Hardware Supported Man-in-the-Middle Attack Interfering with Smartcard Communication Protocols

Max Hoffmann

Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Bastian Richter
Abstract

Smartcards are well known to everyone. Whether it is an ID-card, a bus-ticket, a library card or an EC-card, we receive multiple smartcards in our life. Because of their comfortable size and versatile functionality, they tend to make things easier.

However, because of this small size, their capabilities are limited. Security measures, especially cryptographic operations, like encrypting a plaintext or generating a digital signature, require a lot of computational power and therefore are only used when it is really necessary on smartcards. Since the employed cryptographic algorithms are well known and can be regarded as secure, another possible target for attackers is interfering with the protocol messages. Inexperienced programmers may tend to skip encryption or verification of messages of important content, because of the high computational time. Protocol designers might put less effort in the security of their creations, since they regard the mere application of a cryptographic operation as secure enough. With such flaws, we might be able to attack the protocol messages with a Man-in-the-Middle (MitM) attack.

In this work we build and implement a “MitM Smartcard Reader”, a device which relays the communication between a smartcard and a terminal to custom software that may monitor or even manipulate the messages before sending them back to their original target.

With this hardware we try to attack the read-out process of various smartcards, as well as the youth protection mechanism of the “GeldKarte”.

We show that our device can be used to successfully perform eavesdropping and MitM attacks on smartcard communication in general, but that the youth protection of the “GeldKarte” is well secured against MitM attacks.
Statutory Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

I affirm that the digital version is identical to the submitted written version. I hereby agree that the digital version of this submission is used for plagiarism assessment.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.


__________________________
Max Hoffmann
Contents

1 Introduction ......................................................... 1
   1.1 Motivation ...................................................... 1
   1.2 Related Work .................................................. 1
   1.3 Contribution .................................................. 2
   1.4 Organization of this Thesis ................................. 2

2 Technical Background ........................................... 3
   2.1 Man-in-the-Middle Attacks ................................... 3
   2.2 Protocol Attacks .............................................. 3
   2.3 Smartcards ..................................................... 3
      2.3.1 Hardware .................................................. 4
      2.3.2 I/O Data Transfer ......................................... 5
      2.3.3 Message Sequence ......................................... 6
      2.3.4 Answer to Reset ........................................... 7
      2.3.5 Protocol Parameter Selection .......................... 8
      2.3.6 Communication Speed Configuration .................... 8
      2.3.7 The Protocols T=0 and T=1 ............................ 9
      2.3.8 Application Protocol Data Units ....................... 11
      2.3.9 File Structure ............................................ 11
   2.4 The “GeldKarte” .............................................. 12
   2.5 BCD Encoding .................................................. 13
   2.6 Cryptographic Nonce .......................................... 13

3 Software Used ..................................................... 15

4 MitM Smartcard Reader ........................................... 17
   4.1 Hardware ....................................................... 18
      4.1.1 Layout ..................................................... 18
   4.2 Firmware ....................................................... 19
      4.2.1 Byte Transmission ......................................... 20
      4.2.2 USB Communication ....................................... 20
      4.2.3 Initialization Sequence ................................... 21
      4.2.4 Card Reset ................................................ 22
      4.2.5 Terminal Disconnect ...................................... 22
      4.2.6 Message Routing .......................................... 23
      4.2.7 Device Configuration ..................................... 24
Contents

4.3 Software .................................................. 25
   4.3.1 C++ Class ............................................. 25
   4.3.2 MitM Tool ............................................. 27
4.4 Limitations and Known Issues .................................. 28
4.5 Possible Timing Problems ..................................... 28

5 Attacking Smartcards ........................................... 31
   5.1 Manipulating Data Fields ................................ 31
   5.2 Attacking the Youth Protection of the “GeldKarte” ........ 34
      5.2.1 Online Simulation .................................. 34
      5.2.2 Cigarette Vending Machine ......................... 35

6 Conclusion ...................................................... 39
   6.1 Summary .................................................. 39
   6.2 Future Work ............................................. 39

A Acronyms ......................................................... 41

List of Figures .................................................... 43

List of Tables ..................................................... 45

List of Listings .................................................. 47

Bibliography ...................................................... 49
1 Introduction

MitM attacks are regarded as powerful but difficult attacks, since the establishment of such an attack is generally a hard task. Hijacking the communication of two or more partners, which can be located around the world, is already a technical challenge. With smartcards the scenario completely changes. They are in direct contact with, or, in case of contactless smartcards, really close to their communication partner and are not physically protected against such attacks.

1.1 Motivation

Most normal users do not even know that they are holding a computer in their hands, when they take out their EC-cards. Furthermore they do not think about the security the cards have to provide. It is interesting to examine, whether programmers are looking at smartcards from the same point of view. Is transmitted data protected? Would it be, for example, possible to just change a transmitted account number and enter a hotel room one never paid for? To be able to tackle these questions in the future, we want to create a custom hardware for MitM attacks on smartcards.

1.2 Related Work

In June 2010, Omar S. Choudary presented his master’s thesis in which he built a smart-card MitM device, the “SmartCard Detective” [1]. He implemented several applications for this device to perform attacks on implementations of the Europay International, MasterCard and VISA (EMV) framework and to protect users against this type of attack. The main goal of his work was to protect users against attacks, by providing a way to double check their payments. He added a confirmation step to the payment process, where the user has to approve the payment amount. Users which are attacked, would see the manipulated amount and are able to cancel the transaction.

Early 2015, Martin Land analyzed the payment and the charge protocol of the “GeldKarte” in his bachelor’s thesis [2]. In his work he monitored the communication between a smartcard and a USB card reader with a USB sniffer. He showed that the two inspected protocols are theoretically well secured against attackers, but he did not have access to MitM hardware and therefore could not try to attack those procedures.

In contrast to these works, we create a custom smartcard reader with MitM functionality, but our device is not intended to be used as a stand-alone defender or attack-showcase
like the “SmartCard Detective”. It enables us to monitor and manipulate the actual byte transmissions of a smartcard and a terminal via an external software. Our device does not have to be reprogrammed and, at a price of approximately €25, it is a lot cheaper than the “SmartCard Detective”, which costs nearly £100 (about €136).

1.3 Contribution

We contribute an affordable device for monitoring and manipulating the Application Protocol Data Units (APDUs) exchanged between a smartcard and a terminal. Additionally we provide a C++ class to enable programmers to implement their custom MitM attacks, using our hardware. Furthermore, we use our device to analyze the youth protection mechanism of the “GeldKarte”.

1.4 Organization of this Thesis

We start by providing some technical background. We take a look at smartcards and their communication procedure, Man-in-the-Middle (MitM) attacks, and the “GeldKarte”. Then we present the “MitM Smartcard Reader” we build in this work and outline the workflow of the implementation. We furthermore present the device’s capabilities by manipulating the read-out process of smartcards and by trying to attack the youth protection of the “GeldKarte”. In the end we give a conclusion and offer perspectives for further research.
2 Technical Background

Before we present the “MitM Smartcard Reader” we build in this work, we provide background information about MitM attacks, smartcards and the “GeldKarte” application. We also explain BCD encoding and cryptographic nonces.

2.1 Man-in-the-Middle Attacks

In this work, we try to manipulate messages, which are exchanged between a smartcard and a terminal. This attack is called a Man-in-the-Middle (MitM) attack. In a MitM scenario, two or more communication partners think that they directly communicate with each other. However, an attacker has hijacked the session instead. All transmitted data passes the attacker, who can chose to pass-through, manipulate or drop messages. Without integrity protecting mechanisms, for example a digital signature or a Message Authentication Code (MAC), any data is vulnerable and an ongoing attack cannot be easily detected.

2.2 Protocol Attacks

The attacks performed in this work are also protocol attacks: In a protocol, communication partners exchange messages according to a mutually known structure. A protocol attack regards the communication partners as “black boxes”. What happens inside these black boxes is unknown and unimportant to the attacker. He can just see and work with the output of the black boxes, in our case the APDU which are sent over the I/O line of the smartcard. Any kind of cryptographic operation is regarded as secure in this context, so the attacker’s goal is not to break any encryption schemes or other security measures in a mathematical way.

2.3 Smartcards

Smartcards, also known as chipcards or Integrated Circuit Cards (ICC), are a cheap way to provide pocket-size computers to customers. They are mostly used for identification, access control, and electronic payment. To the user they are robust and come in a comfortable format, which fits in every pocket. Due to their physical size, their computational power, as well as their memory, is limited.

There are essentially two types of smartcards: Contactless and contact-based smartcards. In this work, we focus on contact-based smartcards. Therefore, whenever we use the term smartcard in the following, we always refer to the contact-based variant. The
communication partner of a smartcard, whether it is a door access controller, a vending machine or a card reader, is called a terminal. Smartcards are standardized in the ISO 7816 standard [3].

2.3.1 Hardware

Figure 2.1 shows a typical smartcard. It consists of a plastic plate, which holds an eight-pin microcontroller. In general only six of these eight pins are used and the remaining two are reserved for future use. On special test or debug cards these pins sometimes have a functionality for the particular use-case. Figure 2.2 shows the pin layout of a smartcard. For detailed explanations of the pins’ capabilities and limits refer to [3].

- **Pin 1 - VCC**: The terminal has to provide a voltage of either 5.0V, 3.0V or 1.8V to this pin in order to power the smartcard.

- **Pin 2 - RST**: The reset pin can be used to perform a reset of the smartcard. There are two types of smartcard resets: A card may respond to the low state of the RST pin. In this case it features an active low reset. A card that responds to the high state of the RST pin is a smartcard with an active high reset.

- **Pin 3 - CLK**: If the smartcard has no internal clock, the terminal has to provide a clock signal in the range of 3Mhz to 5Mhz to this pin.

- **Pin 4 - RFU**: Pin 4 is one of the two pins that are reserved for future use.

- **Pin 5 - GND**: The terminal has to provide a ground connection to this pin.

- **Pin 6 - VPP**: A card may require additional voltage for programming or to erase and write to non-volatile memory. In this case the terminal has to provide this voltage on pin 6.

- **Pin 7 - I/O**: The communication between the card and the terminal is transmitted over this pin. Section 2.3.2 gives detailed information regarding smartcard communication.

- **Pin 8 - RFU**: Pin 8 is the second pin that is reserved for future use.
Since there is only one I/O line, the communication between a smartcard and a terminal is performed in a half-duplex fashion: Only one partner can be transmitting at a time, while the other partner has to wait for an ongoing transmission to end before he can send data himself. In the next section, we explain the data transmission over the I/O line in detail.

### 2.3.2 I/O Data Transfer

As mentioned in the previous section, a smartcard provides only one pin for communication, the I/O pin. Because of this, none of the two communication partners can transmit data at the same time. There is always at least one partner in reception mode. This is called half-duplex communication. Byte transmission over the I/O line is specified in ISO 7816 [3]:

The transmission is timed using Elementary Time Units (ETUs). An ETU initially equals 372 divided by the card’s clock frequency (initial ETU). It can be changed by the smartcard’s Answer to Reset (ATR), which we will discuss in Section 2.3.4.

In idle state, the I/O line is kept on high level. A byte transmission is started by pulling the I/O line low for one ETU. This is the start bit. After that the data bits are transmitted from Least Significant Bit (LSB) to Most Significant Bit (MSB), each for the duration of one ETU, where a physical high represents a logical one. This is called direct convention. The ATR may change the convention to indirect convention. In this case, the bits are transmitted from MSB to LSB and a physical high represents a logical zero. In both cases, the ninth transmitted bit is a parity bit, such that the Exclusive OR (XOR) over all transmitted bits is zero (even parity). The bit transmission is followed by a guard time of two ETUs. This process is also shown in Figure 2.3. The receiver may signal a parity error during the guard time, by pulling the I/O line low.
The transmitter is then expected to repeat the corrupted byte.

Figure 2.3: Byte transmission as specified by ISO 7816.

2.3.3 Message Sequence

![Message Sequence Diagram]

A smartcard has to establish a session with a terminal before application data can be exchanged. Figure 2.4 shows the sequence of messages which are exchanged after a smartcard is reset. When a reset is performed by the terminal, the smartcard transmits an Answer to Reset (ATR). This ATR contains configuration data for the rest of the session. We discuss the ATR in detail in Section 2.3.4. The parameters transmitted in the ATR may be changed by the terminal in a Protocol Parameter Selection (PPS) request message. The smartcard replies with a Protocol Parameter Selection (PPS) response message. We give more detail about the PPS in Section 2.3.5. After the exchange of the ATR and, optionally, the PPS, the parameters are applied and the session is established. The terminal may now transmit application data in form of Application Protocol Data Units (APDUs). Therefore the terminal sends a command APDU and the smartcard replies with a response APDU. We explain APDU in Section 2.3.8.
2.3 Smartcards

2.3.4 Answer to Reset

A smartcard can be reset by the terminal via the reset pin as described in Section 2.3.1. A terminal always resets a smartcard when it is plugged in. The software on the terminal may also cause it to reset the smartcard if required. As soon as a smartcard is reset, it sends an Answer to Reset (ATR) to the terminal. This ATR consists of up to 33 bytes of configuration data for the rest of the session. An ATR follows the structure shown in Figure 2.5:

```
TS | T0 | TA1 | TB1 | TC1 | TD1 | TA2 | ... | TD3 | HistoricalBytes | TCK
```

Figure 2.5: Structure of an ATR

The ATR characters encode configuration values for the rest of the session. The TDn bytes always encode the availability of the next TA(n+1) to TD(n+1) bytes in their upper nibble: If bit 8 is one, TD(n+1) is transmitted, if bit 7 is one, TC(n+1) is transmitted, and so on. T0 encodes this for TA1 to TD1. Because of that, the expected length of an ATR can be calculated in parallel to the reception of the bytes.

We only discuss the most important bytes for our attack here in detail. For an explanation of all ATR bytes, refer to ISO 7816 [3].

- **TS:** The mandatory TS character is the initial ATR character and is either 0x3B or 0x3F. 0x3B indicates transmission in direct convention, whereas 0x3F indicates indirect convention. Both transmission conventions are described in Section 2.3.2.

- **T0:** T0 is the second character and also mandatory. It encodes the availability of TA1 (in bit 5) to TD1 (in bit 8) in the upper nibble. The lower nibble holds the number of historical bytes at the end of the ATR.

- **TA1:** Encodes parameters for the communication speed in its nibbles. The meaning of the nibbles is explained in detail in Section 2.3.6. When TA1 is not available, the communication will take place with the default speed, using the initial ETU. Any other than the default value will result in a PPS_request message from the terminal, which we discuss in Section 2.3.5.

- **TD1:** Encodes the availability of TA2 to TD2 in its upper nibble. More important to us, it encodes the primarily offered communication protocol $T$ in the lower nibble. We discuss the two most common protocols $T = 0$ and $T = 1$ in Section 2.3.7.

- **TCK:** The TCK character is sent, when $T = 0$ is not the only offered protocol. It is the XOR checksum over all previous ATR bytes, excluding the TS byte.

After the ATR is received by the terminal it may change some of the configuration parameters in a PPS message.
2.3.5 Protocol Parameter Selection

Upon the reception of an ATR, a terminal may send a PPS request to reconfigure or acknowledge the parameters received by the card. The PPS message contents are specified in ISO 7816 [3]. It consists of up to six bytes:

- **PPSS**: The first byte is always 0xFF and indicates the start of the PPS.

- **PPS0**: Bit 8 is marked as Reserved for Future Use (RFU). Bits 5 to 7 of PPS0 encode the availability of PPS1 to PPS3, with a one indicating that the byte will be sent. The lower nibble encodes the selected protocol for future transmissions.

- **PPS1**: This byte mirrors TA1 from the ATR or changes the values. TA1 encodes the communication speed parameters.

- **PPS2**: If $T = 15$ is selected, PPS2 mirrors or alters TB1.

- **PPS3**: This byte is marked as RFU.

- **PCK**: The last byte is a checksum such that the XOR over all PPS bytes equals zero.

The smartcard will send a PPS response back to the terminal to acknowledge the parameters. A PPS response message contains the same bytes as the PPS request message. If it does not, the terminal shall reset or reject the card. When the PPS was acknowledged correctly, the communication will take place with the agreed parameters. A PPS is always triggered if the TA1 character is transmitted and does not equal 0x11. This indicates a change in the communication speed, as we explain in the next Section.

2.3.6 Communication Speed Configuration

The TA1 character of the ATR controls the communication speed. The upper nibble determines the clock rate conversion factor $F$, as well as the maximum clock frequency for the card, and the lower nibble sets the bit rate adjustment factor $D$. The transmitted values are indexes to select the actual values for $F$ and $D$ from Table 2.1 and 2.2. Some of the values are Reserved for Future Use (RFU). The tables are specified in ISO 7816 [3]. After the transmission of the ATR and, optionally, of the PPS, the new ETU (working ETU) is computed as $\frac{F}{D \cdot f_{\text{card}}}$. If the TA1 character has the value 0x11, this leads to the formula $\text{ETU} = \frac{372}{f_{\text{card}}}$, which is exactly the initial ETU and a maximum card clock frequency of 5MHz, which is the default maximum value at initiation. Therefore the
value 0x11 indicates default parameters and does not trigger a PPS.

<table>
<thead>
<tr>
<th>Upper Nibble</th>
<th>F</th>
<th>$f_{\text{max}}$ (in MHz)</th>
<th>Lower Nibble</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>372</td>
<td>4</td>
<td>0000</td>
<td>RFU</td>
</tr>
<tr>
<td>0001</td>
<td>372</td>
<td>5</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>558</td>
<td>6</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>744</td>
<td>8</td>
<td>0011</td>
<td>4</td>
</tr>
<tr>
<td>0100</td>
<td>1116</td>
<td>12</td>
<td>0100</td>
<td>8</td>
</tr>
<tr>
<td>0101</td>
<td>1488</td>
<td>16</td>
<td>0101</td>
<td>16</td>
</tr>
<tr>
<td>0110</td>
<td>1860</td>
<td>20</td>
<td>0110</td>
<td>32</td>
</tr>
<tr>
<td>0111</td>
<td>RFU</td>
<td>-</td>
<td>0111</td>
<td>64</td>
</tr>
<tr>
<td>1000</td>
<td>RFU</td>
<td>-</td>
<td>1000</td>
<td>12</td>
</tr>
<tr>
<td>1001</td>
<td>512</td>
<td>5</td>
<td>1001</td>
<td>20</td>
</tr>
<tr>
<td>1010</td>
<td>768</td>
<td>7.5</td>
<td>1010</td>
<td>RFU</td>
</tr>
<tr>
<td>1011</td>
<td>1024</td>
<td>10</td>
<td>1011</td>
<td>RFU</td>
</tr>
<tr>
<td>1100</td>
<td>1536</td>
<td>15</td>
<td>1100</td>
<td>RFU</td>
</tr>
<tr>
<td>1101</td>
<td>2048</td>
<td>20</td>
<td>1101</td>
<td>RFU</td>
</tr>
<tr>
<td>1110</td>
<td>RFU</td>
<td>-</td>
<td>1110</td>
<td>RFU</td>
</tr>
<tr>
<td>1111</td>
<td>RFU</td>
<td>-</td>
<td>1111</td>
<td>RFU</td>
</tr>
</tbody>
</table>

Table 2.1: The table for the clock rate conversion factor $F$.

Table 2.2: The table for the bit rate adjustment factor $D$.

### 2.3.7 The Protocols $T=0$ and $T=1$

Once the ATR and PPS messages are exchanged, the terminal can communicate with the smartcard, using a transmission protocol. The protocols $T = 0$ and $T = 1$ are supported by most smartcards. With the selected transmission protocol Application Protocol Data Units (APDUs) are exchanged. We discuss APDUs in Section 2.3.8.

The $T = 0$ protocol is a character transmission protocol, therefore the smallest transmittable unit is a byte. In Figure 2.7 the message exchange in the $T = 0$ protocol can be seen. The terminal sends a command header, which holds the header bytes of an APDU and then receives a procedure byte from the smartcard. Upon reception, data, equivalent to the data field of an APDU, is transmitted from the terminal to the card or from the card to the terminal, depending on the command. In the end a two byte status word is sent by the smartcard.

If one tries to attack messages exchanged between a smartcard and a terminal, both using the $T = 0$ protocol, with a MitM attack, one would have to attack the command header and the data separately, as they are transmitted in different messages.

The $T = 1$ protocol is a block transmission protocol. The terminal sends a command block and the smartcard will answer with a response block as shown in Figure 2.8.
transmits blocks of a fixed format, called a frame, shown in Figure 2.6. The node address (NAD), protocol control byte (PCB), data length (LEN) and error detection (LRC) bytes are mandatory. NAD and PCB encode transmission header values. LEN holds the length of the information field (INF). The value 0xFF is reserved, so the maximum length of the INF field is 254 bytes. The INF field itself is optional, depending on the block type, and contains a command or response data. The EDC field is 1 byte long, if an XOR checksum is used and 2 bytes long if a Cyclic Redundancy Check (CRC) checksum is used. Since the smartcards we experiment with all use the XOR checksum, we regard this field as 1 byte long in the following.

\[
\text{NAD (1 byte) | PCB (1 byte) | LEN (1 byte) | INF (0 - 254 bytes) | EDC (1 or 2 byte)}
\]

Figure 2.6: Structure of a \( T = 1 \) frame.

There are three types of blocks, I-blocks, R-blocks, and S-blocks. I-blocks contain the INF field and transmit a command or a response in form of an APDU. R-blocks do not have the INF field and are transmitted for acknowledgement of long I-blocks. S-blocks contain control messages for the protocol. They do not have the INF field as well. Therefore the exact length of R- and S-blocks is 4 bytes. The maximum length of an I-block is 258 bytes, as there are 4 mandatory bytes and up to 254 data bytes. The minimum length of an APDU is 4 bytes, therefore, together with the 4 mandatory bytes, an I-block is at least 8 bytes long. Attacking messages exchanged between a smartcard and a terminal, both using the \( T = 1 \) protocol, with a MitM attack is not difficult. One would ignore all messages with 4 bytes as those are R- and S-blocks and would analyze the I-blocks’ INF field.

As it is more comfortable to attack the \( T = 1 \) communication, since a command, just like the response, is transmitted in a single message, we concentrate on this protocol in our work and aim on using it in our device.

\[
\begin{array}{c|c}
\text{Smartcard} & \text{Terminal} \\
\hline
\text{Command Header} & \\
\text{Procedure Byte} & \\
\text{Data} & \\
\text{Status Word} & \\
\end{array}
\]

Figure 2.7: The message sequence in the \( T = 0 \) protocol. The Data message is transmitted only in one direction, depending on the Command Header.
2.3 Smartcards

2.3.8 Application Protocol Data Units

The information field (INF) of a $T = 1$ I-block holds an Application Protocol Data Unit (APDU). In $T = 0$ the APDU is split in a command header and the data. Regardless which transmission protocol is used, the terminal will send a command-APDU and the smartcard will reply with a response-APDU.

![Figure 2.8: The message sequence in the $T = 1$ protocol.](image)

CLA | INS | P1 | P2 | Lc | Data | Le

Figure 2.9: Structure of a command-APDU.

Figure 2.9 shows the structure of a command-APDU. The class byte (CLA) contains the command type. The instruction byte (INS) encodes the actual instruction, which operates with the parameters P1 and P2 on the data bytes. Lc holds the length of the data field and Le holds the number of bytes which shall be returned in the response-APDU. The structure of such a response is shown in Figure 2.10.

Data | SW1 | SW2

Figure 2.10: Structure of a response-APDU.

The data field has the length of at most Le bytes. SW1 is concatenated with SW2 and forms a status word, which is essentially a response code to the terminal. The most common status word is $0x9000$ which indicates no error.

2.3.9 File Structure

Smartcards can feature a variety of applications. Those applications are organized as files and folders on the card, as shown in Figure 2.11. The structure can be compared to the Linux file systems:

The root is a Master File (MF). This MF holds references to Elementary Files (EFs) and Dedicated Files (DFs). While an EF can be compared to an actual file, as it can only hold data, a DF can be compared to a folder, as it can hold both, EFs and DFs. Each application usually has its own DF and may employ more DFs to structure its data.
Typically both, EFs and DFs, are referred to as “files” and the exact type is prepend to the name. Examples for this naming convention are DF_BOERSE and EF_BETRAG. The files are protected by access conditions, which are checked on each read or write access. The “GeldKarte” app we take a look at, for example, has no read out protection, except for the files which hold cryptographic key material, but denies all write accesses.

One can access the files with standardized APDUs which we briefly explain in the following. On reset, the MF is selected. With select APDUs other EFs or DFs can be selected. This may be done by traversing the file tree, or by directly using a file identifier. The select command is useful to check whether a file is available on a smartcard, too, as only a successful selection of an available file is acknowledged with the status word 0x9000 by the smartcard. A select APDU has the instruction byte (INS) 0xA4. Using a read APDU which has the INS byte 0xB2, results in a response APDU that holds the file’s requested content, if the access control permitted the read operation. A write APDU will, if permitted by the access control, write the data to the file on the smartcard. Read and write commands can be executed on the selected file or, by using a short file identifier, directly on a specific file. Write APDUs have the INS byte 0xD2.

2.4 The “GeldKarte”

“GeldKarte” is a set of applications for smartcards which features a lot of functionality for electronic payment. It adds the DFs DF_BOERSE, DF_MARKTPLATZ, and DF_FAHRSCHEIN to a smartcard. A subset of its features is cashless payment of small amounts (up to €20) without pin protection, youth protection, electronic ticket management, and timed access control. The “GeldKarte” application can be charged with up to €200 and is used as an electronic wallet. The youth protection is one of the most common use-cases in Germany. It ensures that the original owner of the card is at least eighteen years old. This youth protection is implemented as a “Jugendschutzmerkmal”. If that element is available on the smartcard, this indicates that its owner is an adult.
To ensure confidentiality, the “GeldKarte” uses the Data Encryption Standard (DES) algorithm in the triple-DES variant. The algorithm is used for encryption and to compute a MAC over sensitive data. The MAC can be either eight or sixteen bytes long. The document “Einführung und Überblick Elektronischer Fahrschein und Marktplatz” specifies the protocols and the encoding of the files of the “GeldKarte”.

2.5 **BCD Encoding**

Decimal numbers are sometimes stored in Binary Coded Decimal (BCD) format. In BCD form, each digit is encoded in its binary representation, using 4 bits. Since one byte has 8 bit, two digits can be encoded in a single byte. In hexadecimal representation, each digit is resembled by exactly 4 bit. Therefore the original digits can be read directly in the hexadecimal representation.

A hexadecimal digit can encode the values 0 to 15, though, and a decimal digit only has the values 0 to 9. In BCD format the values of 10 to 15 are unused. This waste of memory is a reason for the rare employment of the BCD format. Two examples of the format can be seen in Figure 2.12.

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hexadecimal Representation</th>
<th>BCD Formatted</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0x0F</td>
<td>0x15</td>
</tr>
<tr>
<td>287</td>
<td>0x01 0x1F</td>
<td>0x02 0x87</td>
</tr>
</tbody>
</table>

Figure 2.12: **BCD** encoding of decimal values. The digits of a BCD formatted value are readable like their decimal representation.

2.6 **Cryptographic Nonce**

For cryptographic operations, random numbers are often mandatory. The term cryptographic nonce or “number only used once”, refers to a random number that is never used again and has not already been used for the current operation. Nonces are typically used in challenge-response protocols or as a salt to prevent dictionary attacks.
3 Software Used

To read out smartcards, or to simulate an age verification process, some tools are required which we list below.

- **Smartcard Analysis**
  These tools are used to analyze a smartcard and to simulate various processes with the “GeldKarte”.
  - **Smart Card ATR Parsing** [5]
    This website helps understanding smartcard ATRs and facilitates verifying our own results when manipulating a smartcard’s ATR.
  - **Chipcardmaster** [6]
    To read a smartcard and to get a graphical representation of its values we use the tool Chipcardmaster by Olaf Jacobsen. We work with version 7.11.
  - **S-Chip AddOn** [7]
    With the S-Chip AddOn by the “Deutscher Sparkassen Verlag GmbH” we can simulate the age verification process and make an electronic test payment of €0.01 with a “GeldKarte” online. Version 2.0.3.1 was installed.

- **Programming**
  These tools are used to implement the firmware for our “MitM Smartcard Reader” and the software it communicates with.
  - **Atmel Studio** [8]
    The firmware of our reader was created in Atmel Studio 6.2.1563.
  - **Cygwin with GNU Make and GCC** [9]
    The software for our reader can be compiled using GNU make and the GCC g++ compiler in a Cygwin32 environment. We used version 4.1 of GNU Make and version 4.9.3 of the GCC.
4 MitM Smartcard Reader

In order to attack the smartcard communication, we construct a device which enables MitM attacks on the APDUs exchanged between a smartcard and a terminal. It is able to route any incoming transmission to an external software, running on a USB-connected PC. This software may arbitrarily manipulate the APDUs before sending it back to the device. This will then pass the data to the original target. For simple eavesdropping, the MitM functionality can be deactivated. In this case the device will transmit the messages to the software and to the original target at the same time. Our device, the “MitM Smartcard Reader”, can be seen in Figure 4.1. Our design principles were to keep the costs low and to facilitate implementing attacks, by circumventing restrictions for the user.

Figure 4.1: The “MitM Smartcard Reader” we build in this work.
4.1 Hardware

Before we describe the firmware on our system, we present the layout of our device. We do not go into detail about every transistor we use, but explain the most important hardware parts and why we chose them.

4.1.1 Layout

A schematic of our device can be seen in Figure 4.2. In the following we present the thoughts behind the core components of our layout and explain the indexes of the schematic.

We implement the firmware on a cheap and durable Atmel xmega32-A4U microcontroller (index 1). Clocked by a 16MHz external quartz, which gets internally amplified, the xmega runs at 32MHz.

A standard smartcard slot (index 2) was used to hold the smartcard and its connections are wired directly to the xmega. For easier oscilloscope access, to examine timing issues during the implementation process, additional pins were placed on the board (index 3). Furthermore the I/O line of the smartcard was also output to an SMA socket (index 4). A second set of pins was placed on the upper edge of the board (index 5). An “EMSEC Smartcard Adapter” can be connected there, as shown in Figure 4.3 and the I/O line of this adapter is output to another SMA socket (index 6). Using the two SMA sockets, the communication between our device and the smartcard, and between our device and the terminal can be easily monitored on an oscilloscope.

The “MitM Smartcard Reader” is powered by a USB-connection (index 7). This connection is also used to exchange data with the software. To add additional use-cases to our device, we put an additional SMA socket onto the board (index 8). The applied voltage of this hub is kept at logical zero until a transmission to the card is finished. It then becomes logical one, to provide a precise timing trigger for future side-channel-attacks.

Figure 4.2: An abstract view on the layout of the “MitM Smartcard Reader”.
4.2 Firmware

Since a terminal may provide voltages in the range of 1.8V to 5.0V, a level converter is build to provide stable 3.3V in the high state to the xmega pins. The level converter is integrated into the connection from the smartcard adapter to our device, as visible in Figure 4.3. We also add a reset switch to our device, as well as an LED board with four LEDs for visual feedback.

Figure 4.3: Our “MitM Smartcard Reader” with connected “EMSEC Smartcard Adapter”.

4.2 Firmware

The firmware is the code that runs on the xmega microcontroller. It controls our device and routes messages to the available endpoints such that a MitM attack becomes possible. To have the device recognized as a virtual serial port, we use Jürgen H.’s “CDC for xmega” code [10], which he provides for free. It is inspired by the free LUFA project [11].

The firmware operates in two stages: Initialization and operation mode. During initialization the smartcard and the terminal establish a communication session with our device by exchanging [ATR] and, in the case of the smartcard, a [PPS]. In operation mode, the communication between the smartcard and the terminal is re-routed to our software, to enable the MitM functionality.
4.2.1 Byte Transmission

The xmega is equipped with a Universal Synchronous/Asynchronous Receiver Transmitter (USART) module. The USART byte transmission is nearly similar to the byte transmission specified in ISO 7816. The only difference is that, instead of a guard time with parity error detection, USART transmits one or two stop bits. If the USART module is configured to use two stop bits, the timing and transmission is exactly the same as with the ISO 7816 method. However, it does not feature the parity error detection. Since we do not encounter any parity errors in our tests, we consider this as unproblematic. The USART baud rate is computed as \( \frac{1}{576} \).

4.2.2 USB Communication

The "MitM Smartcard Reader" communicates with custom software via USB. We provide a C++ class that can be used as a software basis, which we discuss in Section 4.3.1. Data sent via the USB-connection is encoded in a custom package. Its structure can be seen in Figure 4.4.

channel (1 byte) | length (2 bytes) | value (length bytes)

Figure 4.4: Structure of a USB package.

The channel byte encodes the target the data is intended for. It may represent a system channel for control messages, a channel for data that is on the way to the card and a channel for data that is on the way to the terminal. The length field is a 16 bit value that displays the number of bytes transmitted in the value field. These two bytes are required, since the maximum \( T = 1 \) I-block length is 258 bytes as shown in Section 2.3.7. This number is too large to be represented by a single byte.

The system channel can be used to configure our device as we explain in Section 4.2.7.
4.2 Firmware

4.2.3 Initialization Sequence

Figure 4.5: Initialization sequence of the “MitM Smartcard Reader”.

Figure 4.5 shows the initialization steps of our device. After plugging the USB cable into our device to power it up, it remains in an initialization state. When a smartcard is plugged in, a card reset is performed as described in Section 4.2.4. After this reset, the firmware holds a manipulated version of the smartcard’s ATR and the smartcard is configured to communicate with our device at default communication speed, while using the \( T = 1 \) protocol.

The manipulated ATR is guaranteed to make the receiver use the default communication speed, the \( T = 1 \) protocol, and direct convention for byte transmission.

As soon as the smartcard adapter gets plugged into a terminal for the first time, the firmware measures the clock frequency the terminal provides. Since the initial communication speed depends on the clock frequency as mentioned in Section 2.3.2, no communication takes place before the measurement is done. This measurement is done by counting the rising edges on the CLK pin of the smartcard adapter over a fixed amount of time. When the frequency is measured the firmware waits for the RST-line to be pulled high by the terminal. Once this reset signal is detected, the firmware replies with the modified ATR of the smartcard. Thus the terminal communicates at default speed and it uses the \( T = 1 \) protocol. Since the frequency measurement is a long procedure, impatient terminals may already reject the card, as no communication has taken place during the measurement. Therefore the measurement is only done once. In case of a card rejection, the adapter simply has to be pulled out and plugged in again.
When the smartcard and terminal sides are initialized, the device is operational and every transmission is re-routed. By default the device operates now in pass-through mode. Configuring the mode of operation is explained in Section 4.2.7. We explain the message flow in both modes in Section 4.2.6. Before that we go more into detail about the card reset and terminal disconnect procedures.

### 4.2.4 Card Reset

When the firmware resets the smartcard, it first resets the internal state machine and then performs a hardware reset of the smartcard. It clocks the smartcard at 4Mhz and pulls its RST-line to low state. If the smartcard has an active-low reset, it will respond with an **ATR**. If no **ATR** is received within 45000 cycles, the RST-line is pulled to high state. The card should now finally respond with its **ATR** as it is likely to feature an active-high reset. If there is still no response received, the procedure is repeated until an **ATR** is successfully received or the smartcard is pulled out.

Once an **ATR** is received, a manipulated version is stored. It is used later, when the terminal is connected.

The TS byte is set to 0x3B to indicate direct convention.

We omit any communication speed changes, so we do not have to change the **USART** module’s baud rate, because this did not work in our tests. Therefore, the TA1 byte gets removed, if it is available, to signal no changes in communication speed.

We also set the \( T = 1 \) protocol as the primary offered protocol. Therefore, the TD1 byte is added, if it is not already available and the lower nibble of TD1 is set to 1, to indicate the \( T = 1 \) protocol.

The T0 character is adjusted to indicate the absence of TA1 and the availability of TD1 and the checksum byte TCK is also adjusted.

Our device then transmits the **PPS** [0xFF, 0x11, 0x11, 0xFF]. The first byte, PPSS, is always 0xFF to indicate the start of a **PPS** message. The second byte, PPS0, selects the \( T = 1 \) protocol and signals that PPS1 is sent. The third byte, PPS1, sets the communication speed parameters to default, by changing TA1 to 0x11, and the last byte, PCK, is the XOR checksum over the previous PPS bytes.

If the smartcard responds with the same bytes, it is considered to be correctly reset and is ready to use.

### 4.2.5 Terminal Disconnect

A terminal disconnect occurs, when the smartcard adapter is removed from the terminal, or the terminal issues a reset by pulling the RST-line low. The firmware then resets the internal state machine for the terminal and initiates a smartcard reset as described in Section 4.2.4 if the auto-reset feature is activated. We describe the configurable features of our device in Section 4.2.7. After that, it returns to the initialization state and waits for the RST-line to be pulled back to high, as described in Section 4.2.3.
4.2 Firmware

4.2.6 Message Routing

Once the initialization of the smartcard and the terminal is done our device becomes active and the terminal will eventually start sending APDUs to the card. As all communication partners are using the $T = 1$ protocol we can be sure that a message which contains an R- or an S-block has exactly 4 bytes as explained in Section 2.3.7. Therefore every message with less than 5 bytes is forwarded directly to the correct target. The R- and S-blocks of the $T = 1$ protocol can be ignored in our MitM attack, since we only target APDUs.

The message routing is different whether our device is configured to operate in pass-through mode or in MitM mode. We explain the configuration process in Section 4.2.7.

Pass-through mode is the default mode after initialization. Every received message is forwarded to the original target. In addition it is sent to the software to enable eavesdropping. The device will ignore any incoming messages from the software which are not on the system channel. Therefore no active attack is possible in this mode. Figure 4.6 shows this routing process.

![Figure 4.6: An abstract view on the message routing when our “MitM Smartcard Reader” operates in pass-through mode. All messages that are received by the device, are additionally sent to the software.](image)

In MitM mode, an active attack becomes possible. Once an APDU is received from the terminal, the firmware sends this package over the USB-connection to the software. The APDU may be manipulated there and is sent back to the device. Upon reception, it forwards the message to the smartcard. The response takes the same way in reverse: It is transmitted to the software, where it may be altered. It then gets returned to the
device, which forwards it to the terminal. This message routing process is also shown in Figure 4.7.

![Figure 4.7: An abstract view on the message routing when our "MitM Smartcard Reader" operates in MitM mode.](image)

### 4.2.7 Device Configuration

Our device can be configured at runtime by using the system channel to send commands from the software. Upon reception it tries to execute the command and returns a response code. Our C++ class, which we explain in Section 4.3.1, features a function to send a command, which returns the response code from our device.

A full list of commands and an explanation is shown below:

- **MITM_COMMAND_PASS_THROUGH_MODE:**
  The device is set to pass-through mode. The messages are routed as explained before in Section 4.2.6.

- **MITM_COMMAND_MITM_MODE:**
  The device is set to MitM mode. The messages are routed as explained before in Section 4.2.6.

- **MITM_COMMAND_RESET_CARD:**
  The connected smartcard is reset. This may be needed, for example, if own APDUs are injected and the card has to be reset after particular transmissions.

- **MITM_COMMAND_AUTO_RESET_CARD_ON:**
  This turns the auto-reset feature on. If the terminal resets the smartcard adapter, our device will reset the terminal logic and the smartcard.

- **MITM_COMMAND_AUTO_RESET_CARD_OFF:**
  This turns the auto-reset feature off. If the terminal resets the smartcard adapter, our device will only reset the terminal logic.
4.3 Software

In this section we will take a look at the capabilities of the software that runs on the connected PC.

4.3.1 C++ Class

We provide the C++ class `MitMBoard` to enable simple software development for our “MitM Smartcard Reader”. We give a short overview about the functions of this class and provide a short code example.

- **Control Functions:**
  - `MitMBoard();`
    - Our class features a parameterless constructor. The value for the read timeout is set to 1000ms.
  - `uint16_t getTimeout();`
    - Returns the current timeout value.
  - `void setTimeout(uint16_t timeout);`
    - Sets the timeout value to the parameter.
  - `bool connect(const std::string& path);`
    - Establishes the USB-connection with our device.
  - `void disconnect();`
    - Closes the USB-connection. This function is called in the destructor if the connection is still established.
  - `const bool& isConnected();`
    - Returns whether the USB-connection is established.

- **Raw Data Read and Write:**
  - `bool readBytes(buffer_t& result);`
    - Reads all available bytes from the USB-connection into the buffer. This function uses the timeout and returns true on success.
  - `bool writeBytes(const buffer_t& frame);`
    - Sends the buffer to our device. Returns true on success.

- **APDU Functions:**
  - These functions extend the simple data read and write functions with removing or adding of the USB-package-header bytes and the $T=1$ header bytes. An APDU
read call will also store the channel on which the APDU was sent. These functions should be used by the programmers when implementing attacks.

- `bool readAPDU(buffer_t& apdu);`
  Reads an APDU into the buffer. The USB header bytes are removed. The reusable $T = 1$ bytes NAD and PCB are stored.

- `Channel getLastAPDUChannel();`
  Returns the channel of the last APDU which was received by readAPDU.

- `bool writeAPDU(const Channel channel, buffer_t& apdu);`
  Sends the APDU in the buffer to the passed channel on the device. Before sending the APDU is preprocessed:
  The reusable NAD and PCB $T = 1$ bytes from the last received APDU are added. If those bytes were wrong the receiver would reject the package. The LEN and LRC bytes are calculated and added, along with the USB header bytes.

- **System Commands:**
  We offer a function to configure the MitM Smartcard Reader.

  - `int sendCommand(uint8_t command);`
    Prepends the USB header bytes for the system channel to the command and sends this package to the device. Then the device’s response code is returned. On a transmission error, -1 is returned.

Listing 4.1 contains a code example. This example implements an eavesdropping attack. At first a connection is established with the virtual serial port of our device. If there was no error, the code tries to read APDUs endlessly. If a read returns success, the channel of the received APDU is evaluated and the channel information is printed to the console, followed by the actual APDU bytes.
4.3 Software

4.3.2 MitM Tool

To perform the attacks we describe in Chapter 5, we use a small command line tool to interact with our device, using our C++ class. The software features eavesdropping on a communication or performing a MitM attack by manipulating certain response APDUs. In eavesdropping mode, the tool prints every received APDU together with the channel it is sent on in hexadecimal representation.

In MitM mode, it scans the command-APDUs until a target file is selected and then manipulates the response-APDU to the next read command from the terminal. It would be possible to simply scan for a certain response and then manipulate its bytes, but in this case the attack would only work for the smartcard that sends exactly this response. As soon as card-specific data, like a card number, is included, the direct scan would fail.

```c
#include <stdio.h>
#include "mitm_board.h"

int main()
{
  MitMBoard board;
  MitMBoard::buffer_t buffer;

  // connect to the device
  if (!board.connect("/dev/ttyS2"))
  {
    return 1;
  }

  // endless loop
  while (1)
  {
    // if an APDU was transmitted
    if (board.readAPDU(buffer))
    {
      // print the APDU target
      if (board.getLastAPDUChannel() == MitMBoard::ToCardChannel)
      {
        printf("APDU to card: ");
      }
      else
      {
        printf("Response to terminal: ");
      }

      // print the APDU bytes
      for (size_t i = 0; i < buffer.size(); ++i)
      {
        printf("%02X ", buffer[i]);
      }
      printf("\n");
    }
  }
  return 0;
}
```

Listing 4.1: Sample code that implements an eavesdropping attack.
Our attack strategy works for every card, as long as the command `APDU`s do not change. A disadvantage is that the attack is hardcoded and therefore the tool has to be recompiled when the attack changes. We use a custom makefile for `GNU Make` to easily compile and run the tool.

### 4.4 Limitations and Known Issues

Our code works correctly and reliably in most cases. Some smartcards are not supported though. We do not support smartcards which are not capable of transmitting via the `\( T = 1 \)` protocol.

In addition there are situations that still confuse the firmware or the software:
If the software is connected to the device, but the device is reset by pulling the plug or by using the reset switch, the software will of course stop working. Furthermore it is only able to establish a new connection if the device is reset again while the software is not running.

Another issue is the frequency measurement at the terminal initialization. This process takes a lot of time to be accurate and is therefore only executed once. If the smartcard adapter is plugged into another terminal without resetting the device and the terminal clock frequencies are different, the communication will not work.

One last note is that the device does not automatically change its mode of operation, except for a hardware reset. When it is set to `MitM` mode and the software is terminated, the device still sends the `APDUs` over the USB-connection and waits for responses from the software. The user has to remember to reset the device or change the mode of operation back to pass-through mode when the software is terminated but the device is still in use.

### 4.5 Possible Timing Problems

When our “`MitM` Smartcard Reader” is used in `MitM` mode, the distance the data travels gets much longer than it would typically be. The transmission of the `APDUs` to the software and back to the device requires time, as well as the fact that the firmware only sends complete packages. It has to wait for an `APDU` to be fully received from the terminal or the smartcard, before it is able to prepend the USB header bytes. One might think that this introduces timing problems.

This is not the case. As there are millions of different smartcards from different manufacturers, they can not guarantee command completion in a certain time window. In other words, the terminal does not know when a response will arrive. It only knows `that` one will eventually arrive. Therefore, the additional way via our device does not have any dangerous timing impact.

Additionally, the USB communication is very fast in comparison to the terminal and
smartcard communication speed. The terminal transmits with a baud rate of about 8000 to about 13500, depending on the clock frequency that can range from 3MHz to 5MHz as we explained in Section 2.3.1. The smartcard transmits with a baud rate of 10752, since we clock it with static 4MHz. The USB connection however is configured to transmit with a baud rate of 115200. The bottleneck in the message transmission time is the process of waiting for a complete APDU before sending it to the next station.
5 Attacking Smartcards

To proof the usability of our “MitM Smartcard Reader” we build in this work, we try to attack multiple smartcards. At first we show the basic functionality of manipulating APDU by interfering with the readout process of EC-cards. Then we try to attack the “Jugendschutzmerkmal” of the “GeldKarte”. All the APDU dumps are retrieved using our device and are presented as hex-dumps.

5.1 Manipulating Data Fields

To read EC-cards we use the program Chipcardmaster [6]. At first we eavesdrop on the APDU that the Chipcardmaster software sends and on the responses of the card. Partial APDU dumps and their respective cards can be seen, side by side, in Table 5.1. The APDU, which are not included in the table, are select APDU that were not successful, e.g. the requested file was not available on the smartcard.

In the following we explain the transmitted messages. APDU pair 1 contains a select APDU that is sent from the terminal to select a file, indicated by the instruction byte (INS) 0xA4. The smartcard acknowledges the successful execution with the response code 0x9000. The terminal traverses the file tree of the smartcard with further select APDU in the APDU pairs 2, 3, 4, and 5, which are all acknowledged by the smartcard. At that point the file EF_ID is selected. In APDU pair 6, the terminal issues a read-out of data. A read APDU has the INS byte 0xB2. The smartcard returns the requested data and ends the transmission with the status word 0x9000 to indicate success. In APDU pairs 7 and 8 the terminal selects the file EF_BETRAG step by step and the card acknowledges both select operations. APDU pair 9 contains a read APDU and the returned data, as well as the status word, which is 0x9000 again.

To sum up the above, at first the program selects and reads the file EF_ID and then it selects and reads the file EF_BETRAG. We deduce the file names by manually searching the card’s file system for the returned data in the Chipcardmaster tool.
<table>
<thead>
<tr>
<th>Command</th>
<th>APDU Pair</th>
<th>Card “Celine Hagbard”</th>
<th>Card “Hans Wurschtsch”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 A4 04 0C 09 D2 76 00 00</td>
<td>00 A4 04 0C 09 D2 76 00 00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25 54 44 01 00</td>
<td>25 54 44 01 00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 00</td>
<td>90 00</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00 A4 00 0C</td>
<td>00 A4 00 0C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 00</td>
<td>90 00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00 A4 01 0C 02 A7 00</td>
<td>00 A4 01 0C 02 A7 00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 00</td>
<td>90 00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00 A4 00 0C</td>
<td>00 A4 00 0C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 00</td>
<td>90 00</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00 A4 02 0C 02 00 03</td>
<td>00 A4 02 0C 02 00 03</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 00</td>
<td>90 00</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00 B2 01 04 00</td>
<td>00 B2 01 04 00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67 25 50 01 41 80 00 04 35</td>
<td>67 25 50 01 41 80 00 04 34</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3D 18 12 14 03 27 02 80 45</td>
<td>6D 18 12 14 03 27 02 80 45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>55 52 01 4E 00 06 90 00</td>
<td>55 52 01 4E 00 06 90 00</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00 A4 01 0C 02 A2 00</td>
<td>00 A4 01 0C 02 A2 00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 00</td>
<td>90 00</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00 A4 02 0C 02 01 04</td>
<td>00 A4 02 0C 02 01 04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 00</td>
<td>90 00</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>00 B2 01 04 00</td>
<td>00 B2 01 04 00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00 00 00 02 00 00 02 00 00</td>
<td>00 09 94 02 00 00 02 00 00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00 00 00 90 00</td>
<td>00 00 00 90 00</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: APDU dumps from a read-out process and the respective cards. The card number is marked yellow, the expiration date green, and the stored money orange.
5.1 Manipulating Data Fields

We compare the data dumps of multiple cards with the known values of the analyzed cards. In particular we search for the expiration dates, the card numbers, and the stored amounts of money. Finding these values is exceptionally easy, because they are BCD encoded and therefore, in hexadecimal representation, readable like decimal values. In the previously discussed Table 5.1 the card number is highlighted yellow, the expiration date green, and the stored money orange.

We implement the attack on these values in our software, using the technique described in Section 4.3.2. We scan for the select \textit{APDU} of the \textit{APDU} pairs 5 and 8 of Table 5.1. They select the files which hold the data we want to attack. If such an \textit{APDU} is found, the response to the next read \textit{APDU} is attacked, depending on the data it contains:

The file \textit{EF\_ID} contains the card number and the expiration date. We change the five card number bytes to 0x11 and set the expiration date to May of 2099. Since this field is stored in two bytes using the format YY MM, we set those bytes to 0x99 and 0x05.

The file \textit{EF\_BETRAG} contains in the first three bytes the amount of money that is stored on the card. The first two bytes store the euros and the last byte the cents. Although a “GeldKarte” can only be charged with up to \(\text{\euro}200\) by the charging procedure, the BCD format with three bytes allows values up to \(\text{\euro}9999,99\) to be stored. To demonstrate this, we change all three bytes to 0x99. The three bytes are necessary, though, because \(\text{\euro}200,00\) is 0x02 0x00 0x00 in this BCD format.

The program Chipcardmaster does not perform any sanity checks on the received data. Therefore we are able to successfully manipulate the unprotected data read-out. A comparison of the digital appearance of a card with and without our attack is shown in Figure 5.1.

![Figure 5.1: A side-by-side view of the same card as it is read out by the program Chipcardmaster. On the left our attack is not active, whereas on the right the communication is manipulated.](image)

The “Deutscher Sparkasse Verlag GmbH” provides a set of online applets for the S-Chip browser add-on to test the functionality of a “GeldKarte” at home. One of these applets is a read-out applet, just like the Chipcardmaster. We implement a
similar attack, where we just wait for different APDUs as the applet takes a different path through the file tree. The attack works well as shown in Figure 5.2.

![Figure 5.2: A side-by-side view of the same card as it is read out by the Sparkasse browser applets](image)

On the left our attack is not active, whereas on the right the communication is manipulated.

### 5.2 Attacking the Youth Protection of the “GeldKarte”

In the last section, we successfully attacked the read-out of unprotected data. This may lead to success in scenarios where critical data is transmitted without protection. In this section we try to attack the youth protection mechanism of the “GeldKarte”, the “Jugendschutzmerkmal”.

#### 5.2.1 Online Simulation

One of the Sparkasse applets is designed to test whether a “GeldKarte” is enabled for purchasing adult content, like cigarettes. This is done by retrieving the “Jugendschutzmerkmal” which is stored on the card, if available. Like in Section 5.1, we compare the APDU dumps of multiple cards. Again, we retrieve these dumps by using the eavesdropping mode of our tool. Most of the command- and response-APDUs are exactly the same for the examined cards. We examine a complete APDU dump of the youth protection sequence in Section 5.2.2.

Since only the last pair of APDUs is different we examine it in detail in this section. Listing 5.1 shows the last pair of APDUs for three tests of the same card.

The command APDU differs on bytes 15 to 23 on each transmission, highlighted yellow. This also applies for multiple tests with the same card. According to the document “Einführung und Überblick Elektronischer Fahrschein und Marktplatz” [4], the terminal has to provide a nonce to the card, when it requests the “Jugendschutzmerkmal” and the card has to include a MAC over the data, including the nonce, in the response. Hence, those eight bytes that differ in each command APDU contain the nonce value.
5.2 Attacking the Youth Protection of the “GeldKarte”

APDU to card: 08 A2 01 C7 1F 8A 0F B4 0D 83 01 14 87 08 A8 52 CE 71 B7 56 72 F2 80 09 04 00 8E CC 00 00 FF 00 00 96 01 00 00
Response to terminal: 99 02 62 82 8E 08 84 43 22 20 60 92 FE F6 62 82

APDU to card: 08 A2 01 C7 1F 8A 0F B4 0D 83 01 14 87 08 4C B2 F8 E5 03 4D 9E 0D 80 09 04 00 8E CC 00 00 FF 00 00 96 01 00 00
Response to terminal: 99 02 62 82 8E 08 A1 FC 61 56 91 21 A9 C9 62 82

APDU to card: 08 A2 01 C7 1F 8A 0F B4 0D 83 01 14 87 08 28 11 BC EA E2 55 D2 29 80 09 04 00 8E CC 00 00 FF 00 00 96 01 00 00
Response to terminal: 99 02 62 82 8E 08 64 23 C4 7C 11 28 DA 90 62 82

Listing 5.1: The last set of APDU of multiple youth protection tests when using the Sparkasse browser applets. The bytes that differ between the tests were marked yellow in the command APDU and green in the response APDU.

The response APDU changes completely for each card, as it transmits the individual unique “Jugendschutzmerkmal”, if available, and the MAC. Multiple tests with a single card show, that only eight bytes change on each test, highlighted green. These bytes hold the MAC over the previous data, including the nonce from the command APDU. In addition, the byte before this MAC is 0x08, a length indicator, as the “GeldKarte” supports MACs of length eight and sixteen bytes. The same length indicator can be found in front of the nonce.

We implement an attack that replaces the “no Jugendschutzmerkmal available” response of a card, with the “Jugendschutzmerkmal available” response of another card. However, regardless if we leave the MAC unchanged or overwrite it, the applet detects our attack and rejects the card by signaling a problem with the security module.

An additional test where we just increment one of the MAC bytes proves that the attack fails at the MAC verification, as the card gets rejected with the same error message. As a conclusion, the “Jugendschutzmerkmal” is well protected with a MAC that uses a nonce, which is properly checked by the test applet. Hence, if the security mechanisms are working well on both sides, and the MAC is properly verified, we cannot perform a MitM attack.

5.2.2 Cigarette Vending Machine

The results from the applet tests lead to another attempt. We try to attack the “Jugendschutzmerkmal” again, this time on a cigarette vending machine near our university. We hope that the nonce generation is badly implemented or that the security checks are not performed. The best scenario would be that the MAC verification is skipped, that there is no randomness at all, or that just a small, limited amount of random numbers is used. The first thing noticeable is that the machine performs two message sequences when a card is plugged in. The first sequence is the retrieval of the “Jugendschutzmerkmal”.
The second is the retrieval of a certificate and the amount of money from the “GeldKarte”.

<table>
<thead>
<tr>
<th>APDU Pair</th>
<th>Cigarette Vending Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Command:</td>
<td>00 A4 04 0C 09 D2 76 00 00 25 4D 01 02 00</td>
</tr>
<tr>
<td>Response:</td>
<td>90 00</td>
</tr>
<tr>
<td>2 Command:</td>
<td>00 B2 01 BC 00</td>
</tr>
<tr>
<td>Response:</td>
<td>67 25 34 14 51 50 18 66 47 8D 15 12 11 10 17 02 80 45</td>
</tr>
<tr>
<td>3 Command:</td>
<td>B0 EE 80 0F 00</td>
</tr>
<tr>
<td>Response:</td>
<td>10 11 12 13 14 15 16 17 18 19 90 00</td>
</tr>
<tr>
<td>4 Command:</td>
<td>08 B2 0A C4 14 BA 0F B4 0D 83 01 16 87 08</td>
</tr>
<tr>
<td>Response:</td>
<td>81 3C E3 3A C2 01 0A C4 03 00 00 00 C5 07 99 99 12 31</td>
</tr>
<tr>
<td></td>
<td>23 59 00 E6 27 CC 03 FF 00 00 CE 12 00 10 29 43 17 96</td>
</tr>
<tr>
<td></td>
<td>26 00 0D E9 C9 AA 58 79 F9 F8 1A 22 C6 0C 4A 75 67 65</td>
</tr>
<tr>
<td></td>
<td>6E 64 73 63 68 75 74 7A 8E 08 28 64 1E 8F AF D4 17 40</td>
</tr>
<tr>
<td></td>
<td>90 00</td>
</tr>
</tbody>
</table>

Table 5.2: APDU dump from the first message sequence of the cigarette vending machine.

At first EF_ID is selected and read, then key information is requested and finally the “Jugendschutzmerkmal” is read.

Table 5.2 shows the exchanged APDUs of the first sequence. It starts with the selection of EF_ID by APDU pair 1. The file is then read with APDU pair 2. Pair 3 obtains key information from the smartcard. The last APDU pair reads the “Jugendschutzmerkmal”. Again one can identify the nonce, highlighted yellow, and the MAC, highlighted green, as well as the length identifiers in front of them. The sequence follows the specification in the document “Einführung und Überblick Elektronischer Fahrschein und Marktplatz” [4, p. 40].

The second sequence is not part of the youth protection. The exchanged APDUs can be seen in Table 5.3.

APDU pair 1 selects the file DF_MARKTPLATZ. The next APDU pairs read the files EF_ID (pair 2), EF_SIGD (pair 3), EF_SIG (pair 4), and EF_BETRAG (pair 5). This is important to know for our attacks: The messages of this sequence are of no significance for the youth protection mechanism. Because of that, we do not have to pay attention to them when we implement attacks.

We think that the file DF_MARKTPLATZ is only selected to verify that a “GeldKarte” is inserted, since the selected file is not used later. The read APDUs all use file identifiers to read the files without selecting them first. The content of these files
5.2 Attacking the Youth Protection of the “GeldKarte”

Table 5.3: APDU dump from the second message sequence of the cigarette vending machine. At first DF_MARKTPLATZ is selected and then the files EF_ID, EF_SIGD, EF_SIG, and EF_BETRAG are read by the APDU pairs 2, 3, 4, and 5.

<table>
<thead>
<tr>
<th>APDU Pair</th>
<th>Cigarette Vending Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Command: 00 A4 04 0C 09 D2 76 00 00 25 45 50 02 00</td>
</tr>
<tr>
<td></td>
<td>2 Command: 00 B2 01 BC 00</td>
</tr>
<tr>
<td></td>
<td>3 Command: 00 B2 01 AC 00</td>
</tr>
<tr>
<td></td>
<td>4 Command: 00 B2 01 A4 00</td>
</tr>
<tr>
<td></td>
<td>5 Command: 00 B2 01 C4 00</td>
</tr>
</tbody>
</table>

might be used for the rest of the transaction and for user interaction. For example, EF_BETRAG has to be read to show the user how much money is available on his smartcard.

We implement an attack similar to the attack on the Sparkasse applet, but the vending machine instantly rejects our card when we manipulate the message that contains the “Jugendschutzmerkmal”. As the MAC verification is working, we focus on the nonces. We compare the random numbers transmitted in APDU pair 4 of Table 5.2 from fifteen runs, shown in Listing 5.2. Our goal is to find repetitions or patterns, which we can use to precompute the nonces. We cannot find any relations, though.
In conclusion, we are not able to successfully attack the youth protection of the “GeldKarte”
on the cigarette vending machine near our university. The MAC verification is working
and the nonces appeared to be random.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | dump 01: 82 | F1 | 76 | 48 | EA | 43 | D8 | 1B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 2 | dump 02: 53 | 43 | 4F | 3A | 98 | 40 | E2 | 9B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 3 | dump 03: F3 | 71 | 97 | 02 | 8B | 5C | OD | 5E |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 4 | dump 04: 0F | 8F | 8A | 27 | 1B | 6D | 1F | 27 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 5 | dump 05: 69 | DE | 17 | 12 | 17 | 63 | 5D | F4 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 6 | dump 06: 20 | FE | 08 | A6 | 1D | 87 | 82 | 86 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 7 | dump 07: CB | 91 | 59 | 7B | 7C | DD | 3A | 72 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 8 | dump 08: 8E | C0 | CD | 89 | 80 | C4 | 0E | B3 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 9 | dump 09: 27 | D4 | 44 | B4 | 1C | AB | 1C | DA |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|10 | dump 10: 40 | 59 | 14 | D1 | BD | 5E | 7F | C0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|11 | dump 11: 65 | 16 | 14 | 81 | B8 | E6 | F1 | 4C |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|12 | dump 12: D5 | 9C | C7 | 5E | 2A | 1B | 8F | 0B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|13 | dump 13: 2C | 50 | 54 | 18 | 93 | 32 | 91 | 74 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|14 | dump 14: D3 | D7 | A1 | 0E | F7 | 0E | 94 | 07 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|15 | dump 15: FE | 3F | 70 | 26 | 68 | 0A | 0C | 22 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Listing 5.2: The random values sent by the cigarette vending machine.
6 Conclusion

In this section, we summarize the results of this thesis and give ideas for future work.

6.1 Summary

In this thesis we have built a device that is capable of intercepting the communication between a smartcard and a terminal. It forwards the transmitted messages to custom software, which may manipulate the received data, before sending it back to the original target. Attacks can be implemented with the least possible restrictions, as we provide basic functions to implement every kind of MitM attack. We successfully manipulated unprotected data transmissions, like the read-out procedure of an EC-card. We used this to alter a smartcards digital appearance, by changing the available amount of money, its expiration date, and its card number. We also attacked protected data transmissions in the case of the youth protection of the “GeldKarte”. This attempt failed, because the “Jugendschutzmerkmal”, the youth protection of the “GeldKarte”, was well protected with a salted MAC. We could not successfully attack it in a real life scenario, too, because the security mechanisms on the cigarette vending machine we attacked were correctly implemented.

6.2 Future Work

For future work it is possible to optimize the device’s hardware. The board layout can be optimized to greatly reduce the size of our device. Currently, there is a lot of unused space on the board. The intermediate pins between the smartcard and the xmega (index 3 of Figure 4.2) can be omitted. The level converter, as well as the LEDs and the reset switch should be integrated onto the board. For more detailed visual feedback we suggest to increase the number of LEDs to eight and to adjust the firmware accordingly.

The firmware may be improved as well: One could try to permit higher transmission speeds between the terminal, the device, and the smartcard, to reduce waiting times on longer processes. One might also try to fix the known issues described in Section 4.4.

Regarding the attack on the cigarette vending machine, further research could be put into the process of payment. The available money was read without any protection in APDU pair 5 of Table 5.3, so an attack similar to the attacks on the read-out processes of the Chipcardmaster and the Sparkasse applet can increase this amount. If the machine is badly programmed, it might ignore the transaction result if enough money was initially
on the card. It might think that if enough money was on the card, the transaction will always succeed. In that case, goods would be returned although the actual payment process fails.

Examined from a more general point of view, the banking sector is highly aware of possible threats and therefore puts a lot of effort into the security mechanisms in its products. We think that it is interesting to analyze smartcards in other sectors, for example those which are used for access control, as the probability of finding a successful attack may be higher there.

In summary, the device should now be put to use to inspect contact-based smartcards in varying environments in order to identify insecure transmissions in security-sensitive contexts.
## A  Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>APDU</td>
<td>Application Protocol Data Unit</td>
</tr>
<tr>
<td>ATR</td>
<td>Answer to Reset</td>
</tr>
<tr>
<td>BCD</td>
<td>Binary Coded Decimal</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>DF</td>
<td>Dedicated File</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>EF</td>
<td>Elementary File</td>
</tr>
<tr>
<td>EMV</td>
<td>Europay International, MasterCard and VISA</td>
</tr>
<tr>
<td>ETU</td>
<td>Elementary Time Unit</td>
</tr>
<tr>
<td>ICC</td>
<td>Integrated Circuit Card</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>MF</td>
<td>Master File</td>
</tr>
<tr>
<td>MitM</td>
<td>Man-in-the-Middle</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>PPS</td>
<td>Protocol Parameter Selection</td>
</tr>
<tr>
<td>RFU</td>
<td>Reserved for Future Use</td>
</tr>
<tr>
<td>USART</td>
<td>Universal Synchronous/Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A typical smartcard. The white part is the plastic case, and the golden part is the microcontroller. .................................................. 5
2.2 Detailed view of a smartcard microcontroller, including the pin descriptions. ................................................................. 5
2.3 Byte transmission as specified by ISO 7816. .............................................. 6
2.4 The message sequence between a smartcard and a terminal. .................. 6
2.5 Structure of an [ATR] ................................................................. 7
2.6 Structure of a $T = 1$ frame. .......................................................... 10
2.7 The message sequence in the $T = 0$ protocol. The Data message is transmitted only in one direction, depending on the Command Header. ... 10
2.8 The message sequence in the $T = 1$ protocol. .................................................. 11
2.9 Structure of a command-[APDU] ..................................................... 11
2.10 Structure of a response-[APDU] ..................................................... 11
2.11 A smartcard’s file system. ............................................................ 12
2.12 BCD encoding of decimal values. The digits of a BCD formatted value are readable like their decimal representation. ...................... 13
4.1 The “MitM Smartcard Reader” we build in this work................................. 17
4.2 An abstract view on the layout of the “MitM Smartcard Reader”. .............. 18
4.3 Our “MitM Smartcard Reader” with connected “EMSEC Smartcard Adapter” ........................................................................... 19
4.4 Structure of a USB package. .............................................................. 20
4.5 Initialization sequence of the “MitM Smartcard Reader”. ......................... 21
4.6 An abstract view on the message routing when our “MitM Smartcard Reader” operates in pass-through mode. All messages that are received by the device, are additionally sent to the software. ............... 23
4.7 An abstract view on the message routing when our “MitM Smartcard Reader” operates in MitM mode. ......................................... 24
5.1 A side-by-side view of the same card as it is read out by the program Chipcardmaster [6]. On the left our attack is not active, whereas on the right the communication is manipulated. ................................. 33
5.2 A side-by-side view of the same card as it is read out by the Sparkasse browser applets [7]. On the left our attack is not active, whereas on the right the communication is manipulated. ................................. 34
List of Tables

2.1 The table for the clock rate conversion factor $F$. . . . . . . . . . . . . . . . 9
2.2 The table for the bit rate adjustment factor $D$. . . . . . . . . . . . . . . . 9
3.1 APDU dumps from a read-out process and the respective cards. The card
number is marked yellow, the expiration date green, and the stored money
orange. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2 APDU dump from the first message sequence of the cigarette vending
machine. At first EF_ID is selected and read, then key information is
requested and finally the “Jugendschutzmerkmal” is read. . . . . . . . . . 36
3.3 APDU dump from the second message sequence of the cigarette vending
machine. At first DF_MARKTPLATZ is selected and then the files EF_ID,
EF_SIGD, EF_SIG, and EF_BETRAG are read by the APDU pairs 2, 3, 4,
and 5. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
List of Listings

4.1 Sample code that implements an eavesdropping attack . . . . . . . . . . . 27

5.1 The last set of [APDU] of multiple youth protection tests when using the Sparkasse browser applets. The bytes that differ between the tests were marked yellow in the command [APDU] and green in the response [APDU] . . . . . . . . . . . . . . . 35

5.2 The random values sent by the cigarette vending machine . . . . . . . . . . . . . . . . . 38
Bibliography


