezeigt wird. Dieses Format enthält den eigentlichen Wert des Wertblocks als vorzeichenbehaftete 32-Bit-Ganzzahl in drei Kopien: Zweimal nicht invertiert und einmal invertiert. Außerdem enthält es ein zusätzliches Byte – welches von der Karte nicht ausgewertet wird und zum Beispiel verwendet werden kann, um die Nummer eines Reserve-Blocks anzugeben – in vier Kopien: Zweimal nicht invertiert und zweimal invertiert. Dieses Format kann nur durch ein Schreibkommando erzeugt werden und wird von der Karte bei allen Wertblock-Kommandos beibehalten. Die vier Wertblock-Kommandos sind:

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 Zwischenspeicherregister.

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.

Kapitel 3

In des tumben Toren Hand ist das beste Werkzeug Tand.

DANIEL DÜSENTRIEB

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]).

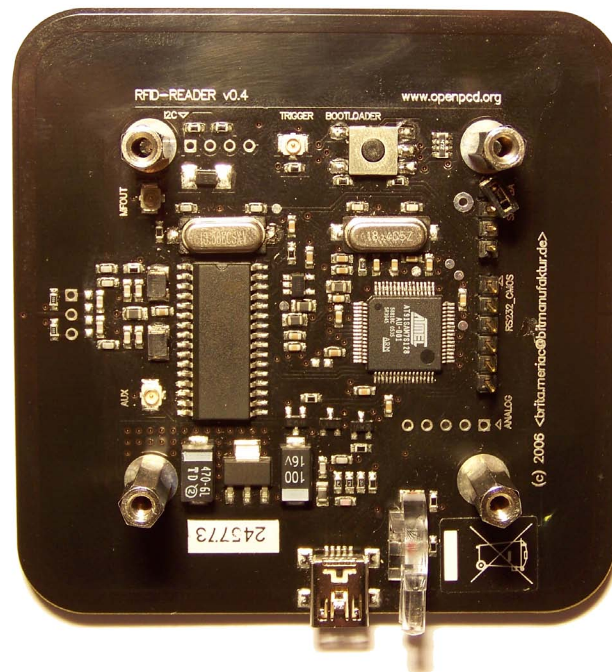
OpenPICC seinerseits kann beim Design eines RFID-Lesers helfen, da es ebenfalls direkten Zugang zu den empfangenen Signalen und die Möglichkeit zum Senden von Testsignalen liefert.

Beide Projekte benutzen einen AT91SAM7-Mikroprozessor von Atmel und benutzen USB zur Stromversorgung und Kommunikation. Weiterhin gibt es einen UART (*Universal Asynchronous Receiver Transmitter*, Universeller asynchroner Sender/Empfänger) mit TTL-Level, welcher für Debugausgaben benutzt werden kann. Die Mikroprozessoren verfügen über integrierten Flash-Speicher für Code und Daten sowie integriertem RAM und sind leistungsfähig genug, um eine Vielzahl von Aufgaben autonom in der Firmware ohne Interaktion mit einem PC auszuführen. So gibt es beispielsweise eine OpenPCD-Firmware-Variante, welche kontinuierlich nach aufgelegten Mifare-Classic-Karten sucht, bei vorhandener Karte selbsttätig mit einem Standardschlüssel authentisiert und dann alle Blöcke des ersten Sektors ausliest und auf dem UART ausgibt. (Dazu wurde librfid[Wel] von Harald Welte verwendet und in die Firmware integriert.)

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)

Abbildung 3.1. OpenPCD-Platine



auf Basis von FreeRTOS ([Bar]) geschrieben, welche nur noch wenige Hardwarefunktionen von der alten Firmware übernimmt.

3.1. OpenPCD

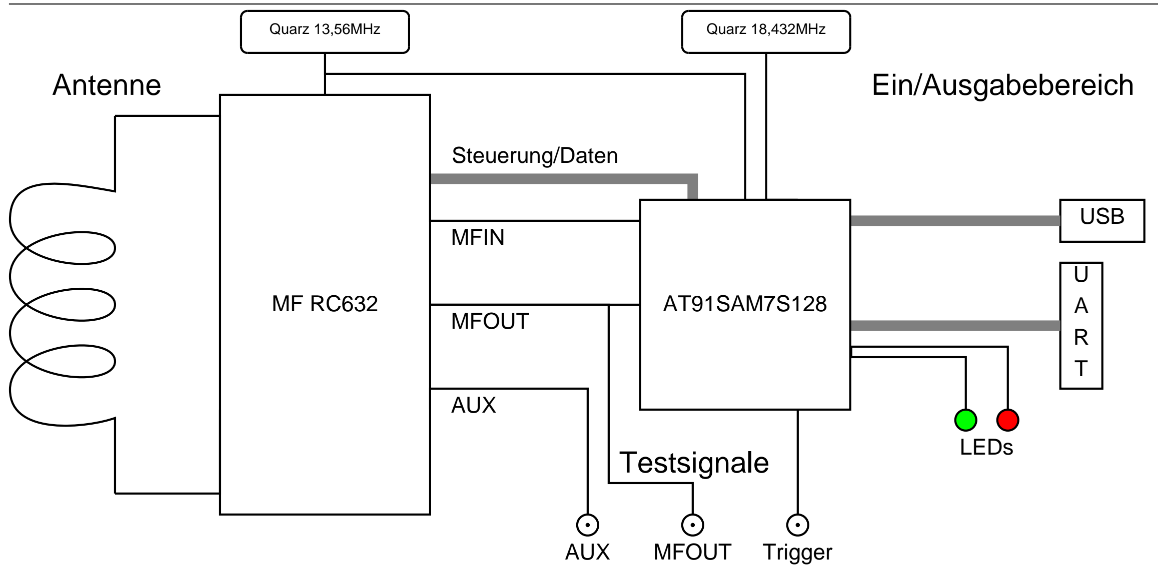
OpenPCD (dargestellt in Abbildung 3.1) benutzt den RFID-Leser-IC MF RC632 von NXP (vormals Philips) für Kommunikation nach ISO 14443 Typ A oder B sowie für die proprietäre Mifare-Classic-Verschlüsselung Crypto-1. Der RC632 ist mit dem AT91SAM7S128-Prozessor verbunden, welcher wiederum die Steuerung aller Funktionen und eventuelle Kommunikation mit einem angeschlossenen PC übernimmt.

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 PLL[2]-Schaltung zu einer Arbeitsfrequenz von etwa 48 MHz führt.

[2]PLL (*Phase Locked Loop*, Phasenregelschleife)

Abbildung 3.2. Vereinfachter Übersichtsschaltplan für OpenPCD



3.1.1. RC632

Der NXP MF RC632 ist ein Multiprotokoll-Leser-IC[3], welcher die RFID-Funkprotokolle nach ISO 14443 Typ A und B sowie ISO 15693 direkt unterstützt. Er wird mit einem Mikroprozessor über ein serielles oder paralleles Kommunikationsinterface sowie eine IRQ[4]-Leitung verbunden. Der IC selber enthält keinen vollständigen ISO-Stack, sondern übernimmt nur den kompletten Analogteil (Modulation bzw. Demodulation, Erzeugung des Trägerfeldes) der Kommunikation sowie ISO 14443 Part 2 und gegebenenfalls die proprietäre Mifare-Verschlüsselung. ISO 14443 Part 3 – und wenn gewünscht Part 4 – muss im angeschlossenen Mikroprozessor (bzw. einem dahinterstehenden Host-PC) implementiert werden.

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)

3.1.2. MFIN/MFOUT

Die Signale MFIN bzw. MFOUT am RC632 sind ein digitaler Eingang bzw. Ausgang und lassen sich für eine Reihe von Funktionen konfigurieren. Vom Hersteller vorgesehen ist ein Konzept namens *Active Antenna*. Dabei werden zwei RC632 benutzt: In einem davon wird der Digitalschaltkreis lahmgelegt und das demodulierte Empfangssignal auf MFOUT gelegt und das zu modulierende Sendesignal von MFIN gelesen. In dem anderen wird der Analogteil deaktiviert und das zu sendende kodierte Signal auf MFOUT gelegt und das zu dekodierende Signal von MFIN gelesen. Diese beiden Chips können jetzt über Kreuz ($MFIN_a$ an $MFOUT_b$, $MFIN_b$ an $MFOUT_a$) mit einer einfachen digitalen Zweidrahtleitung verbunden werden.

Der IC, in dem nur noch der Analogteil aktiv ist, kommt in unmittelbare physikalische Nähe zur Antenne (damit wird die Antenne zu einer ‚aktiven Antenne'), und der andere Chip kann in einiger Entfernung von der Antenne angebracht werden. Nur der Chip mit dem aktiven Digitalteil braucht die Schlüssel für die proprietäre Verschlüsselung zu kennen und kann vor Angreifern besser geschützt werden, etwa in Zugriffskontrollanwendungen (indem sich der Chip mit dem Digitalteil im Gebäude befindet, die aktive Antenne aber ausserhalb des Gebäudes angebracht wird).

Für Untersuchungszwecke sind MFIN und besonders MFOUT aber noch viel interessanter. Auf MFOUT können vier verschiedene Signale konfiguriert werden: Das modified Miller-kodierte Sendesignal, das unkodierte Sendesignal, das empfangene Subträgersignal sowie das empfangene Manchester-kodierte Signal nach der Subträger-Demodulation. Auf MFIN können drei Signale konfiguriert werden: das zu sendende Modulationssignal (modified Miller-kodiert, im Falle von ISO 14443-A), das zu empfangene Subträger-Signal oder das zu empfangene Manchester-kodierte Signal nach der Subträger-Demodulation. Im OpenPCD wird MFOUT auf einem Steckverbinder nach aussen geführt und kann direkt auf einem Oszilloskop dargestellt werden.

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 Digitalausgang bereitstellen. (MFIN ist außerdem mit einem speziellen Pin des Mikroprozessors verbunden, der ein beliebiges pulsweitenmoduliertes 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

Abbildung 3.3. OpenPICC-Platine v0.2

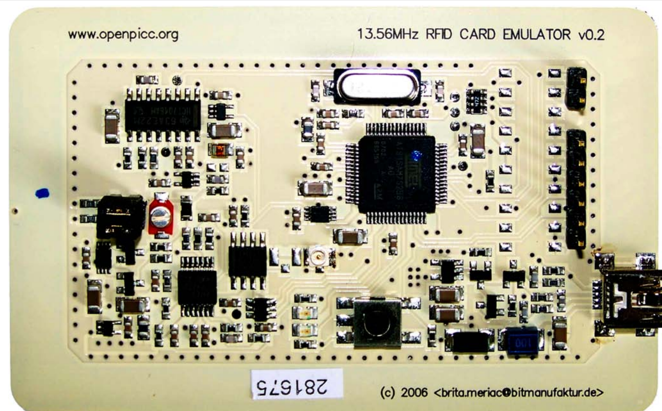
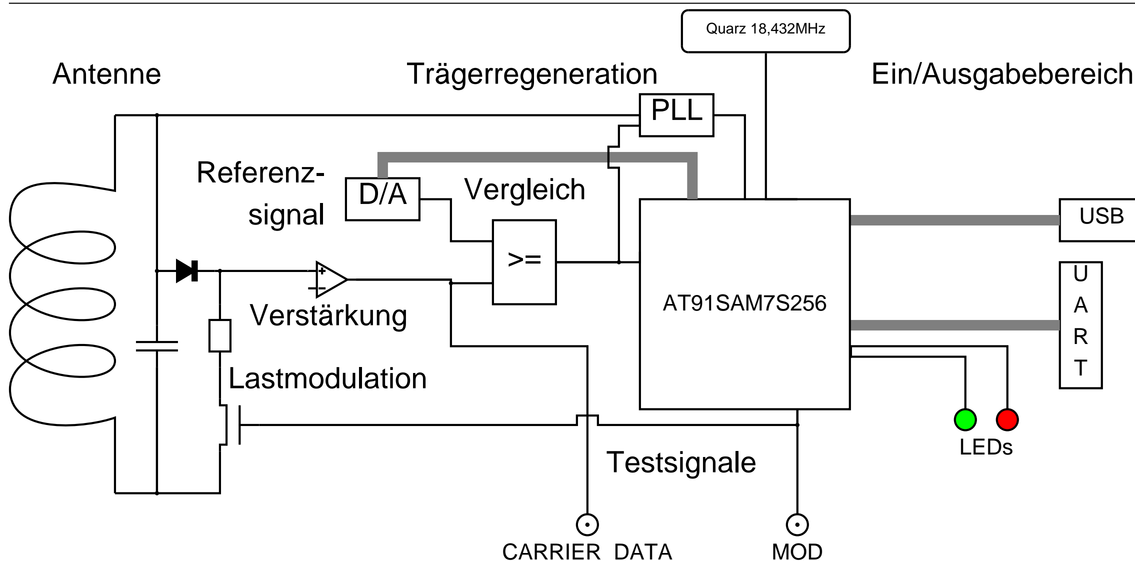


Abbildung 3.4. Vereinfachter Übersichtsschaltplan für OpenPICC v0.2 ohne Modifikationen



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.

3.2.1. Analogteil

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

Amplitudendemodulator

Zur Amplitudendemodulation wird das Antennensignal zunächst mit einer Diode durch Abschneiden einer Wellenhälfte gleichgerichtet und dann in einer mehrstufigen OpAmp[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.

Um das Analog- in ein Digitalsignal zu wandeln, wird ein Komparator benutzt, welcher das amplitudendemodulierte Signal mit einer, über einen Digital/Analogwandler vorgebbaren, Referenzspannung vergleicht. Durch Wahl der Referenzspannung kann der Mikroprozessor den Vergleichsschwellwert setzen, etwa um unterschiedliche Feldstärken bei unterschiedlicher Entfernung zum Lesegerät auszugleichen.

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

Die Aufgabe des Trägerregenerators ist es, einen kontinuierlichen Trägertakt von 13,56 MHz bereitzustellen, auch während der Träger in den Modulationszeiten von ISO 14443-2 abgeschaltet wird. Dieser Trägertakt wird an den Prozessor geleitet und kann dann (über diverse Teiler und Timer) verwendet werden, um synchron Daten zu empfangen und zu senden.

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

[5] *Operational Amplifier*, Operationsverstärker

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 Frequenzteiler 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 zuständing 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

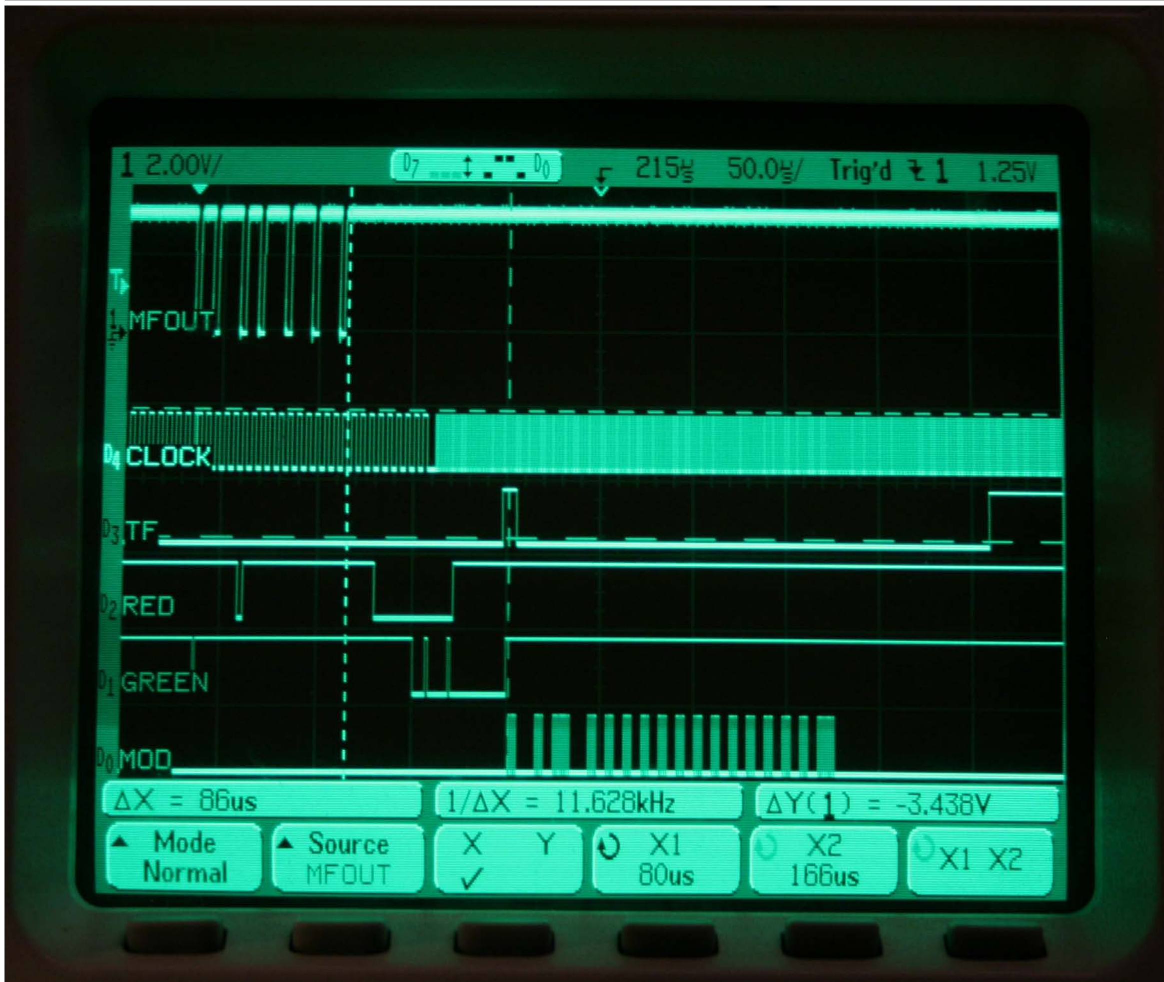
Es kann entweder der PWM-Controller oder der SSC mit dem Modulationssignal MOD verbunden werden.

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.

Abbildung 3.5. Genereller Empfangs- und Sendeablauf, am Beispiel von REQA und ATQA nach ISO 14443 Typ A

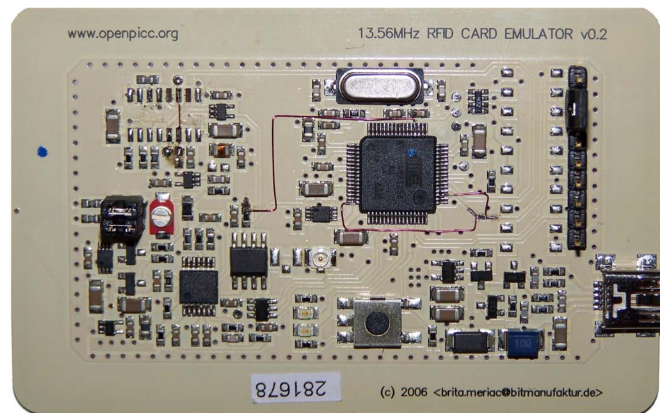


3.2.3. Empfangs- und Sendeablauf

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-

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 entprechend 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 $fc/16$-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.

Außerdem führt die starre Verknüpfung des Ausgangs des Amplitudendemodulators mit dem Auftrennen der PLL-Regelschleife dazu, dass die Trägerregeneration für andere Protokolle als ISO 14443 Typ A nicht eingesetzt werden kann. Für ISO 14443 Typ B beispielsweise ist ein Auftrennen der Regelschleife nicht notwendig, da dort nur 10% ASK eingesetzt wird. Es ist sogar schädlich, da die Regelschleife durch die starre Verbindung sehr oft und lange aufgetrennt wird und dann die Synchronisation mit dem Träger verliert.

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.

Aufgrund noch ungeklärter Probleme mit dem SSC kann auch die ursprüngliche Idee, die steigende TF-Flanke als Sendesignal zu verwenden nicht benutzt werden. Stattdessen kommt jetzt ein Umweg zum Einsatz: Das TF-Signal wird mit Trägersignal AND-verknüpft und dann wird dieses AND-verknüpfte Signal als Takt-Signal für den SSC verwendet. Damit kann der SSC direkt nach dem Vorbereiten der zu sendenden Bitfolge gestartet werden, wird aber erst aktiv, wenn der verknüpfte Takt einsetzt.

Teil II.

Analyse

Zeitlinie

28. Dezember 2007 Erster öffentlicher Vortrag zum Reverse Engineering und ersten Schwächen in Mifare Classic ([NP07]).

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

8. Januar 2008 Erster Artikel in den niederländischen Print-Medien, in dem die Verbindung zum OV-Chipkaart-Projekt gezogen wird ([Haa08]). Das Thema bleibt 2 Wochen lang Top-Thema in den Nachrichten, unterstützt durch die Demonstration eines Replay-Angriffs auf den Mifare-Ultralight-Teil des OV-Chipkaart-Projekts am 14. Januar 2008.

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

10. März 2008 NXP stellt Mifare Plus (neu) vor ([NXP08b]).

12. März 2008 Forscher der Radboud Universität Nijmegen demonstrieren einen eigenen Angriff auf Mifare Classic, indem sie Karten eines Zugangskontrollsystems der Universität kopieren ([Dig08b]). Das niederländische Innenministerium wurde bereits am 7. März informiert und hat Sofortmaßnahmen ergriffen, da es selber ähnliche Mifare-Classic-basierende Zugangskontrollsysteme einsetzt. Die Details dieses Angriffes werden noch zurückgehalten, sollen aber im Oktober 2008 veröffentlicht werden.

15. März 2008 Ein Paper zu Keystream-Recovery bei Mifare Classic wird von Mitgliedern der niederländischen Forschergruppe veröffentlicht ([KGHG08]).

15. April 2008 Vorstellung von algebraischen Attacken auf Mifare Classic ([CNO08]).

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 ([Dig08a]). Der Radboud-Angriff wird also voraussichtlich Anfang Oktober auf dem 13th European Symposium on Research in Computer Security (ESORICS08) veröffentlicht werden.

Three may keep a secret, if two of them are dead.

Benjamin Franklin

Kapitel 4

Analyse des Funk-Protokolls

Die Kombination aus OpenPICC als Amplitudendemodulator und OpenPCD als Subcarrier-Demodulator und RFID-Lesegerät erlaubt es, Experimente mit dem Protokoll durchzuführen und dabei die gesendeten Digitalsignale der beiden beteiligten Seiten (PCD und PICC) mit einem Logikanalysator aufzuzeichnen. Abbildung 4.1 auf der nächsten Seite zeigt so einen Aufbau: OpenPICC und OpenPCD liegen direkt übereinander und die Ziel-Karte ist dazwischengelegt. Die OpenPICC-Software ist konfiguriert, vollständig stumm zu sein, so dass nur der Analogteil und die Analog/Digitalwandlung aktiv sind. OpenPCD ist konfiguriert, das empfangene Subträger-demodulierte Manchester-kodierte Signal auf MFOUT (siehe Abschnitt 3.1.2 auf Seite 30) auszugeben. Außerdem ist ein mechanischer Schalter in die OpenPCD-Stromversorgung eingeschleift, um dem OpenPCD auf Wunsch vollständig die Spannung entziehen zu können.

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[1]-Daten exportiert. Eine selbst geschriebene Software übernimmt die modified Miller- und Manchester-Dekodierung der Daten.

4.1. Erste Sniffing-Ergebnisse

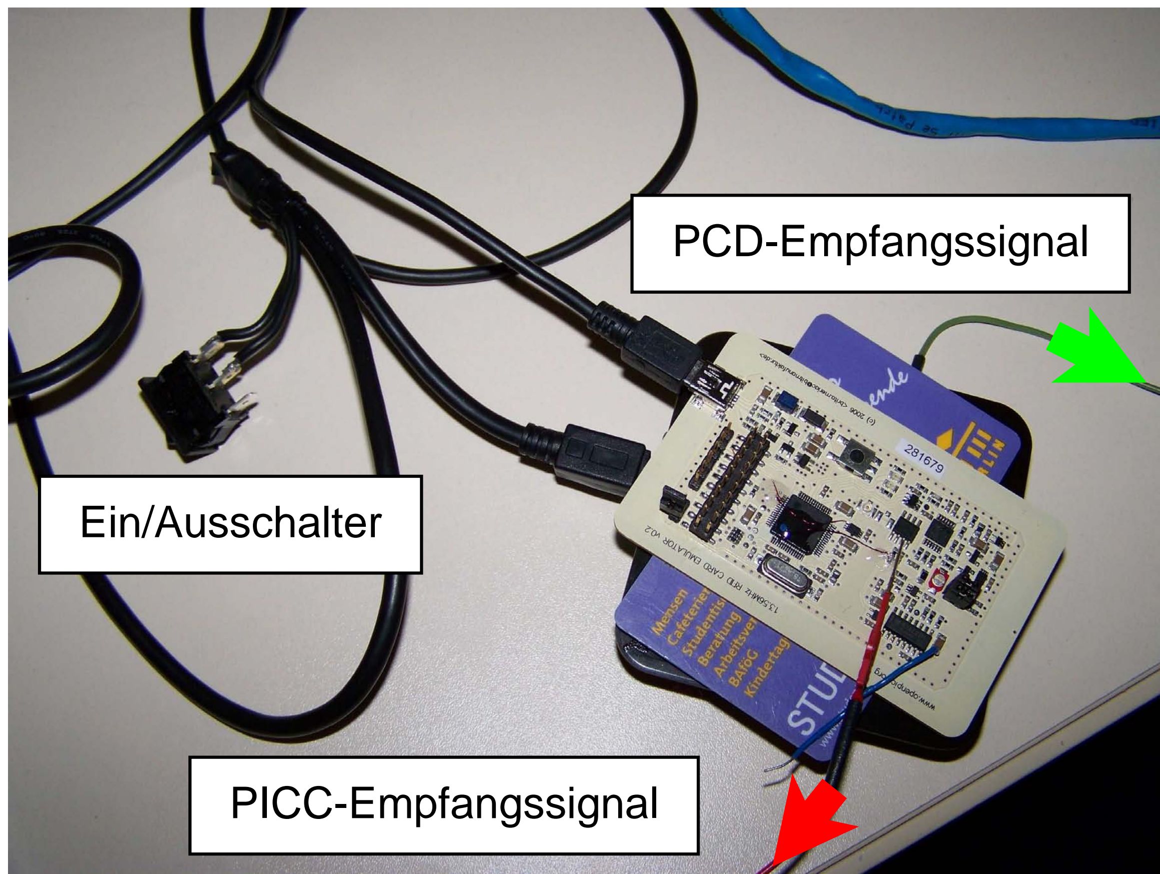
Eine der ersten mit diesem Versuchsaufbau erfolgreich gesnifften Sitzungen zeigt Tabelle 4.1 auf Seite 43. Die Spalte „Zeit“ gibt die Zeit in μs zwischen dem Beginn der vorherigen Übertragung und dem Beginn der aktuellen Übertragung an. Die Spalte „Von“ zeigt an, ob das Frame vom Lesegerät (L) oder der Karte (K) gesendet wurde. Außerdem markiert ein $\surd$ in dieser Spalte, wenn die CRC bzw. BCC des Frames korrekt ist. Die Spalte „Inhalt“ schließlich gibt den dekodierten Inhalt des Frames wieder, wobei alle Bytes deren Paritätsbits nicht stimmen, mit einem $\overline{\text{Überstrich}}$ versehen sind.

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

[1] CSV (*Comma Separated Values*, Kommaseparierte Werte)

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 Kaskadierungsebenen) 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)

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

Nr.	Zeit[μs]	Größe	Von	Inhalt
1	0	7 Bits	L	26
2	+157	2 Bytes	K	04 00
3	+34158	2 Bytes	L	93 20
4	+270	5 Bytes	K $\surd$	B4 79 F7 D7 ED
5	+46431	9 Bytes	L $\surd$	93 70 B4 79 F7 D7 ED C7 27
6	+865	3 Bytes	K $\surd$	08 B6 DD
7	+23127	4 Bytes	L $\surd$	60 00 F5 7B
8	+492	4 Bytes	K	F3 FB AE ED
9	+10515	8 Bytes	L	$\overline{\texttt{7C}}$ 74 $\overline{\texttt{07}}$ EB $\overline{\texttt{0F}}$ 7B D5 $\overline{\texttt{1B}}$
10	+775	4 Bytes	K	$\overline{\texttt{3D}}$ $\overline{\texttt{0E}}$ $\overline{\texttt{A0}}$ E2
11	+59213	4 Bytes	L	$\overline{\texttt{65}}$ $\overline{\texttt{8D}}$ 65 1F
12	+449	18 Bytes	K	52 F6 46 35 $\overline{\texttt{89}}$ $\overline{\texttt{BA}}$ E2 E9 B2 $\overline{\texttt{2D}}$ $\overline{\texttt{F8}}$ CD $\overline{\texttt{AE}}$ $\overline{\texttt{C8}}$ $\overline{\texttt{6C}}$ B2 $\overline{\texttt{DE}}$ 04

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

4.2. Wiederholte Challenge-Response-Läufe

Eines der ersten Experimente mit diesem Sniffing-Aufbau war, wiederholt das *Mutual-Authentication*-Protokoll auszuführen, um Challenge-Response-Paare für die weitere Analyse zu erhalten. Zu diesem Zweck dient der Ein/Ausschalter in der OpenPCD-Stromversorgung: Durch manuelles Ausschalten der Stromversorgung werden sowohl das Funkfeld abgeschaltet und die Karte zurückgesetzt, als auch der RC632 komplett zurückgesetzt.

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.

Tabelle 4.2 auf der nächsten Seite zeigt die ersten vier aufgezeichneten Challenge-Response-Paare. Die Kartenchallenges in den Transaktionen 2, 3 und 4 sind identisch. Mehr noch: Die Lesegerät-Challenges in den Transaktionen 2 und 4 sind identisch[2].

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, $\overline{\texttt{1E}}$ $\overline{\texttt{98}}$ $\overline{\texttt{43}}$ $\overline{\texttt{FB}}$...).

[2] Das veranlasste mich zuerst, einen Fehler im Versuchsaufbau zu vermuten der evt. alte Aufzeichnungen wieder einspielen könnte. Das hat sich aber nicht bewahrheitet.

Tabelle 4.2. Erste vier aufgezeichnete Challenge-Response-Paare

Nr.	Von der Karte	Vom Lesegerät	Von der Karte
1	FF CF 80 E3	$\overline{\texttt{38}}$ C5 $\overline{\texttt{B5}}$ 45 $\overline{\texttt{84}}$ $\overline{\texttt{D5}}$ 04 $\overline{\texttt{7F}}$	DF $\overline{\texttt{58}}$ $\overline{\texttt{61}}$ B3
2	7D DA 7E 41	$\overline{\texttt{1E}}$ $\overline{\texttt{98}}$ $\overline{\texttt{43}}$ $\overline{\texttt{FB}}$ D6 $\overline{\texttt{CD}}$ $\overline{\texttt{65}}$ $\overline{\texttt{E5}}$	A6 $\overline{\texttt{23}}$ 0A 9C
3	7D DA 7E 41	53 03 $\overline{\texttt{8F}}$ 3A $\overline{\texttt{66}}$ $\overline{\texttt{85}}$ $\overline{\texttt{D5}}$ $\overline{\texttt{48}}$	$\overline{\texttt{87}}$ $\overline{\texttt{8E}}$ 75 $\overline{\texttt{D3}}$
4	7D DA 7E 41	$\overline{\texttt{1E}}$ $\overline{\texttt{98}}$ $\overline{\texttt{43}}$ $\overline{\texttt{FB}}$ D6 $\overline{\texttt{CD}}$ $\overline{\texttt{65}}$ $\overline{\texttt{E5}}$	A6 $\overline{\texttt{23}}$ 0A 9C

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.

Um zu testen, ob die UID in die Berechnung des Initialzustands eingeht, habe ich einige Modifikationen der OpenPCD-Firmware vorgenommen. Wie schon in Abbildung 2.7 auf Seite 24 dargestellt, erwartet das Authentisierungs-Kommando auf dem RC632 die UID der Karte, die authentisiert werden soll, als Parameter. Normalerweise wird dort die UID aus der Antikollisionsphase eingegeben. Meine modifizierte Firmware erlaubt es, eine beliebig veränderte Version der UID 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_0 ... a_{b-1} \overline{a_b} a_{b+1} ... a_{Länge(a)}
end function
procedure FindeUIDBits(UID, KEY)
    M ← ∅                                   // Menge der momentan zu kippenden UID Bits
    i ← 0
    repeat
        k ← BitFlippen(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.

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

Bit im Schlüssel														Bits in UID	Erfolg
→	0													0	Ja
	1													1	Ja
	2													2	Ja
	3													3	Ja
	4													4	Ja
	5													5	Nein
→	5												0	5	Ja
	6												1	6	Ja
	7												2	7	Ja
	8												3	8	Ja
	9												4	9	Nein
→	9											0	4	9	Ja
	10											1	5	10	Nein
→	10										0	1	5	10	Ja
	11										1	2	6	11	Ja
→	12									0	2	3	7	12	Ja
	13									1	3	4	8	13	Ja
→	14								0	2	4	5	9	14	Ja
→	15							0	1	3	5	6	10	15	Ja
	16							1	2	4	6	7	11	16	Ja
→	17						0	2	3	5	7	8	12	17	Ja
	18						1	3	4	6	8	9	13	18	Ja
→	19					0	2	4	5	7	9	10	14	19	Ja
	20					1	3	5	6	8	10	11	15	20	Ja
	21					2	4	6	7	9	11	12	16	21	Ja
	22					3	5	7	8	10	12	13	17	22	Ja
	23					4	6	8	9	11	13	14	18	23	Ja
→	24				0	5	7	9	10	12	14	15	19	24	Ja
→	25			0	1	6	8	10	11	13	15	16	20	25	Ja
	26			1	2	7	9	11	12	14	16	17	21	26	Ja
→	27		0	2	3	8	10	12	13	15	17	18	22	27	Ja
	28		1	3	4	9	11	13	14	16	18	19	23	28	Ja
→	29	0	2	4	5	10	12	14	15	17	19	20	24	29	Ja
	30	1	3	5	6	11	13	15	16	18	20	21	25	30	Ja
	31	2	4	6	7	12	14	16	17	19	21	22	26	31	Ja

Kapitel 5

My God, it's full of stars.

David Bowman in *2010: The Year We Make Contact*

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 GSM[1]-Kommunikation eingesetzt werden ([And94]). Auch der CSS[2]-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 ([NEsP08, KNP08]).

Die Hauptarbeit wurde hier von Jan Krissler (auch bekannt als starbug) vom Berliner Chaos Computer Club und von Karsten Nohl von der Universität Virginia ausgeführt. Der Vorgang teilt sich in zwei Schritte: Zunächst muss man den Chip öffnen und fotographieren und dann müssen die Fotos ausgewertet werden.

5.1. Chip öffnen und fotographieren

Um mit dem Chip arbeiten zu können, muss dieser zunächst aus dem Kartenkörper gelöst werden[3]. Dazu eignen sich rauchende Salpetersäure oder Aceton (letzteres ist weniger

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

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

gefährlich 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 schiefe Schnittbilder nimmt und nur die Teile, die dieselbe Ebene zeigen, miteinander kombiniert.

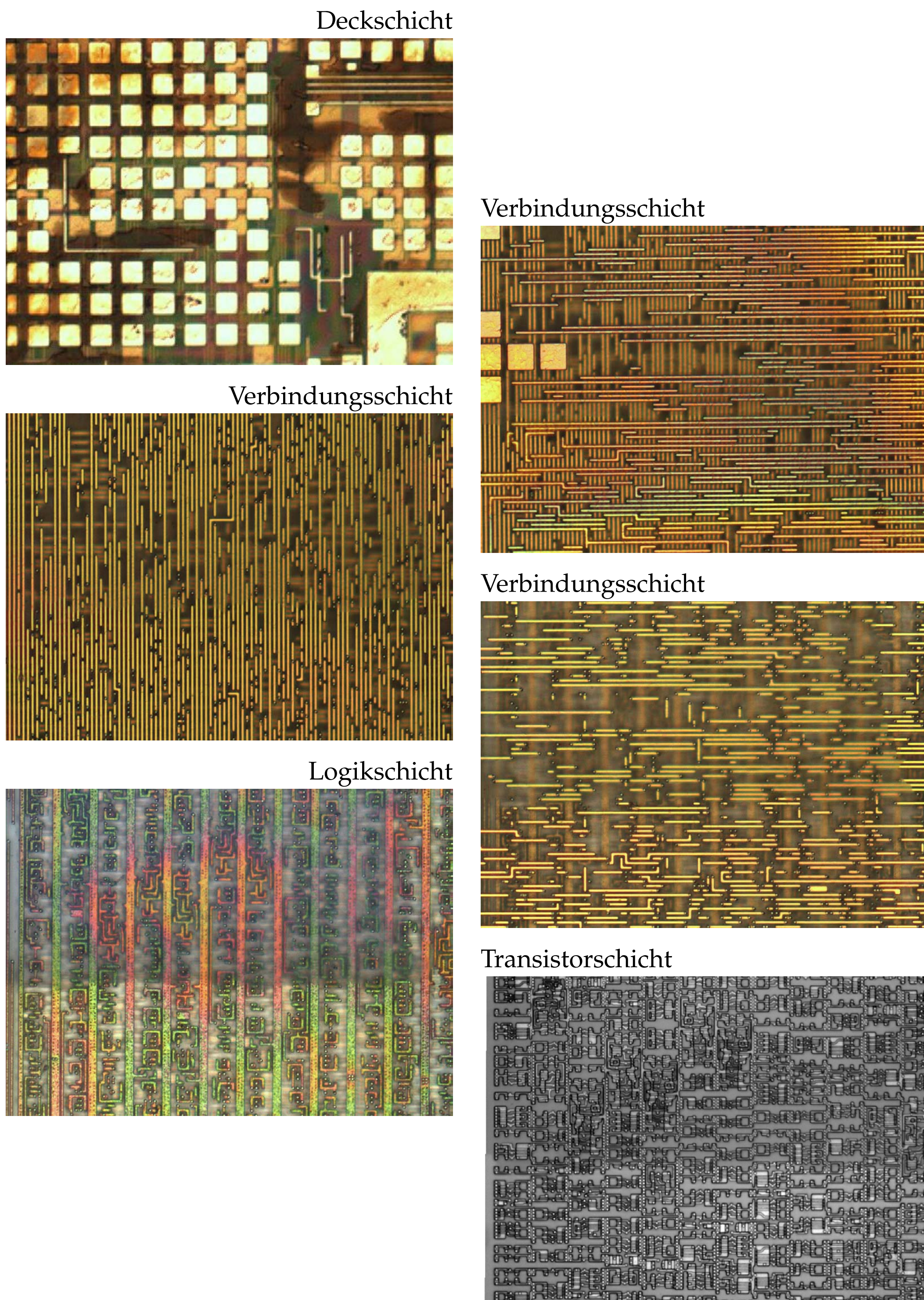
Jetzt wird der Chip von oben nach unten vorsichtig abpoliert und zwischendurch immer wieder unter einem Lichtmikroskop (500x Vergrößerung) kontrolliert und photographiert. Da die von der Kamera abgebildete Oberfläche nicht den vollen Chip umfasste, wurden pro Polierschritt mehrere (ca. 20) Bilder gemacht, die jeweils einen anderen Ausschnitt zeigen. Diese Bilder wurden dann in einer Panoramasoftware (hugin, [hug]) zusammengesetzt, um ein Gesamtbild der jeweiligen Ebene zu erzeugen.

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



können die Transistoren und deren Verschaltung miteinander rekonstruiert werden. Ein oder mehrere Transistoren sind zu Gates zusammengefasst, welche bestimmte logische Funktionen erfüllen: NOT, NOR, NAND, usw. Da es auf dem gesamten Chip ca. 10 000 Gates gibt, wäre es sehr langwierig, wenn man diese alle einzeln betrachten müsste. Zum Glück stammen diese aus einer Bibliothek von nur ca. 70 unterschiedlichen Gates, wobei es teilweise unterschiedliche Varianten für die gleiche Funktionalität gibt.

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 Krissler 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 vorhersagbar 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 $\frac{13{,}56\,\text{MHz}}{128} \approx 105{,}9\,\text{kHz}$ getaktet.

Da das Register nur $2^{16} - 1$ Zustände[1] 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.

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

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

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 dritte *Nonce*, 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]

```
#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", bit);
           /* 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

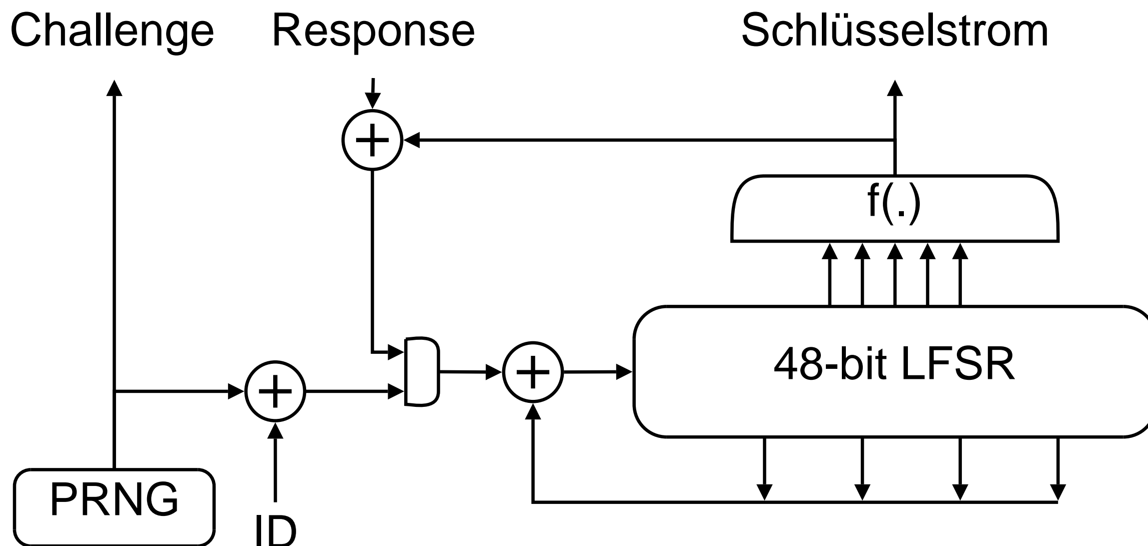


Abbildung 6.2. Detailansicht der Filterfunktion $f(\cdot)$ von Crypto-1

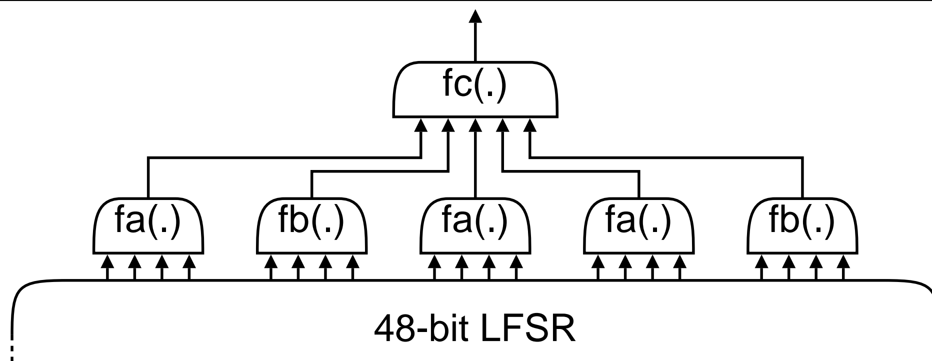


Abbildung 6.1 zeigt die Übersicht über den Crypto-1-Stromchiffregenerator wie sie zuerst in [NP07] veröffentlicht wurde. Abbildung 6.2 zeigt ein Detail der Filterfunktion, wie es zuerst in [Noh08] veröffentlicht wurde. Die folgende Beschreibung der Chiffre ist nicht vollständig, aber ausreichend, um alle Protokollangriffe ableiten zu können und umfasst alle vormals veröffentlichten Details. Die vollständige Chiffre findet sich als Beispielimplementierung in Anhang A ab Seite 71.

Zur Initialisierung und *mutual authentication* (auf der Kartenseite) wird das Zustandsshiftregister 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[2] in das Zustandsshiftregister geschoben (während dieser

[2]Die 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.

Zeit wird der generierte Schlüsselstrom verworfen). An dieser Stelle wird die Verschlüsselung aktiviert und alle aus- bzw. eingehenden Bits werden mit dem Schlüsselstrom XOR-verknüpft. Wenn die Tag-Nonce vom Lesegerät empfangen wurde und dieses damit sein Zustandsregister aktualisiert hat, generiert das Lesegerät seinerseits eine Nonce, aktualisiert mit dieser seinen Zustand und sendet sie verschlüsselt zum Tag. Anschließend werden, zuerst vom Lesegerät, dann vom Tag, jeweils 4 Byte als Antwort auf die *Challenge* erzeugt und gesendet, die aus dem reinen Schlüsselstrom bestehen. Tag und Lesegerät überprüfen die Antwort der jeweils anderen Seite und wenn sie korrekt ist, ist die *mutual authentication* erfolgreich durchgeführt und die Crypto-1-Chiffre initialisiert.

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.

> The single biggest problem with communication is the illusion that it has taken place.
>
> George Bernard Shaw

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

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

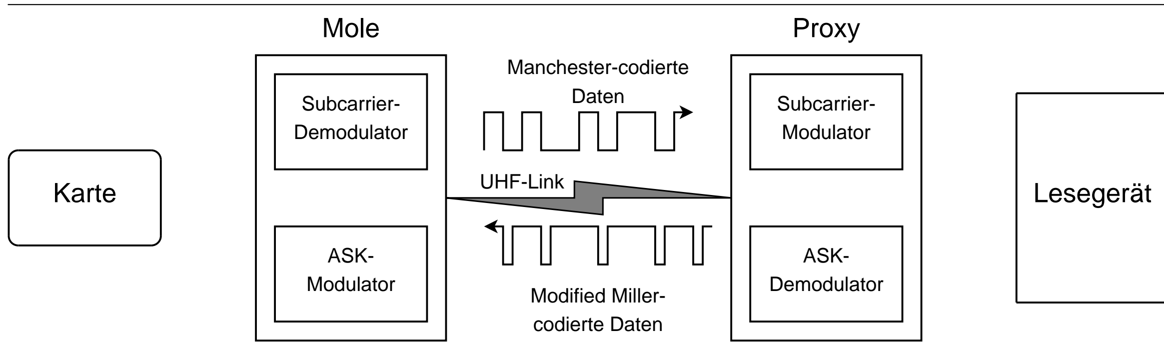
, um alle möglichen Schlüssel im Versuchsaufbau von Grunwald auszuprobieren (bei 365,25 Tagen pro Jahr). Falls die Wahrscheinlichkeit für alle Schlüssel gleich ist, darf im Erwartungswert damit gerechnet werden, dass der korrekte Schlüssel nach der Hälfte der Zeit, also ca. 111 493 Jahren gefunden wird.

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-

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.

Eine interessante Erkenntnis aus Hancke's Versuchen ist, dass das Timing zwischen Karte und Lesegerät wesentlich weniger strikt ist, als vom Standard vorgegeben. Selbst mit den ca. 20 μs Verzögerung, die durch den Versuchsaufbau eingefügt wurden, funktionierte das Selektieren der Karte und die Datenübertragung noch. Lediglich die Antikollisionsprozedur beim Vorhandensein einer anderen Karte nahe dem Lesegerät, zusätzlich zum *Proxy*, würde nicht mehr funktionieren, was aber bei der Angriffsart kein Problem darstellt, da die nähere Umgebung des Lesegeräts vom Angreifer kontrolliert wird. Mehrere Karten in der Nähe des *Mole* stellen kein Problem dar, da ihre Signale alle mit der gleichen Verzögerung übermittelt werden.

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[1]. 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 Maximalabstand zwischen *Mole*/*Leech* und Karte von etwa 40 cm und zwischen *Proxy*/*Ghost* und Lesegerät von etwa 50 m.

7.2.1. Gegenmaßnahmen

Relaying ist eine generische Klasse von Angriffen, die im Prinzip auf jede Art von kontaktloser Kommunikation anzuwenden ist. Gegenmaßnahmen beruhen oftmals auf Zeitmessung zwischen Anfrage des Lesers und der Antwort der Karte und diese sind bei Mifare Classic nicht möglich: Der einzige zeitkritische Teil der Kommunikation ist die Antikollisionsprozedur, die aber vom *Ghost* autonom abgewickelt werden kann, sobald er die UID der angegriffenen Karte kennt. Bei der eigentlichen Mifare-Classic-Authentisierung sind keine derart engen Zeitgrenzen vorgesehen, so dass ein Angreifer in der Regel mindestens 1 ms Zeit zum Forwarden hat.

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 einzugrenzen, 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] In der Praxis ist er vermutlich noch geringer. Eigene Experimente mit dem Nokia 6131 NFC zeigen einen Arbeitsabstand im Bereich von 2 bis 4 cm.

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 verstellt, 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.

Beide Maßnahmen reduzieren den gefühlten Komfort des Benutzers deutlich, bis in den Bereich des Komforts einer kontaktbehafteten Karte. Bei der zweiten Maßnahme wird außerdem ein zusätzliches Verschleißteil eingeführt, welches die erwartete Lebensdauer der Karte senkt.

7.3. Man-in-the-Middle-Attacken

Eine Schwäche im Mifare-Funkprotokoll wird in [Han05] erstmals indirekt erwähnt[2]: 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

$$\mathrm{CRC}(n \oplus x) \quad = \quad \mathrm{CRC}(m) \oplus \mathrm{CRC}(x)$$

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:

$$\begin{aligned} c &= g(k) \oplus m \\ m' &= g(k) \oplus c \\ &= g(k) \oplus g(k) \oplus m \\ &= m \end{aligned}$$

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

[2] Weiterhin 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:

$$\begin{aligned} m &= n \parallel \mathrm{CRC}(n) \\ c &= g(k) \oplus (n \parallel \mathrm{CRC}(n)) \end{aligned}$$

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

$$\begin{aligned} c' &= c \oplus (x \parallel \mathrm{CRC}(x)) \\ &= (g(k) \oplus (n \parallel \mathrm{CRC}(n))) \oplus (x \parallel \mathrm{CRC}(x)) \\ &= g(k) \oplus ((n \oplus x) \parallel (\mathrm{CRC}(n) \oplus \mathrm{CRC}(x))) \\ &= g(k) \oplus ((n \oplus x) \parallel (\mathrm{CRC}(n \oplus x))) \end{aligned}$$

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

$$\begin{aligned} m'' &= g(k) \oplus c' \\ &= g(k) \oplus g(k) \oplus ((n \oplus x) \parallel (\mathrm{CRC}(n \oplus x))) \\ &= (n \oplus x) \parallel \mathrm{CRC}(n \oplus x) \end{aligned}$$

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

Für diese Methode muss der Angreifer sowohl die Kommunikation verändern können, als auch wenigstens Teile des übertragenen Klartextes kennen oder raten können. Der erste Punkt ist beispielsweise bei einer Relay-Attacke erfüllt, wenn der Angreifer nicht nur die demodulierten und noch kodierten Signale überträgt, sondern zusätzlich noch Manchester- bzw. Modified Miller-Kodierer bzw. Dekodierer verwendet. Im Sinne der Man-in-the-Middle-Attacke ist es auch gar nicht notwendig, die Reichweite der Karte-Lesegerät-Verbindung zu erhöhen und je nach Angriffsziel auch nicht erwünscht. Das löst das Problem der Relay-Attacke, dass der Angreifer in der Regel einen Komplizen braucht, welcher den *mole*/*leech* in die Nähe der Karte des Opfers bringt, da eine MITM-Attacke auch bei Karten im eigenen Besitz sinnvolle Ergebnisse erzielt.

Der zweite Punkt – Kenntnis des Klartextes – ist je nach Szenario und Ziel des Angriffs ebenfalls einfach. Ein mögliches Szenario stellt die Benutzung einer Mifare-Classic-Karte als elektronische Börse in einer Mensa oder Cafeteria dar. Dort gibt es zwei Hauptangriffsziele: Aufladeautomaten und Kassen.

Im Falle des Aufladeautomaten kann der Angreifer den aufgeladenen Betrag verändern. Selbst wenn er die übertragenen Daten nicht entschlüsseln kann, kann er aufgrund der Länge bestimmte Rückschlüsse auf die eingesetzten Kommandos ziehen. Beispiele sind in Tabelle 7.1 auf der nächsten Seite aufgelistet. Besonders gut sind die Aufwert- bzw. Abbuchungskommandos auszunutzen, da Art der übertragenen Daten bekannt ist: 4 Byte Ganzzahl. Es ist sinnvoll anzunehmen, dass die meisten Kassensysteme dort den tatsächlichen Geldbörsenbetrag in Cents speichern werden. Beim Aufladeautomaten kennt der Angreifer also den gesamten Betrag im Voraus und kann ihn beliebig manipulieren.

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

Kommando(sequenz)	Beobachtbare Längen	
Sektor lesen	Lesegerät	4 Byte
	Karte	18 Byte
Sektor schreiben	Lesegerät	4 Byte
	Karte	4 Bit
	Lesegerät	18 Bytes
	Karte	4 Bit
Aufwerten/Abbuchen	Lesegerät	4 Byte
	Karte	4 Bit
	Lesegerät	6 Byte
Transfer	Lesegerät	4 Byte
	Karte	4 Bit

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` $\oplus$ `00 00 04 00h`.

Der Angriff auf das Kassensystem funktioniert ähnlich, nur andersherum. Hier sollte der Angreifer schon vorher den Preis ausrechnen, um die zu kippenden Bits bestimmen zu können. Je nach Implementierung ist dieser Angriff weniger erstrebenswert, da das Kassensystem unter Umständen den neuen Betrag von der Karte ausliest und anzeigt. Eine aufmerksame Kassiererin könnte dann bemerken, dass der Kartenwert nicht so stark gesunken ist, wie zu erwarten war. Das kann ausgeglichen werden, indem der Angreifer die Manipulation auch auf das Lesekommando ausweitet.

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 Lesekommando 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 sind opaque, 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 Zugriffsrechte des Sektor-Trailers (Tabelle 2.3 auf Seite 22) zurückgeht: Schlüssel A ist niemals lesbar, der Klartext der ersten 6 Bytes des Sektor-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 Zugriffsrechte 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

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 natürlich 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 ggbf. 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.

Kapitel 8

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> John von Neumann

Angriffe auf die Crypto-1-Chiffre

8.1. Schwächen in der Initialisierung

Die Crypto-1-Initialisierung verwendet die Karten-UID und die Karten-*Nonce* während der anfänglichen Vorbereitung des Zustandsshiftregisters. In der Theorie sorgt die Einbeziehung der Karten-UID dafür, dass unterschiedliche Karten unterschiedliche Initialzustände erzeugen, so dass jeder abgeleitete Sitzungszustand kartenspezifisch ist. Das würde zum Beispiel eine Replay- oder MITM-Attacke verhindern, bei der eine falsche UID in der Antikollisionsphase vorgetäuscht wird.

Da die UID und *Nonce* aber XOR-verknüpft werden und beide Werte 32 Bit lang sind, kann für jede Kommbination von $(\text{UID}_x, \mathit{Nonce}_x)$ und $(\text{UID}_y, \mathit{Nonce}_y)$, 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 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.

8.2. Schwächen in der Schlüsselstromgenerierung

Fehlende Nichtlinearität

Durch die Linearität des Zustands-LFSR können dessen Zustände beliebig vorwärts und rückwärts berechnet werden, wenn der Zustand zu einem bestimmten Zeitpunkt bekannt ist ([SPMR06], zeigt dieses Vorgehen für ein CRC-LFSR). Um den Zustand des Registers zu einem bestimmten Zeitpunkt zu bestimmen, kommt zum Beispiel eine Lookup-Tabelle im Zusammenhang mit einem *Known-Plaintext*-Angriff in Frage. Der nötige Klartext kann mit dem Angriff aus Abschnitt 7.5 auf Seite 63 beschafft werden.

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

Der bisher beste bekannte Angriff auf Crypto-1 ist die algebraische Attacke, die von Courtois, Nohl und O'Neil in [CNO08] beschrieben wird. Dieser Angriff funktioniert, indem Courtois et al. ein Gleichungssystem aufstellen, welches die Schlüssel- und Schlüsselstrombits sowie eine große Anzahl an Zwischenvariablen enthält. Dieses Gleichungssystem wird dann in eine Instanz des SAT-Problems umgewandelt und mit einem frei verfügbaren SAT-Lösungsprogramm gelöst.

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.

Der Angriff benötigt ungefähr 50 bekannte Schlüsselstrombits, die zum Beispiel mit dem Angriff aus Abschnitt 7.5 auf Seite 63 oder mit einer *Chosen-Plaintext*-Attacke auf ein Lesegerät gewonnen werden können. Die Verarbeitungsdauer liegt – auf einem Standard-PC ohne besondere Hardware und ohne vorberechnete Tabellen – bei etwa 200 Sekunden. Mit 4 gewählten Karten-*Nonces* kann der Schlüssel bereits in ca. 12 Sekunden gebrochen werden.

Teil IV.

Abschluss

Kapitel 9

> Any sufficiently advanced technology is indistinguishable from magic.
>
> Third Law of Prediction,
> Arthur C. Clarke

Zusammenfassung und Ausblick

Die Arbeit hat gezeigt, dass der am weitesten verbreitete Typ von kontaktlosen Speicherkarten mit Sicherheitsfunktionen wesentlich weniger sicher ist, als zunächst angenommen. Die Vielzahl und Art der aufgezeigten Angriffe reduziert die Sicherheit dieses Kartentyps in die Nähe einer Speicherkarte ohne Sicherheitsfunktionen.

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.

Ein anderes offenes Forschungsgebiet, an dem aber bereits gearbeitet wird, ist die weitergehende Automatisierung der Reverse-Engineering-Techniken. Prototypen einer Software, die selbsttätig Leiterbahnen verfolgen und die Gate-Verknüpfung rekonstruieren kann, existieren bereits und leisten bei anderen Projekten gute Dienste. Optimal wäre es, wenn man dieses Programm mit einer Simulationsumgebung kombinieren würde, die aufgrund der erkannten Verknüpfungen die Funktion erkennen und simulieren kann. Das wäre dann ein vollständiges Softwarepaket, welches nur noch Chip-Fotos als Eingabe benötigt und daraus vollautomatisiert den Algorithmus extrahieren kann.

Anhang A

Crypto-1-Beispielimplementierung

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

```
1  /*
2   * Philips/NXP Mifare Crypto-1 implementation v1.0
3   *
4   * By Karsten Nohl, Henryk Plötz, Sean O'Neil
5   *
6   */
7
8  #ifndef CRYPTO1_H_
9  #define CRYPTO1_H_
10
11 #ifdef HAVE_SYS_TYPES_H
12 #include <sys/types.h>
13 #else
14 typedef unsigned int size_t;
15 #endif
16
17 #ifdef HAVE_STDINT_H
18 #include <stdint.h>
19 #else
20 typedef unsigned char uint8_t;
21 typedef unsigned short uint16_t;
22 typedef unsigned long uint32_t;
23 typedef unsigned long long uint64_t;
24 #endif
25
26 /*
27  * This type holds data bytes with associated parity bits.
28  * The data is in the low byte while the associated parity bit
29  * is in the least-significant bit of the high byte.
30  */
31 typedef uint16_t parity_data_t;
32
33 struct _crypto1_state;
34
35 struct _crypto1_ops {
36   void(*init)(struct _crypto1_state *state, const uint64_t key);
```

```
37   void(*mutual_1)(struct _crypto1_state *state, const uint32_t uid, const
         uint32_t card_challenge);
38   void(*mutual_2_reader)(struct _crypto1_state *state, parity_data_t
         *reader_response);
39   int (*mutual_3_reader)(struct _crypto1_state *state, const parity_data_t
         *card_response);
40   int (*mutual_2_card)(struct _crypto1_state *state, const parity_data_t
         *reader_response);
41   void(*mutual_3_card)(struct _crypto1_state *state, parity_data_t
         *card_response);
42   void(*transcrypt_bits)(struct _crypto1_state *state, parity_data_t *data,
         size_t bytes, size_t bits);
43 };
44
45 typedef struct _crypto1_state {
46   uint64_t lfsr; /* The 48 bit LFSR for the main cipher state
47                   * and keystream generation */
48   uint16_t prng; /* The 16 bit LFSR for the card PRNG state,
49                   * also used during authentication. */
50
51   uint8_t is_card; /* Boolean whether this instance should
52                     * perform authentication in card mode. */
53
54   const struct _crypto1_ops *ops;
55 } crypto1_state;
56
57 enum crypto1_cipher_implementation {
58   CRYPTO1_IMPLEMENTATION_CLEAN,
59   CRYPTO1_IMPLEMENTATION_OPTIMIZED
60 };
61
62 int crypto1_new(crypto1_state *state, uint8_t is_card, enum
       crypto1_cipher_implementation implementation);
63 void crypto1_init(crypto1_state *state, uint64_t key);
64 void crypto1_mutual_1(crypto1_state *state, uint32_t uid, uint32_t
       card_challenge);
65 int crypto1_mutual_2(crypto1_state *state, parity_data_t *reader_response);
66 int crypto1_mutual_3(crypto1_state *state, parity_data_t *card_response);
67 void crypto1_transcrypt(crypto1_state *state, parity_data_t *data, size_t
       length);
68 void crypto1_transcrypt_bits(crypto1_state *state, parity_data_t *data, size_t
       bytes, size_t bits);
69
70 int _crypto1_new_opt(crypto1_state *state);
71 int _crypto1_new_clean(crypto1_state *state);
72
73 /* Reverse the bit order in the 8 bit value x */
74 #define rev8(x) ((((x)>>7)&1) ^((((x)>>6)&1)<<1)^\
75               ((((x)>>5)&1)<<2)^((((x)>>4)&1)<<3)^\
76               ((((x)>>3)&1)<<4)^((((x)>>2)&1)<<5)^\
77               ((((x)>>1)&1)<<6)^(((x)&1)<<7))
78 /* Reverse the bit order in the 16 bit value x */
79 #define rev16(x) (rev8 (x)^(rev8 (x>> 8)<< 8))
80 /* Reverse the bit order in the 32 bit value x */
81 #define rev32(x) (rev16(x)^(rev16(x>>16)<<16))
```

```
82  /* Return the nth bit from x */
83  #define bit(x,n) (((x)>>(n))&1)
84
85  /* Convert 4 array entries (a[0], a[1], a[2] and a[3]) into a 32 bit integer,
86   * where a[0] is the MSByte and a[3] is the LSByte */
87  #define ARRAY_TO_UINT32(a) ( ((uint32_t)((a)[0]&0xff) << 24) | \
88                         ((uint32_t)((a)[1]&0xff) << 16) | \
89                         ((uint32_t)((a)[2]&0xff) << 8) | \
90                         ((uint32_t)((a)[3]&0xff) << 0) )
91  /* Copy an uint32_t i into 4 array entries (a[0], a[1], a[2] and a[3]), where
92   * a[0] is the MSByte */
93  #define UINT32_TO_ARRAY(i, a) do{ \
94    (a)[0] = ((i)>>24) & 0xff; \
95    (a)[1] = ((i)>>16) & 0xff; \
96    (a)[2] = ((i)>> 8) & 0xff; \
97    (a)[3] = ((i)>> 0) & 0xff; \
98  }while(0);
99
100 /* Calculate the odd parity bit for one byte of input */
101 #define ODD_PARITY(i) (( (i) ^ (i)>>1 ^ (i)>>2 ^ (i)>>3 ^ \
102                     (i)>>4 ^ (i)>>5 ^ (i)>>6 ^ (i)>>7 ^ 1) & 0x01)
103
104 /* Like UINT32_TO_ARRAY, but put the correct parity in the 8th bit of each array
105  * entry (thus the array entries must at least be 16 bit wide) */
106 #define UINT32_TO_ARRAY_WITH_PARITY(i, a) do{ \
107   (a)[0] = ((i)>>24) & 0xff; (a)[0] |= ODD_PARITY((a)[0])<<8; \
108   (a)[1] = ((i)>>16) & 0xff; (a)[1] |= ODD_PARITY((a)[1])<<8; \
109   (a)[2] = ((i)>> 8) & 0xff; (a)[2] |= ODD_PARITY((a)[2])<<8; \
110   (a)[3] = ((i)>> 0) & 0xff; (a)[3] |= ODD_PARITY((a)[3])<<8; \
111 }while(0);
112
113 #endif /*CRYPTO1_H_*/
```

Listing A.2: `crypto1.c` – API

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

#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);
}

/*
```

```
53   * Second stage of mutual authentication.
54   * If this is the reader side, then the first 4 bytes of reader_response must
55   * be preloaded with the reader nonce (and parity) and all 8 bytes will be
56   * computed to be the correct reader response to the card challenge.
57   * If this is the card side, then the response to the card challenge will be
58   * checked.
59   */
60  int crypto1_mutual_2(crypto1_state *state, parity_data_t *reader_response)
61  {
62        if(state->is_card) {
63              return state->ops->mutual_2_card(state, reader_response);
64        } else {
65              state->ops->mutual_2_reader(state, reader_response);
66              return 1;
67        }
68  }
69
70  /*
71   * Third stage of mutual authentication.
72   * If this is the reader side, then the card response to the reader
73   * challenge will be checked.
74   * If this is the card side, then the card response to the reader
75   * challenge will be computed.
76   */
77  int crypto1_mutual_3(crypto1_state *state, parity_data_t *card_response)
78  {
79        if(state->is_card) {
80              state->ops->mutual_3_card(state, card_response);
81              return 1;
82        } else {
83              return state->ops->mutual_3_reader(state, card_response);
84        }
85  }
86
87  /*
88   * Perform the Crypto-1 encryption or decryption operation on 'length' bytes
89   * of data with associated parity bits.
90   */
91  void crypto1_transcrypt(crypto1_state *state, parity_data_t *data, size_t length)
92  {
93        crypto1_transcrypt_bits(state, data, length, 0);
94  }
95
96  /*
97   * Perform the Crypto-1 encryption or decryption operation on 'bytes' bytes
98   * of data with associated parity bits.
99   * The additional parameter 'bits' allows processing incomplete bytes after the
100  * last byte. That is, if bits > 0 then data should contain (bytes+1) bytes where
101  * the last byte is incomplete.
102  */
103 void crypto1_transcrypt_bits(crypto1_state *state, parity_data_t *data, size_t
        bytes, size_t bits)
104 {
105       state->ops->transcrypt_bits(state, data, bytes, bits);
106 }
```

Listing A.3: `crypto1_clean.c` – Unoptimierte direkte C-Implementierung

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

#include "crypto1.h"

/* IMPLEMENTATION section ==================== */
/* == PRNG function ========================= */

/* 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;

  /* The register is stored and returned in reverse bit order, this way, even
   * if we cast the returned 32 bit value to a 16 bit value, the necessary
   * state will be retained. */
  prng = rev32(prng);
  for (int i = 0; i < n; i++)
    prng = (prng<<1) | ( ((prng>>15)^(prng>>13)^(prng>>12)^(prng>>10)) & 1 );
  return rev32(prng);
}

/* == keystream generating filter function === */
/* This macro selects the four bits at offset a, b, c and d from the value x
 * and returns the concatenated bitstring x_d || x_c || x_b || x_a as an integer
 */
#define i4(x,a,b,c,d) ((uint32_t)( \
    (((x)>>(a)) & 1)<<0 \
  | (((x)>>(b)) & 1)<<1 \
  | (((x)>>(c)) & 1)<<2 \
  | (((x)>>(d)) & 1)<<3 \
))

/* These macros are linearized boolean tables for the output filter functions.
 * E.g. fa(0,1,0,1) is (mf2_f4a >> 0x5)&1
 */
const uint32_t mf2_f4a = 0x9E98;
const uint32_t mf2_f4b = 0xB48E;
const uint32_t mf2_f5c = 0xEC57E80A;

/* 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
                         * and when key stream is used.
```

```
55                      * Irrelevant for software implementations, but important
56                      * to consider in side-channel attacks */
57
58   const uint32_t i5 = ((mf2_f4b >> i4 (x, 7+d, 9+d,11+d,13+d)) & 1)<<0
59                     | ((mf2_f4a >> i4 (x,15+d,17+d,19+d,21+d)) & 1)<<1
60                     | ((mf2_f4a >> i4 (x,23+d,25+d,27+d,29+d)) & 1)<<2
61                     | ((mf2_f4b >> i4 (x,31+d,33+d,35+d,37+d)) & 1)<<3
62                     | ((mf2_f4a >> i4 (x,39+d,41+d,43+d,45+d)) & 1)<<4;
63
64   return (mf2_f5c >> i5) & 1;
65 }
66
67 /* == LFSR state update functions ============ */
68 /* Updates the 48-bit LFSR in state using the mifare taps, optionally
69  * XORing in 1 bit of additional input, optionally XORing in 1 bit of
70  * cipher stream output (e.g. feeding back the output).
71  * Return current cipher stream output bit. */
72 static uint8_t mifare_update (struct _crypto1_state * const state,
73     const uint8_t injection, const uint8_t feedback)
74 {
75   const uint64_t x = state->lfsr;
76   const uint8_t ks = mf20(state->lfsr);
77
78   state->lfsr = (x >> 1) |
79    ((((x >> 0) ^ (x >> 5)
80     ^ (x >> 9) ^ (x >> 10) ^ (x >> 12) ^ (x >> 14)
81     ^ (x >> 15) ^ (x >> 17) ^ (x >> 19) ^ (x >> 24)
82     ^ (x >> 25) ^ (x >> 27) ^ (x >> 29) ^ (x >> 35)
83     ^ (x >> 39) ^ (x >> 41) ^ (x >> 42) ^ (x >> 43)
84     ^ injection ^ (feedback?ks:0) ) & 1) << 47);
85
86   return ks;
87 }
88
89 /* Update the 48-bit LFSR in state using the mifare taps 8 times, optionally
90  * XORing in 1 bit of additional input per step (LSBit first).
91  * Return corresponding cipher stream. */
92 static uint8_t mifare_update_byte (struct _crypto1_state * const state,
93     const uint8_t injection, const uint8_t feedback)
94 {
95   uint8_t ret = 0;
96   for(int i = 0; i < 8; i++)
97     ret |= mifare_update(state, bit(injection, i), feedback) << i;
98   return ret;
99 }
100
101 /* Update the 48-bit LFSR in state using the mifare taps 32 times, optionally
102  * XORing in 1 byte of additional input per step (MSByte first).
103  * Return the corresponding cipher stream. */
104 static uint32_t mifare_update_word (struct _crypto1_state * const state,
105     const uint32_t injection, const uint8_t feedback)
106 {
107   uint32_t ret = 0;
108   for(int i = 3; i >= 0; i--)
```

```
109     ret |= (uint32_t)mifare_update_byte(state, (injection >> (i*8)) & 0xff,
            feedback) << (i*8);
110   return ret;
111 }
112 
113 
114 
115 /* API section =============================== */
116 /* Initialize the LFSR with the key */
117 static void crypto1_clean_init(struct _crypto1_state * const state,
118     const uint64_t key)
119 {
120   state->lfsr = 0;
121   state->prng = 0;
122 
123   /* Shift in keybytes in reverse order */
124   uint64_t k = key;
125   for(int i=0; i<6; i++) {
126     state->lfsr <<= 8;
127     state->lfsr |= (k & 0xff);
128     k >>= 8;
129   }
130 }
131 
132 /* Shift UID xor card_nonce into the LFSR without active cipher stream
133  * feedback */
134 static void crypto1_clean_mutual_1(struct _crypto1_state * const state,
135     const uint32_t uid, const uint32_t card_challenge)
136 {
137    const uint32_t IV = uid ^ card_challenge;
138 
139   /* Go through the IV bytes in MSByte first, LSBit first order */
140    mifare_update_word(state, IV, 0);
141 
142    state->prng = card_challenge; /* Load the card's PRNG state into our PRNG */
143    return;
144 }
145 
146 /* Shift in the reader nonce to generate the reader challenge,
147  * then generate the reader response */
148 static void crypto1_clean_mutual_2_reader(struct _crypto1_state * const state,
149     parity_data_t * const reader_response)
150 {
151   /* Unencrypted reader nonce */
152   const uint32_t reader_nonce = ARRAY_TO_UINT32(reader_response);
153 
154   /* Feed the reader nonce into the state and simultaneously encrypt it */
155   for(int i = 3; i >= 0; i--) { /* Same as in mifare_update_word, but with added
          parity */
156     reader_response[3-i] = reader_response[3-i] ^ mifare_update_byte(state,
            (reader_nonce >> (i*8)) & 0xff, 0);
157     reader_response[3-i] ^= mf20(state->lfsr)<<8;
158   }
159 
160   /* Unencrypted reader response */
```

```
161   const uint32_t RR = prng_next(state, 64);
162   UINT32_TO_ARRAY_WITH_PARITY(RR, reader_response+4);
163
164   /* Encrypt the reader response */
165   crypto1_transcrypt(state, reader_response+4, 4);
166 }
167
168 /* Generate the expected card response and compare it to the actual
169  * card response */
170 static int crypto1_clean_mutual_3_reader(struct _crypto1_state * const state,
171     const parity_data_t * const card_response)
172 {
173   const uint32_t TR_is = ARRAY_TO_UINT32(card_response);
174   const uint32_t TR_should = prng_next(state, 96) ^ mifare_update_word(state, 0,
        0);
175
176   return TR_is == TR_should;
177 }
178
179 /* Shift in the reader challenge into the state, generate expected reader
180  * response and compare it to actual reader response. */
181 static int crypto1_clean_mutual_2_card(struct _crypto1_state * const state,
182     const parity_data_t * const reader_response)
183 {
184   /* Reader challenge/Encrypted reader nonce */
185   const uint32_t RC = ARRAY_TO_UINT32(reader_response);
186   /* Encrypted reader response */
187   const uint32_t RR_is = ARRAY_TO_UINT32(reader_response+4);
188
189   /* Shift in reader challenge */
190   const uint32_t keystream = mifare_update_word(state, RC, 1);
191 #ifdef PRINT_CORRECT_READER_CHALLENGE
192   printf("%016LX\n", RC ^ keystream);
193 #else
194   (void)keystream;
195 #endif
196
197   /* Generate expected reader response */
198   const uint32_t RR_should = prng_next(state, 64) ^ mifare_update_word(state, 0,
        0);
199
200   return RR_should == RR_is;
201 }
202
203 /* Output the card response */
204 static void crypto1_clean_mutual_3_card(struct _crypto1_state * const state,
205     parity_data_t * const card_response)
206 {
207   /* Unencrypted tag response */
208   const uint32_t TR = prng_next(state, 96);
209   UINT32_TO_ARRAY_WITH_PARITY(TR, card_response);
210
211   /* Encrypt the response */
212   crypto1_transcrypt(state, card_response, 4);
213 }
```

```
214
215 /* Encrypt or decrypt a number of bytes */
216 static void crypto1_clean_transcrypt_bits(struct _crypto1_state * const state,
217     parity_data_t * const data, const size_t bytes, const size_t bits)
218 {
219   for(int i = 0; i< bytes; i++) {
220     data[i] = data[i] ^ mifare_update_byte(state, 0, 0);
221     data[i] = data[i] ^ (mf20(state->lfsr) << 8);
222   }
223   for(int i = 0; i< bits; i++) {
224     data[bytes] ^= mifare_update(state, 0, 0) << i;
225   }
226 }
227
228 static const struct _crypto1_ops crypto1_clean_ops = {
229     crypto1_clean_init,
230     crypto1_clean_mutual_1,
231     crypto1_clean_mutual_2_reader,
232     crypto1_clean_mutual_3_reader,
233     crypto1_clean_mutual_2_card,
234     crypto1_clean_mutual_3_card,
235     crypto1_clean_transcrypt_bits
236 };
237
238 int _crypto1_new_clean(crypto1_state * const state)
239 {
240   state->ops = &crypto1_clean_ops;
241   return 1;
242 }
```

Listing A.4: `tests.c` – Testcases und Nutzungsbeispiel

```
1  /*
2   * Philips/NXP Mifare Crypto-1 implementation v1.0
3   *
4   * By Karsten Nohl, Henryk Plötz, Sean O’Neil
5   *
6   */
7
8  #include <stdio.h>
9  #include <string.h>
10 #include <ctype.h>
11
12 #include "crypto1.h"
13
14 #define ARRAY_LENGTH(x) (sizeof(x)/sizeof(x[0]))
15 //#define DEBUG_PRINTS
16 #define MAX_PLAINTEXT_CIPHERTEXT_PAIRS 5
17 #define MAX_PLAINTEXT_LEN 18
18
19 /* format is either "xx xx xx xx..."
20  * or "xx b? xx b? xx b?..."
21  * where xx is the byte in hex, b is the parity bit and ? may be anything or
22  * not there at all
23  */
24 typedef char* printable_parity_data_t;
25
26 #define FLAGS_NO_CHECK_PARITY_AUTH 1
27
28 struct plaintext_ciphertext_pair {
29  printable_parity_data_t plaintext;
30  printable_parity_data_t ciphertext;
31  size_t num_bits;
32 };
33
34 struct testcase {
35  uint64_t key;
36  uint32_t uid;
37  uint32_t card_challenge;
38  uint32_t reader_challenge;
39  printable_parity_data_t reader_response;
40  printable_parity_data_t card_response;
41
42  int data_pairs;
43  struct plaintext_ciphertext_pair data[MAX_PLAINTEXT_CIPHERTEXT_PAIRS];
44 };
45
46 #define KEY_1 0xffffffffffffULL
47 #define KEY_2 0xa0a1a2a3a4a5ULL
48
49 #define UID_1 0xB479F7D7
50 #define UID_2 0x8CBA5DD3
51
52 /* These test cases come from sniffed transactions with known key and
53  * contents between an OpenPCD and a Mifare Classic 1k card */
54 const struct testcase testcases[] = {
```

```
55 /*  key, uid, card chal, reader chal,
56     reader resp,
57     card resp, number of test ciphertexts,
58     plaintext/ciphertext pairs ... */
59    {KEY_1, UID_1, 0xF3FBAEED, 0x07C9A995,
60       "7C 1! 74 1 07 1! EB 1 0F 0! 7B 1 D5 0 1B 0!",
61       "3D 1! 0E 1! A0 0! E2 1", 2, {
62       {"30 1 00 1 02 0 a8 0", "65 0! 8D 0! 65 1 1F 0"},
63       {"B4 1 79 0 F7 0 D7 1", "52 0 F6 1 46 0 35 1"}, }
64    },
65    {KEY_1, UID_1, 0x2D4DAAC5, 0x68368F0C,
66       "ED 1 73 1! 6B 0 02 1! 88 1 42 1 5B 0 A4 1!",
67       "A2 1! D4 0! 3C 0! C3 1", 2, {
68       {"30 1 00 1 02 0 a8 0", "5B 0 6F 1 96 1 CF 1 "},
69       {"B4 1 79 0 F7 0 D7 1", "BB 1 FD 1! 82 1 D2 0!"}, }
70    },
71    {KEY_1, UID_2, 0x9347B9F4, 0x3BA73C6D,
72      "E5 1! 0A 1 5B 0 84 1 44 1 E5 1! C1 0 0C 1",
73      "A7 0 A2 1! DA 0 ED 0!", 4, {
74      {"A0 1 01 0 d6 0 a0 1", "E3 0 B6 0 0E 1! A5 1"},
75      {"0a", "00", 4},
76      {"00 1 00 1 00 1 00 1 00 1 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!"},
77      {"0a", "0C", 4}, }
78    },
79    {KEY_2, UID_2, 0x0DF547C9, 0x55414992,
80      "85 0 1E 1 29 1! 49 0 BF 0 44 1 5B 1! EB 1",
81      "A5 0! 86 1! F4 0 37 1!", 2, {
82      {"30 1 04 0 26 0 ee 1", "86 1! E0 1! 1B 0! 9E 0" },
83      {"00 1 00 1 00 1 00 1 00 1 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 58 1! F2 0 F9 1 00 0! A9 0! 5F 0! A5 1 1C 0 95
          0! E7 0! 0D 0 19 0 25 1! F6 0! E1 1 51 0 79 0"}, }
84    },
85 };
86
87 static int get_hex_value(char c)
88 {
89  return ( (c>='a'&&c<='f') ? (10+c-'a') : ( (c>='A'&&c<='F') ? (10+c-'A') : (
       c-'0' ) ) ) & 0xf;
90 }
91
92 static int convert_printable_parity_data(const printable_parity_data_t in,
     parity_data_t *out, size_t *outlen)
93 {
94  size_t l = strlen(in);
95  int outpos = 0;
96  uint16_t byte = 0;
97  enum { FIRST_NIBBLE, SECOND_NIBBLE, PARITY} next = FIRST_NIBBLE;
98  for(int inpos = 0; inpos<l; inpos++) {
99    const char c = in[inpos];
100
101    if(outpos == *outlen) {
102      for(;inpos<l; inpos++)
103        if(isxdigit(in[inpos]))
```

```
104        return 0;
105      return 1;
106    }
107
108    switch(next) {
109    case FIRST_NIBBLE:
110      if(isxdigit(c)) {
111        byte = get_hex_value(c);
112        next = SECOND_NIBBLE;
113      }
114      break;
115    case SECOND_NIBBLE:
116      if(isxdigit(c)) {
117        byte = (byte << 4) | get_hex_value(c);
118        next = PARITY;
119      } else {
120        out[outpos++] = byte;
121        next = FIRST_NIBBLE;
122      }
123      break;
124    case PARITY:
125      if(isxdigit(c)) {
126        if(!isxdigit(in[inpos+1])) {
127          byte = byte | (get_hex_value(c) << 8);
128          out[outpos++] = byte;
129          next = FIRST_NIBBLE;
130        } else {
131          out[outpos++] = byte;
132          byte = get_hex_value(c);
133          next = SECOND_NIBBLE;
134        }
135      }
136      break;
137    }
138  }
139
140  if(next != FIRST_NIBBLE) {
141    if(outpos == *outlen)
142      return 1;
143    else
144      out[outpos++] = byte;
145  }
146
147  *outlen = outpos;
148  return 1;
149 }
150
151 static int compare_parity_data(parity_data_t *a, parity_data_t *b, size_t len,
        size_t num_bits, unsigned int *warnings)
152 {
153   char warned = 0;
154   for(int i=0; i<len; i++) {
155     if(i==len-1 && num_bits > 0) {
156       parity_data_t c = a[i], d = b[i];
157       c &= ~((~0)<<num_bits);
```

```
158         d &= ~((~0)<<num_bits);
159         if(c != d)
160           return 0;
161       } else {
162         if( (a[i]&0xff) != (b[i]&0xff) ) {
163           return 0;
164         } else if(a[i] != b[i]) {
165           if(!warned) {
166             printf(" + Warning: Parity mismatch at offset %i\n", i);
167             if(warnings)
168               (*warnings)++;
169           }
170           warned = 1;
171         }
172       }
173     }
174     return 1;
175   }
176
177   static unsigned int do_test(crypto1_state *state, const struct testcase
          *testcase, unsigned int *warnings, const int flags)
178   {
179     parity_data_t reader_response[8], reader_response_backup[8];
180     parity_data_t card_response[4], card_response_backup[8];
181     parity_data_t plaintext[MAX_PLAINTEXT_LEN], ciphertext[MAX_PLAINTEXT_LEN];
182
183     size_t l = ARRAY_LENGTH(reader_response);
184     if(!convert_printable_parity_data(testcase->reader_response,
            (parity_data_t*)&reader_response, &l)) {
185       printf(" > Internal error 1\n");
186       return 1;
187     }
188
189     l = ARRAY_LENGTH(card_response);
190     if(!convert_printable_parity_data(testcase->card_response,
            (parity_data_t*)&card_response, &l)) {
191       printf(" > Internal error 2\n");
192       return 1;
193     }
194
195     memcpy(reader_response_backup, reader_response, sizeof(reader_response));
196     memcpy(card_response_backup, card_response, sizeof(card_response));
197
198     crypto1_init(state, testcase->key);
199     crypto1_mutual_1(state, testcase->uid, testcase->card_challenge);
200
201     if(!state->is_card) {
202       /* Preload the reader nonce into the reader response */
203       UINT32_TO_ARRAY_WITH_PARITY(testcase->reader_challenge, reader_response);
204     }
205
206     if(!crypto1_mutual_2(state, reader_response)) {
207       printf(" > Mutual 2 failed\n");
208       return 1;
209     }
```

```
210   if(!crypto1_mutual_3(state, card_response)) {
211     printf(" > Mutual 3 failed\n");
212     return 1;
213   }
214 
215   if(!(flags & FLAGS_NO_CHECK_PARITY_AUTH)) {
216     if(!compare_parity_data(reader_response, reader_response_backup,
          ARRAY_LENGTH(reader_response), 0, warnings)) {
217       printf(" > Compare reader response failed\n");
218       return 1;
219     }
220     if(!compare_parity_data(card_response, card_response_backup,
          ARRAY_LENGTH(card_response), 0, warnings)) {
221       printf(" > Compare card response failed\n");
222       return 1;
223     }
224   }
225 
226   for(int i=0; i<testcase->data_pairs; i++) {
227     size_t pl = ARRAY_LENGTH(plaintext), cl = ARRAY_LENGTH(ciphertext);
228     if(!convert_printable_parity_data(testcase->data[i].plaintext,
          (parity_data_t*)&plaintext, &pl)) {
229       printf(" > Internal error 3\n");
230       return 1;
231     }
232     if(!convert_printable_parity_data(testcase->data[i].ciphertext,
          (parity_data_t*)&ciphertext, &cl)) {
233       printf(" > Internal error 4\n");
234       return 1;
235     }
236 
237     if(testcase->data[i].num_bits == 0)
238       crypto1_transcrypt(state, ciphertext, cl);
239     else
240       crypto1_transcrypt_bits(state, ciphertext, cl-1,
            testcase->data[i].num_bits);
241 
242 #ifdef DEBUG_PRINTS
243     for(int j=0; j<cl; j++) {
244       printf("%02X", ciphertext[j]&0xff);
245     }
246     printf("\n");
247 #endif
248 
249     if(!compare_parity_data(ciphertext, plaintext, pl, testcase->data[i].num_bits,
           warnings)) {
250       printf(" > Compare plaintext %i failed: Is ", i);
251       for(int j=0; j<cl; j++) {
252         printf("%02X", ciphertext[j]&0xff);
253       }
254       printf(", should be ");
255       for(int j=0; j<cl; j++) {
256         printf("%02X", plaintext[j]&0xff);
257       }
258       printf("\n");
```

```
259      return 1;
260    }
261  }
262
263  return 0;
264 }
265
266 static unsigned int do_tests(crypto1_state *state, const char *mode, const int
        flags)
267 {
268   unsigned int result = 0;
269   unsigned int warnings = 0;
270   for(int i = 0; i< ARRAY_LENGTH(testcases); i++) {
271     int ret = do_test(state, &testcases[i], &warnings, flags);
272     if(ret > 0) {
273       printf(" + Testcase %i failed in %s mode\n", i, mode);
274     } else {
275       if(warnings > 0)
276         printf(" + Testcase %i produced %i warnings in %s mode\n", i, warnings,
              mode);
277     }
278     result += ret;
279     warnings = 0;
280   }
281   return result;
282 }
283
284 static unsigned int test_one_implementation(enum crypto1_cipher_implementation
        implementation, const int flags)
285 {
286   crypto1_state state;
287   int result = 0;
288
289   if(crypto1_new(&state, 0, implementation)) {
290     result += do_tests(&state, "reader", flags);
291   } else {
292     printf(" + failed to initialize implementation in reader mode\n");
293     result += 1;
294   }
295
296   if(crypto1_new(&state, 1, implementation)) {
297     result += do_tests(&state, "card", flags);
298   } else {
299     printf(" + failed to initialize implementation in card mode\n");
300     result += 1;
301   }
302
303   return result;
304 }
305
306 int main(void)
307 {
308   unsigned int errors = 0;
309
310   printf("Testing clean implementation\n");
```

```
311    errors += test_one_implementation(CRYPTO1_IMPLEMENTATION_CLEAN, 0);
312
313    /* The optimized implementation doesn't correctly handle parity
314     * bits during authentication, simply silence those warnings (and
315     * don't use this implementation for any real hardware) */
316    printf("Testing optimized implementation\n");
317    errors += test_one_implementation(CRYPTO1_IMPLEMENTATION_OPTIMIZED,
           FLAGS_NO_CHECK_PARITY_AUTH);
318
319    printf("======\n");
320    if(errors > 0) {
321      printf("%u errors during test run\n", errors);
322      return 1;
323    } else {
324      printf("0 errors during test run\n");
325      return 0;
326    }
327  }
```

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

[And94] ANDERSON, Ross: A5 (Was: HACKING DIGITAL PHONES). Posting in der Newsgroup sci.crypt. mid:2ts9a0$95r@lyra.csx.cam.ac.uk. Version: 17. Juni 1994. – Verfügbar über Google Groups: http://groups.google.com/group/sci.crypt/msg/ba76615fef32ba32

[Bar] BARRY, Richard: *FreeRTOS*. http://www.freertos.org/. Website

[BC93] BRANDS, Stefan ; CHAUM, David: Distance-bounding protocols (extended abstract). In: *Theory and Application of Cryptographic Techniques*, 1993, S. 344–359

[Ber96] BERGER, Dominik: Die Rolle kontaktloser Chipkarten im Banking. Version: Dezember 1996. http://www.asa.or.at/pdf/asanews199612.pdf. In: *ASA News*. Austrian Smart Card Association, Dezember 1996 (5)

[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

[BGW01] BORISOV, Nikita ; GOLDBERG, Ian ; WAGNER, David: Intercepting Mobile Communications: The Insecurity of 802.11. In: *Proceedings of the Seventh Annual International Conference on Mobile Computing And Networking*, 2001, 180–189

[BS96] BIHAM, Eli ; SHAMIR, Adi: *The next stage of differential fault analysis: How to break completely unknown cryptosystems*. DRAFT. http://www.fit.vutbr.cz/~cvrcek/cards/nextstage.ps. Version: Oktober 1996

[CNO08] COURTOIS, Nicolas T. ; NOHL, Karsten ; O'NEIL, Sean: *Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards*. Cryptology ePrint Archive, Report 2008/166. http://eprint.iacr.org/2008/166. Version: 2008

[Dig08a] DIGITAL SECURITY GROUP, RADBOUD UNIVERSITY NIJMEGEN: *Radboud University Nijmegen allowed to publish Mifare Classic Chip article*. Version: 18. Juli 2008. http://www.sos.cs.ru.nl/applications/rfid/pressrelease-courtdecision.en.html. Pressemitteilung

[Dig08b] DIGITAL SECURITY GROUP, RADBOUD UNIVERSITY NIJMEGEN: *Security Flaw in Mifare Classic*. Version: 12. März 2008. http://www.sos.cs.ru.nl/applications/rfid/pressrelease.en.html. Pressemitteilung

[Fin06] FINKENZELLER, Klaus: *RFID-Handbuch*. 4., aktualisierte und erw. Auflage. Hanser, 2006

[Gru06] GRUNWALD, Lukas: New Attacks against RFID-Systems. In: *Black Hat Briefings USA*, 2006

[Haa08] HAAN, Ferry: *Geheime code van ov-kaart ligt op straat*. Version: 8. Januar 2008. http://www.volkskrant.nl/economie/article492709.ece/%20Geheime_code_van_ov-kaart_ligt_op_straat. Zeitungsartikel

[Han05] HANCKE, Gerhard P.: *A Practical Relay Attack on ISO 14443 Proximity Cards*. http://www.cl.cam.ac.uk/~gh275/relay.pdf. Version: Februar 2005

[Han06] HANCKE, Gerhard P.: Practical Attacks on Proximity Identification Systems (Short Paper). In: *Proceedings of IEEE Symposium on Security and Privacy 2006*, 2006

[HK05] HANCKE, Gerhard P. ; KUHN, Markus G.: An RFID Distance Bounding Protocol. In: *Proceedings of IEEE/Create-Net SecureComm 2005*, 2005

[hug] *hugin – Panorama photo stitcher*. http://hugin.sourceforge.net/. Website

[ISOa] ISO/IEC JTC1/SC17: *ISO/IEC 14443: Identification cards – Contactless integrated circuit(s) cards – Proximity cards*. Internationaler Standard,

[ISOb] ISO/IEC JTC1/SC17: *ISO/IEC 15693: Vicinity cards*. Internationaler Standard,

[ISOc] ISO/IEC JTC1/SC17: *ISO/IEC 7810: Identification cards – Physical characteristics*. Internationaler Standard,

[ISOd] ISO/IEC JTC1/SC17: *ISO/IEC 7816: Identification cards – Integrated circuit cards with contact*. Internationaler Standard,

[ISOe] ISO/IEC JTC1/SC27: *ISO/IEC 9798: Information technology – Security techniques – Entity authentication*. Internationaler Standard,

[KGHG08] KONING GANS, Gerhard de ; HOEPMAN, Jaap-Henk ; GARCIA, Flavio D.: *A Practical Attack on the MIFARE Classic*. http://aps.arxiv.org/abs/0803.2285. Version: 2008

[KNP08] KRISSLER, Jan ; NOHL, Karsten ; PLÖTZ, Henryk: Chiptease. In: *c't magazin für computer technik* (2008), Nr. 8, S. 80–85

[KW05] KFIR, Ziv ; WOOL, Avishai: *Picking Virtual Pockets using Relay Attacks on Contactless Smartcard Systems*. Cryptology ePrint Archive, Report 2005/052. http://eprint.iacr.org/2005/052. Version: Februar 2005

[Lib08] LIBBENGA, Jan: *Dutch boffins clone Oyster card*. Version: 23. Juni 2008. http://www.theregister.co.uk/2008/06/23/dutch_clone_oyster_card/. Online-Bericht

[Mif06] *Wie alt müsste ein Mifare Hacker eigentlich werden?* Version: 2006. http://smartnfc.com/index.php?option=com_content&task=view&id=109&Itemid=48. Website

[NEsP08] Nohl, Karsten ; Evans, David ; starbug ; Plötz, Henryk: Reverse-Engineering a Cryptographic RFID Tag. In: *17th USENIX Security Symposium*, USENIX, 2008

[Noh08] Nohl, Karsten: *Cryptanalysis of Crypto-1*. http://www.cs.virginia.edu/~kn5f/pdf/Mifare.Cryptanalysis.pdf. Version: März 2008

[NP07] Nohl, Karsten ; Plötz, Henryk: Mifare – Little Security, Despite Obscurity. In: *"24C3: Volldampf voraus", 24th Chaos Communication Congress*, Chaos Computer Club, Dezember 2007. – Folien und Mitschnitte online verfügbar unter http://events.ccc.de/congress/2007/Fahrplan/events/2378.en.html

[NXP] NXP Semiconductors Austria GmbH: *MIFARE Milestones*. http://mifare.net/about/milestones.htm. Website

[NXP08a] NXP Semiconductors: *MF01ICU1 – Functional specification MIFARE Ultralight*. Version: 4. Februar 2008. http://www.nxp.com/acrobat/other/identification/M028634_MF0ICU1_Functional_Spec_V3.4.pdf. Datenblatt. – Version 3.4

[NXP08b] NXP Semiconductors: *NXP introduces new security and performance benchmark with MIFARE Plus*. Version: März 2008. http://www.nxp.com/news/content/file_1418.html. Pressemitteilung

[Oss06] Ossmann, Martin: Experimenteller RFID-Reader. In: *elektor electronics worldwide* (2006), September, Nr. 426, S. 36–41

[Pat99] Patrizio, Andy: *DVD Piracy: It Can Be Done*. Version: 1. November 1999. http://www.wired.com/science/discoveries/news/1999/11/32249. Online-Bericht

[RGC+06] Rieback, Melanie ; Gaydadjiev, Georgi ; Crispo, Bruno ; Hofman, Rutger ; Tanenbaum, Andrew: A Platform for RFID Security and Privacy Administration. In: *Proc. USENIX/SAGE Large Installation System Administration conference*. Washington DC, USA, December 2006, 89–102

[scn94] Siemens License MIFARE. Version: März 1994. http://www.smartcard.co.uk/members/newsletters/1994/mar94.pdf. In: *Smart Card News*. Smart Card News Ltd., März 1994, Seite 44

[sil] *The Silicon Zoo – A collection of logic cells found in silicon chips*. http://siliconzoo.org/. Website

[SPMR06] Stigge, Martin ; Plötz, Henryk ; Müller, Wolf ; Redlich, Jens-Peter: *Reversing CRC–Theory and Practice*. http://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-05/SAR-PR-2006-05_.pdf. Version: 2006

[Ver08] VERDULT, Roel: *Proof of concept, cloning the OV-Chip card*. http://www.cs.ru.nl/~flaviog/OV-Chip.pdf. Version: 2008

[Wel] WELTE, Harald: *librfid – A Free Software RFID stack*. http://openmrtd.org/projects/librfid/. Website

[Wes] WESTHUES, Jonathan: *Proxmark3 – A Test Instrument for HF/LF RFID*. http://cq.cx/proxmark3.pl. Website

[Wik08] WIKIPEDIA: *Linear feedback shift register — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Linear_feedback_shift_register&oldid=230168303. Version: 2008. – [abgerufen 8-August-2008]

[Win08] WINTER, Brenno de: *Onderzoekers kraken versleuteling rfid-chips*. Online-Bericht. http://webwereld.nl/articles/49231/onderzoekers-kraken-versleuteling-rfid-chips.html. Version: 2. Januar 2008

Alle URLs wurden zuletzt am 18.08.2008 geprüft.