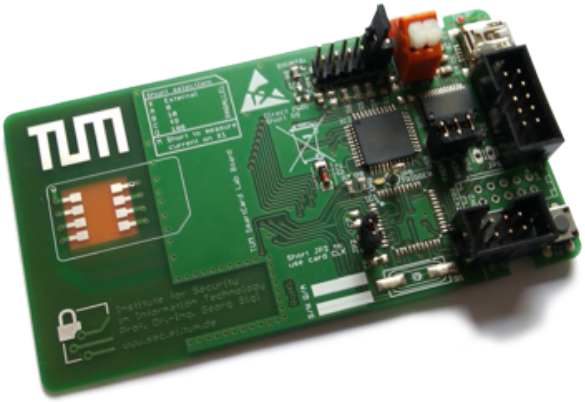
Technische Universität München
Fakultät für Elektro- und Informationstechnik
Lehrstuhl für Sicherheit in der Informationstechnik

Smart Card Laboratory

Laboratory Script

Prof. Dr.-Ing. Georg Sigl



Note: If you find any errors or would like to provide feedback and/or suggestions for improvement please send your comments to: *m.gruber@tum.de*

2015-2016 – Oscar M. Guillén Hernández

2016-2017 – Martha Johanna Sepulveda Florez

2017-2018 – Michael Gruber

Contents

Chapter 1: Lab overview

1.1 Task description

The goal of the lab is to introduce the concepts of Side-channel Analysis (SCA) and to help the students understand the implementation of SCA attacks and their countermeasures in a practical way. The *target of evaluation* for the lab is a *Smart Card* for a Pay-TV system. This Smart Card is used to decrypt video streams transmitted over the network. Analogous to how a commercial Pay-TV system works, only the PC's with a valid card connected to their reader are able to decrypt and decode the protected content.
The laboratory is divided into two phases:

1. Attacking and cloning the reference card.
2. Improving the security of your own card.

During Phase 1, you put on a "black hat" and become the attacker. Your goal is to extract the secret key stored within the device using of Differential Power Analysis (DPA). Later you use the extracted key to clone the original card by programming your own Smart Card emulator. In Phase 2 you put on a "white hat" and implement, test and improve different countermeasures aimed to protect the Smart Card that you programmed. You test the resistance of your countermeasures by improving the attack scripts. During this phase, the focus is placed on comparing the trade-offs between the security obtained and the cost in terms of program size and execution time of the countermeasures.

1.2 Organizational matters

1.2.1 Teamwork

The laboratory is performed in groups of four people. Each group is split into two teams, Team A and Team B. During Phase 1, Team A is responsible for developing the scripts for the DPA attacks, while Team B works on programming the Smart Card emulator which is used to clone the original card. In Phase 2, Team A is responsible for implementing countermeasures against DPA in the Smart Card emulator, while Team B improves the attack scripts to bypass these countermeasures.

> **Tip:** Pay attention to good communication within your group and help the other team with their tasks if needed. Remember that your final grade will depend on the performance of your group.

1.2.2 Milestones

There are three major milestones for the laboratory:

1. Intermediate presentation.
2. Final presentation.
3. Oral exam.

During the intermediate presentation you display the results of the first part of the lab ($i.e.$ Phase 1). These include your attack strategy to perform DPA, the design and implementation details of your clone card, and an outlook for the next steps. During the final presentation you describe in detail the DPA countermeasures that were implemented and the improvements made to the attack scripts in order to break them. Here you also show the comparison between the different countermeasures, their resistance, and the impact of code size and speed in your Smart Card emulator. Lastly, in the oral exam you are going to be asked questions based on the work you described in your lab protocol.

> **Tip:** On the slides for the first lecture you can find more information about the content which is expected to be covered during the intermediate and final presentations.

1.2.3 Laboratory hours

You are free to work on the lab tasks where and when you please. The laboratory room **N1003** is open from **Monday to Friday from 7:00 am to 9:00 pm**. Unless otherwise instructed, you are free to use any of the computers in the room. Please note that there are other students using the laboratory room during the semester as well and remember to **log out** when you are not using them.

1.3 Environment

For the completion of the assignments in the lab you are given several hardware tools. These include:

- Two Smart Cards emulators.
- AVR programmer (AVRISP MkII).
- Programmer helper (used to supply power and clock pulses to the card while programming).
- Logic analyzer.

Or if you are using the **new cards**:

- Two Smart Cards emulators.
- Arduino-based AVR programmer.
- Logic analyzer.

In addition to these tools, you are given a set of Python and Matlab scripts and you will have access to different tools which are installed in the computers found in the lab room. This section aims to provide a reference on how to make the most out of this.

1.3.1 Smart Card Emulator Hardware

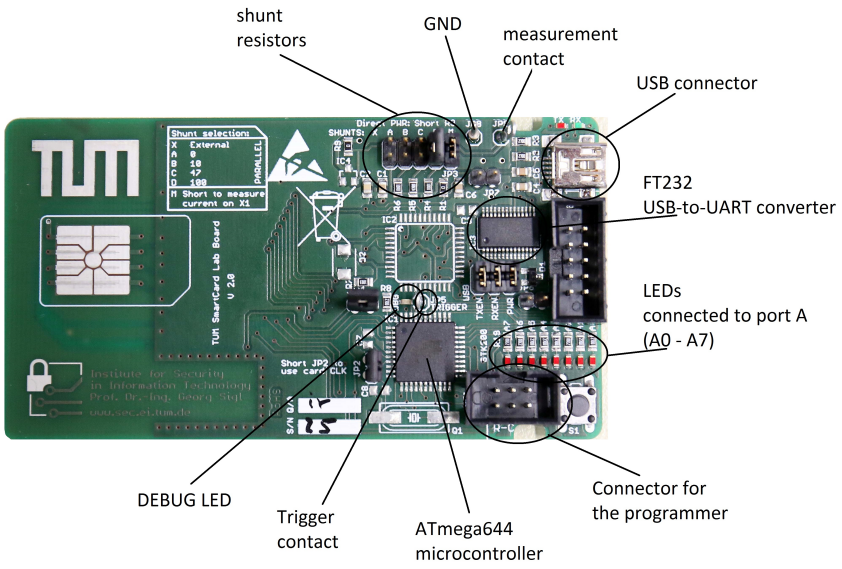


Figure 1.1: Smart Card

Each group receives two Smart Cards, the first one is a reference card with a working OS and a pre-programmed key, this will be your target of evaluation for the DPA attacks during the first part of the lab. The second one is a blank card, which is used to implement your clone card. Figure 1.1 shows the top side of the provided hardware. It contains an ATmega644 8-bit microcontroller, an USB-to-UART converter, which can be used to debug your program, different shunt resistors for the power measurements, connectors and debug LEDs. All the signals needed for the Smart Card (e.g. power supply, clock, I/O, reset) are transmitted over the ISO 7816 interface [1]. The pin assignment is shown in Figure 1.3.

For the **new cards**, you will notice that many of the components in the old cards are no longer there or have been changed. The microcontroller used in this design is the ATmega64 The jumpers that connect to the shunt resistors have been changed to DIP-switches. With them, you can now configure shunt resistors to be in the VCC or Ground path. There is just one debug-LED in the new boards. The USB-to-UART converter and the USB connector have been removed, however the serial interface pins are still available and can be found next to the programmer connector (ISP). Lastly there is a capacitor connected to the power supply of the microcontroller, which will help you get much cleaner measurements than before. For the measurements that you will perform in the lab, we suggest connecting the probes as shown in figure 1.2.

Tip: You can find the schematic of the board as well as the data sheet of the microcontroller attached to the lab materials. Before you start the lab take some time to get familiar with the hardware.

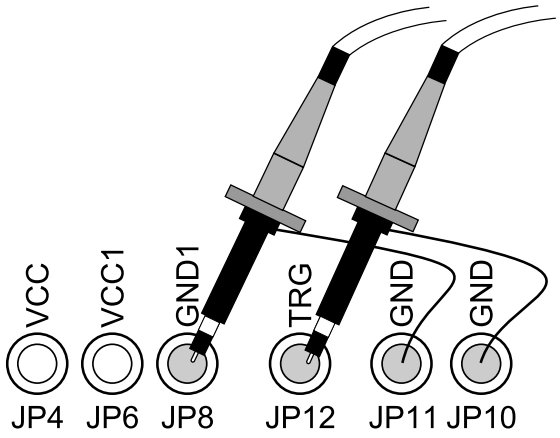


Figure 1.2: Suggested connection for the **new** cards.

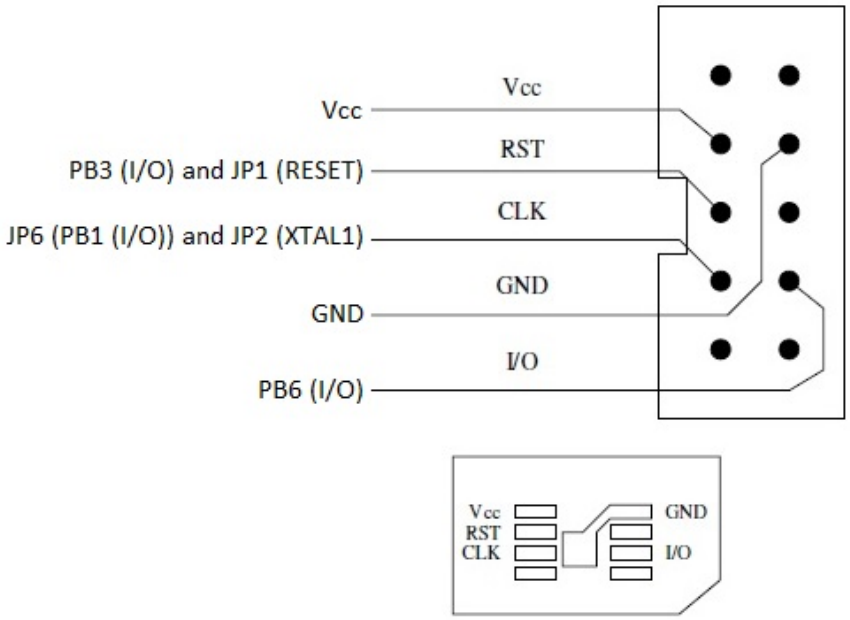


Figure 1.3: Pin assignment of a Smart Card and the connections in hardware.

1.3.2 Matlab

Matlab can be used to program your DPA scripts. To ensure you start the right version of Matlab (newer than R2014b), type the following command (from the Linux console):

```
$ module list
```

If an older version appears on the list, you can unload it using the following command:

```
$ module unload matlab
```

Finally load the latest module with the following command:

```
$ module load matlab/R2014b
```

Afterwards type "*matlab*" in the console and press enter to start the loaded module.
You can include the last command to the ".bashrc" file in your home directory to load the Matlab module automatically when you sign in to your computer.

1.3.3 Software Version Control

Introduction

A version control system is used during the lab to help you keep track of changes between different versions of your source code and documentation ($e.g.$ C code, Matlab scripts, presentation slides, etc). Having a way to track the changes of a project over time is very important when collaborating with different people on a single project. The version control system that you will be using in the lab is Git, specifically GitLab from the LRZ. `http://gitlab.lrz.de/`

Important commands

The following is a list of the most important commands. You can also find several cheat sheets in Internet. For a more in depth understanding, an online e-book on SVN is available at: `https://git-scm.com/book/en/v2`.

- Create a working copy::

  ```
  git clone username@host:/path/to/repository
  ```

- Update your working copy:

  ```
  git pull
  ```

- Print the status of working copy files and directories:

  ```
  git status
  ```

- Add files:

  ```
  git add <filename>
  ```

- Commit your changes:

```
git commit -m "descriptive␣comment"
```

- Send your changes to the repository:

  ```
  git push origin master
  ```

- Display the history of changes:

  ```
  git log
  ```

1.3.4 Logic analyzer

You are provided a logic analyzer as part of the material given for the lab. This tool will help you understand the communication protocol between the Smart Card and the terminal. To use it, you need to first download the software which you will run from your computer.
Software: logic 1.1.15 (64-bit)
`http://downloads.saleae.com/Logic%201.1.15%20(64-bit).zip`

Alternatively you can use the software pulseview which is already installed. To display the data values at the I/O line, use the following configuration of the logic analyzer:

- Async Serial Analyzer
- Baudrate = 4.8Mhz / 372 clocks = 12900 (bit/sec)
- 8 bits per transfer
- 1 bit stop
- Even parity
- LSB first
- Non inverted

If you have not used a logic analyzer before, take some time to familiarize yourself with the hardware. A good starting point is the documentation found in the website of the manufacturer: `https://www.saleae.com/`.

1.3.5 Pay-TV scripts

Working in the lab

The lab materials found in the compressed file which you received at the beginning of the course include several scripts. The following is a description of their use.

- client
  With this script, the video stream can be started. The correct SmartCard has to be inserted to decrypt the stream. You can execute the script, using the following parameters (please note that the **last two digits** correspond to your group number -$eg.$ 20001 for Group 01):

  ```
  $ ./client tueisec-sigltv 20042
  ```

  If you have problems running the script, make sure you that the file has execute rights for your user. You can add the rights using the following command:

  ```
  $ chmod +x ./client
  ```

- test_system
  This script helps each team within a group to test their own implementations independently. With it a clone card can be tested even before obtaining the correct master key. The script can also be used to test if the key extracted using DPA is the right one, without the need of a clone card.

Working from home

If you have a smart card reader at home, you can test your clone card without coming to the lab. The following instructions were tested under Debian GNU/Linux.
First install the following packages on your computer: pcsc-tools, pcscd, python-crypto, python-pyscard.
Download the chunks of the encrypted video stream into your computer (please note that "x" corresponds to your group number -$eg.$ 20001 for Group 1-):

1. Log into the LRZ VPN from home.
2. Store the chunks for offline use

   ```
   $ nc TUEISEC-SiglTV.sec.ei.tum.de 2000x > enc-video-file
   ```

3. Use the script to decrypt the video stream using your Smart Card

   ```
   $ cat enc-video-file | python receiver.py | vlc - vlc://quit
   ```

Alternatively, if your internet connection is fast enough, you may stream the encrypted content using:

- ```
  $ nc TUEISEC-SiglTV.sec.ei.tum.de 2000x | \
  python receiver.py | vlc - vlc://quit
  ```
  ```

1.3.6 Programming AVRs using a Makefile

Programming AVRs can be done without an IDE, all you need is a Makefile that takes care of compiling, linking, and flashing. A template for a Makefile can be found at the following link, where you need to make several adjustments.
`https://www.mikrocontroller.net/articles/Beispiel_Makefile`
A simple example program that makes an LED blink is in the following listing.

Listing 1.1: Blinking LED sample program

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{

    // Configure pin 7 on port A as output
    DDRA |= (1 << PINA7); // Debug-LED

    // Infinite loop
    while(1)
    {
        // Invert the output of pin 7 on port A
        PORTA ^= (1 << PINA7);
        _delay_ms(250);
    }
}
```

To flash the program (to the AVR) enter the following command in the terminal.

Listing 1.2: avrdude flash command

```
avrdude -p atmega644 -P usb -c avrisp2 -v -U flash:w:main.hex
```

1.3.7 Programming AVRs using Eclipse

Eclipse and the AVR tools are already installed on the lab computers. In order to configure Eclipse for programming the AVR microcontroller the following steps are required:

1. Install the C support for Eclipse CDT:
   - Start Eclipse (version 3.8, Juno)
   - Go to "Help>Install New Software".
   - In the "work with:" text box write:
     `http://download.eclipse.org/tools/cdt/releases/juno`
   - Press enter
   - Check the boxes for CDT Main Features and CDT Optional Features
   - Hit "Next>" and follow the instructions on the next pages.
2. If you run into dependency problems with the previous step, make sure that the software repository is correctly configured:
   - Go to "Help>Install New Software".
   - Click in the blue link "Available Software Sites".

- If it is not there, add the repository for Juno by including the following URL: `http://download.eclipse.org/releases/juno/`

3. Install the AVR Plugin (manual installation is required):

   - Download the AVR Eclipse Plugin stable release version 2.3.4: `http://sourceforge.net/projects/avr-eclipse/files/avr-eclipse%20stable%20release/2.3.4/avreclipse-p2-repository-2.3.4.20100807PRD.zip/download`
   - Open Eclipse
   - Go to "Help > Install New Software"
   - Click on the "Add..." Button.
   - In the new dialog hit "Archive...", select the downloaded file and click on "OK".
   - Select the AVR Eclipse Plugin from the list, hit "Next>" and follow the instructions on the next pages.

4. Create a project:

   - Open Eclipse
   - Select File > New > Project
   - In the new Project Wizard select C Project.
   - Select AVR Cross Target Application
   - Enter a project name (e.g. "hello_world")
   - Click "Next >"
   - Automatic build configurations can be selected
   - Click "Next >"
   - Select the target processor as ATmega644 (ATmega64 if using the **new cards**) and clock frequency to 3276800 (the values for MCU type and MCU frequency can later be changed via the project properties)
   - Click Finish

5. Create a source file:

   - Select the project you just created (e.g. "hello_world")
   - Click on "File > New > Source File"
   - Type main.c for the name
   - Click Finish.
   - Type the code from Listing 1.3 in the editor
   - Click "File > Save"
   - Click "Project > Build Project"

6. Generate the .hex file:

   - Right click on the Project and select "Build Configuration > Release"
   - Right click again on the Project and select "Build Project"
   - Open the "Release" folder, inside you should find the .hex file that will be programmed into the device

Listing 1.3: Blinking LED sample program

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{

    // Configure pin 7 on port A as output
    DDRA |= (1 << PINA7); // Debug-LED

    // Infinite loop
    while(1)
    {
        // Invert the output of pin 7 on port A
        PORTA ^= (1 << PINA7);
        _delay_ms(250);
    }
}
```

7. Configure the AVRISP MKII (previous cards)

    - Click "Project > Properties"
    - Expand the AVR menu
    - Select AVRDude
    - Create a new configuration
    - Select Atmel AVR ISP mkII
    - On "Override default port -P" write "usb"
    - The command line preview should look like this:
      avrdude -cavrisp2 -Pusb
    - Click "OK" and then click "Apply"

8. Configure the Arduino-based programmer (**new cards**)

    - Click "Project > Properties"
    - Expand the AVR menu
    - Select AVRDude
    - Create a new configuration
    - Atmel STK500 Version 1.x Firmware
    - On "Override default port -P" write you serial port, e.g. "/dev/ttyUSB0"
    - Change the bitrate to "38400"
    - The command line preview should look like this:
      avrdude -cstk500v1 -P/dev/ttyUSB0 -b38400

9. Program the device

    - Click on "AVR > Upload Project to Target Device"
    - Wait till the program is loaded
    - You may need to disconnect the programmer from the card in order to correctly see the LED blinking.

1.3.8 Using the serial port for debugging

The AVRISP MkII programmer which is included in the lab material does not provide debugging capabilities. Because of this, the hardware of the Smart Card emulator includes a USB-to-UART converter which may be used to connect the serial port from the microcontroller to a USB port in your computer. You may use this interface to send debug information to a console. To setup a serial console in Linux, please take a look at the following tutorials:
`https://help.ubuntu.com/community/SerialConsoleHowto`
`http://goo.gl/KswwA`

Chapter 2: The Pay TV system: SiglTV

2.1 Introduction

Hardware attacks such as Side-channel Analysis are a huge threat to the security of embedded devices, especially when the hardware is handed out to consumers as each one of them could be a potential attacker. This scenario is depicted in our laboratory. Each group receives a Smart Card, on which a cryptographic algorithm will be used to decrypt a protected video stream in a similar way to a real PayTV system. You can only decrypt and view the video stream if the card with the corresponding key is inserted into the card reader. To understand this scenario, we will take a look into how the SiglTV works.

2.2 Implementation

A server is used to transmit a video stream which should only be seen by authorized customers. First, the video stream is divided into chunks d_i with a size of 16 kB. These chunks are encrypted using AES. For each chunk, a random chunk key k_c is generated and used to encrypt d_i. The random chunk key k_c is then encrypted with a master key k_m using AES-128 as well. The encrypted data chunk d_{ic} and the encrypted chunk key k_{cc} are transmitted to the client. To obtain the chunk key, the client has to decrypt k_{cc} with the correct master key k_m. This operation is done by the SmartCard, which contains a secure copy of the master key k_m. Afterwards the client decrypts the video chunk. In the lab the client is a PC with a card reader.

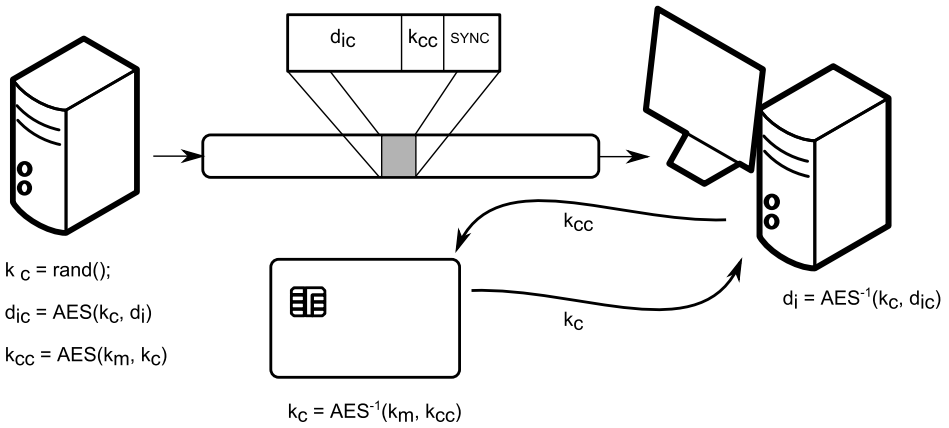


Figure 2.1: Structure of the PayTV-System

Figure 2.1 shows the general structure of a PayTV-system. The important parts can be summarized as follow:

- Server:
  - Video data stream is divided into chunks d_i
  - A random key k_c encrypts each data chunk
  - The random key k_c is then encrypted with a master key k_m to generate k_{cc}
  - k_m is known only by the server and the card
- Server sends packet that include:
  - An encrypted data chunk, d_{ic}
  - The encrypted random key, k_{cc}
  - Synchronization data
- PC:
  - Sends the encrypted random key, k_{cc} , to the SmartCard
  - Uses k_c to decrypt d_{ic}
  - Displays the plaintext data chunk d_i
- SmartCard
  - Decrypts k_{cc} with k_m to obtain k_c

Chapter 3: Differential Power Analysis (DPA)

3.1 DPA Attacks

Power consumed by an electronic circuit varies according to the activity of the transistors and other components within it. As a result, measurements of this physical property will contain information of the operations and data being processed within a device. DPA attacks exploit this side-channel information by finding the relation between the measured power consumption of a cryptographic device and the secret key.[3, p. 119]

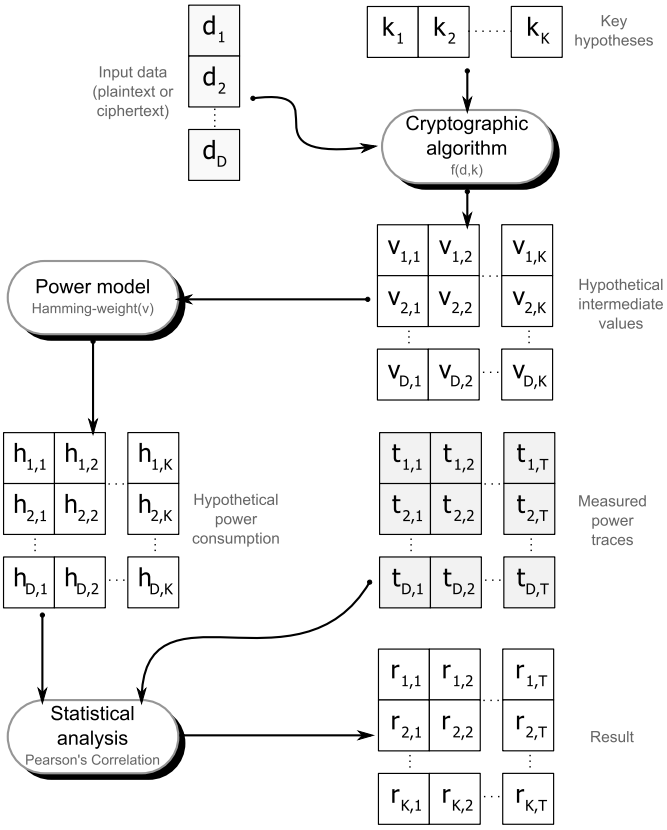


Figure 3.1: Block diagram of the steps of a DPA attack. The blocks in light gray correspond to the inputs to your script. The blocks in white correspond to the values which are generated within the script.

Figure 3.1 shows a block diagram of the steps involved in a DPA attack [3, p. 120ff]. The first step is to choose an intermediate result to attack. We are looking for a function that makes use

of the secret we want to extract and values which we know. This function can be expressed as $v = f(d, k)$, where d is a known, variable, data value, k is a *part* of the key and v is the intermediate value resulting of this operation ($e.g.$ $v = f(d, k) = Sbox(d \oplus k)$ when targeting the 8-bit S-box function in the first round of AES, $cf.$ Figure 3.2). The second step is to measure the power consumption while the device encrypts or decrypts a data block D. For each of these runs, you as an attacker need to know the corresponding data values d. To perform the attack, you have to calculate a hypothetical intermediate value using the known function $v = f(d, k)$ and known data values d for every possible choice of k. *This is why k has to be a part of the key and not the complete key itself.* Think about it as a Divide-and-Conquer approach. The next step is to map the matrix of hypothetical intermediate values V to a matrix of hypothetical power consumption values H, making use of a power model. The most commonly used power models are the Hamming-distance and the Hamming-weight model. For this lab, you will make use of the latter. The Hamming-weight power model assumes that the instantaneous power being consumed by a device is proportional to the number of bits which are different to zero in the binary representation of a given value ($e.g.$ the value $4_{10} = 00000100_2$ would have a Hamming-weight of 1_{10}, while the value $255_{10} = 11111111_2$ would have a Hamming-weight of 8_{10}). The last step of the DPA attack is the comparison of each column of the hypothetical power consumption matrix H with each column of the measured power traces matrix T. For this we make use of the Pearson's Correlation Coefficient in Equation 3.1. For which $i = \{1..K\}$, $j = \{1..T\}$ and $\overline{h}_i$ and $\overline{t}_j$ correspond to the mean values of the hypothetical values and measurements, respectively. The result matrix R will contain values which may range from -1 to 1 (in practice the correlation values for a DPA attack are smaller, $e.g.$ $\leq |0.3|$). The position with the highest absolute value in R will correspond with high probability to the correct key and time instance when it is being leaked. A higher number of measurements used will yield a better match.

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \overline{h}_i) \cdot (t_{d,j} - \overline{t}_j)}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \overline{h}_i)^2 \cdot (t_{d,j} - \overline{t}_j)^2}} \tag{3.1}$$

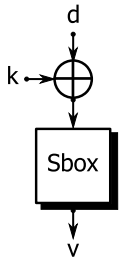


Figure 3.2: AES S-box transformation during the first round. k is an unknown, constant value, while d is a known, variable value.

Tip: More details on the DPA steps may be found in Chapter 6 of the reference book [3].

3.2 Measurements with Picoscope

To obtain the measurements of the smart card's power consumption you will be using the PC Oscilloscopes that are found in the lab. Each measurement is known as a power trace and is composed of several samples. Traces contain the side-channel information which will be used to recover the cryptographic key.
The hardware for the Smart Card emulator includes different shunt resistors. Which one is used can be configured by setting the jumpers in *JP3*, marked as "shunt resistors" in Figure 1.1. The setup for the DPA attack is shown in Figure 3.3. The measuring instrument is the USB oscilloscope "Picoscope 5204". The trigger input of the oscilloscope (*EXT*) is connected to the trigger pin of the SmartCard (*JP5*). *Channel A* of the oscilloscope should be connected to the measurement point on the card, connecting the probe tip to *JP9*. The alligator clip of the probe must be connected to the ground connection *JP8*.
To automatically record your power measurements (aka. power traces) you can use the provided measuring script "trace_measurement_v2.py". This script communicates with the terminal and sends random data to be decrypted by the card. The power traces and the decrypted values are saved to the output directory as a ".h5" file. You can modify the amount of traces, sample rate, amount of samples and output directory by modifying the script. Various constants, like the different sample rates, are defined in the file "pshelper.py". Please note that the sample window is taken from the falling edge of the trigger signal and extends backward in time. Figure 3.4 shows the duration of the sample window in comparison to the trigger signal. The hdf5 file format used to save the traces is similar to a file system and contains three datasets: plaintext, ciphertext and traces. Additionally you can choose the chunk size of each dataset, which if well calculated has the advantage to load a large amount of data very fast. You can use the "hdf5viewer" to inspect the organization of the output file. This program is already installed in all the lab computers. Please note that HDF5 uses row-major ordering to store matrices. When loading the file to MATLAB, which uses column-major ordering, the datasets will be transposed (rows are swapped for columns and vice-versa). In contrast to Matlab, Python makes use of row-major ordering and loads the tables in the correct order.

More information about the hdf5 file format:
`https://www.hdfgroup.org/HDF5/`

To start the script enter:

```
$ python trace_measurement_v2.py
```

To be able to see all ten AES rounds in the measurements, configure your script to:

```
sample_rate = psc.F_125_MHZ # sample rate [S/sec]
n_samples = 625000 # number of samples per trace
```

Later you will be attacking the last round, therefore you can limit the number of points collected by modifying the sample time to:

```
n_samples = 62500 # number of samples per trace
```

Try different resistor configurations and different sample rates to get used to taking measurements and to find out the one that suits you better.

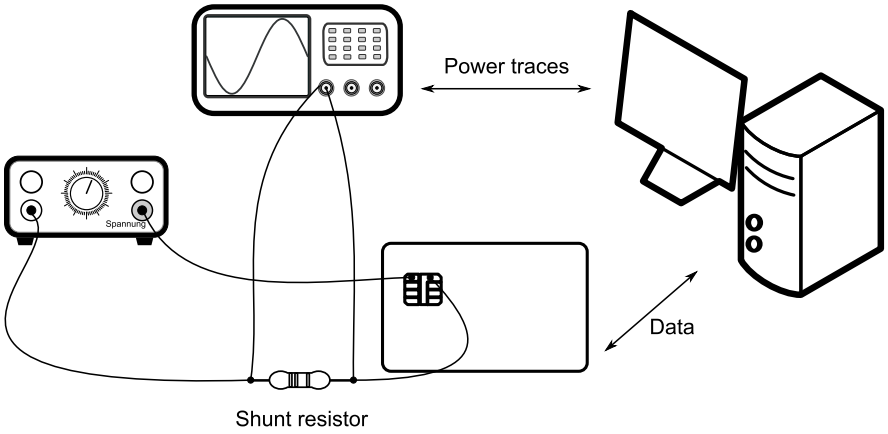


Figure 3.3: DPA Setup

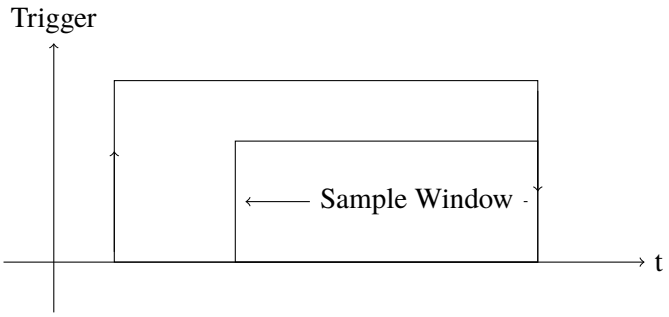


Figure 3.4: Dependency of the trigger and the sample window

3.3 Improvements to the DPA scripts

To perform a DPA attack, you often have to handle a lot of data. Therefore you should take care about the following points: trace compression and memory management.

3.3.1 Trace compression

Usually there is a high information redundancy in power traces. This means that traces may be compressed to reduce the number of points to analyze and therefore reduce the complexity of the DPA attack without loosing much information. Redundancy comes from the fact that the oscilloscope will sample the power consumption several times during a clock cycle of the Smart Card. Basic possibilities of trace compression are:[3, p. 82f]

- Keeping only the maximum value per clock cycle

- Summing (or mean) over the whole trace
- Summing (or mean) over a window

Try out different compression techniques with different window sizes and log your results to find the one that works the best for you.

3.3.2 Memory management

To handle the big amount of data (especially for Phase 2 of the lab) you have to take care of memory management.

Think about the amount of data which you are loading into the RAM while processing the power traces. Try and find a strategy to keep the RAM usage low while still being able to process all the data. One common strategy would be processing data in parts at a time and later combining the intermediate results to calculate the final result. A useful resource is:
`http://blogs.mathworks.com/loren/2014/12/03/reading-big-data-into-matlab/`

3.4 DPA countermeasures

DPA attacks work because the power consumption of the SmartCard depends on the computation of intermediate values in a cryptographic algorithm as it is run within the microcontroller. The goal of DPA countermeasures is to avoid or at least to reduce these dependencies. The two major types of countermeasures are: hiding and masking[3, p. 167].

3.4.1 Hiding

The goal of hiding is to minimize the correlation between the values of the secret data being processed and the power they consume. Basically there are two approaches to break this dependency:

- Randomize the current consumption for the operations.
- Maintain the same current consumption for each operation and each value.

Since for the lab we are not able to modify the structure of the chip, we cannot apply a countermeasure to maintain the same current consumption for different operations and values. Only software countermeasures are viable to randomize the current consumption for the operations. As a result, hiding must be performed in the time axis and not in amplitude. There are two possible options:

1. Shuffling
2. Dummy Operations

Shuffling is used to modify the time instance of the intermediate values and is performed by modifying the order in which the operations are performed. Dummy operations on the other hand inserts operations which are not part of the original algorithm in order to affect the alignment of the measurements. Both countermeasures rely on the fact that for DPA to work, the

measurements need to be aligned. In order to provide enough resistance, these countermeasures require randomization, background information on random number generators can be found in Section 4.4.
Both *shuffling* and *dummy operations* need to be implemented and tested during the second part of the lab.

3.4.2 Masking

Masking aims to make the power consumption independent of the intermediate values, by transforming the intermediate values in such a way that an attacker will not be able to predict them, and thus will not be able to compute a valid hypothesis. This is done by introducing random values which are combined with the intermediate values of the block cipher and removed at a later time. Since the real values are protected behind the cover of non predictable values we say that they are being "masked". There are different methods to apply a mask. For AES we make use of Boolean masking, where the operation used is an XOR function. We can define a masked value v' to be the result of the combination of the real intermeadiate value v and a randomly selected mask m such that:

$$v' = v \oplus m$$

Section 4.3.2 provides a deeper explanation and describes the implementation of this method.

3.5 Attacking the countermeasures

3.5.1 Attacks on hiding

There are a few approaches to attack hiding countermeasures. Firstly, it is possible that your attack script will still work by using enough traces if the random source is biased. Secondly, you can improve the attack using trace compression as a pre-processing step. And finally, you can try to re-align the power traces before performing the DPA attack. Read more about attacking hiding countermeasures in [5].

> **Tip:** Talk to your group members to understand how the countermeasures were implemented and come up with a plan to attack them.

3.5.2 Attacks on masking

The first attack to try out is a conventional DPA attack with a high number of traces ($i.e.$ up to $10,000$). An attack may still be successful if there is an error in the implementation or when there is a bias in the random numbers used for the masks. After this has been tested, the next step is to improve the attack scripts to perform what is called a *second-order attack*. Second-order DPA attacks exploit the joint leakage of two intermediate values instead of a single instance. Therefore, even in a protected implementation the secret key may be extracted. The main concept is that if two values make use of the same mask, combining them will cancel the effect of such mask. The points in the measurement as well as the hypothesis must be modified to perform this

attack. To combine the traces you will perform a pre-processing step and later you can apply DPA as usual, but with a different hypothesis. The new hypothetical intermediate values may be computed as:

$$w = (v \oplus m) \oplus (u \oplus m) = v \oplus u$$

Read more about second-order DPA attacks in Chapter 10.3 of [3].

Tip: To make sure your $2nd$-order DPA script works correctly, you can set the masks to a known fixed value. Afterwards you can test the script again using random masks.

Chapter 4: Smart Card Emulator

One of the most important tasks during the laboratory is the creation of the code for your Smart Card emulator. The emulator is used in the first part of the lab to create a clone of the reference card and in the second part to assess the strength of your countermeasures against side-channel analysis. The code for the smart card emulator for this laboratory should be composed of three main blocks which are:

- Operating System (Command Interpreter)
- Communication Interface
- Cryptographic Core

4.1 Operating System

Operating System (OS) is the software that takes care of the basic hardware functions of a device such as peripheral control and provides common services for other applications. For this laboratory the Operating System will be very simple, its functionality is limited to managing the code that takes care of configuring the hardware, interpreting the commands being received through the Communication Interface, and controlling the inputs and outputs to the Cryptographic Core. In this sense it can be thought as a *state machine*, more than an actual OS, where the behavior of the Smart Card at a given time depends on its current state.
For the basic functionality required for the lab there are three active states to be taken into account: *Power-up*, *Wait* and *Execute Command*. During Power-up the hardware is configured and the initialization routines are executed. This state is finished by sending an Answer-to-Reset to the terminal (described in Section 4.2.1). The card then waits for commands to be sent by the terminal, we call this the Wait state. Once commands are sent, the Communication Interface block is in charge of sampling the signals, transforming them into bits and concatenating these bits into each of the bytes which ultimately form the commands. Error detection and correction mechanisms should be implemented during this process to ensure correct data reception (take a look at Section 7.3 of [1]). If the command is correctly received, then the card goes into the Execute Command state where the command is decoded and executed. After completing execution the card will return to the Wait state. These states and the transitions between them are shown in Figure 4.1.

4.2 Communication Interface

Data communication occurs over the ISO7816 I/O contact on the card. Communication between the card and the terminal is asynchronous and half duplex. This means that the I/O pin is

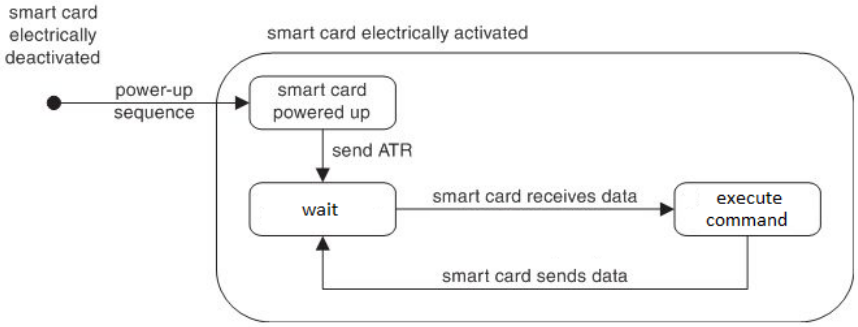


Figure 4.1: Flow chart of the Smart Card states

bidirectional and is used for both transmitting and receiving data. The electrical characteristics specified in ISO7816 for serial communication are similar to the ones used in the more widely known RS-232 standard. During the Wait state the I/O line is set to a high level using a pull-up resistor. Data transfer of a character is initiated by a start bit, which sets the I/O signal to low for one elementary-time-unit (etu). The start bit is followed by eight data bits, one parity bit and two stop bits (see Figure 4.2). The parity bit is set to high if the count of bits set to high in the data section is odd and is set to low if the count is even. The time between the stop bit and the next start bit receives the name of guard time. After the stop bit, the I/O line returns to a high level. If the receiver does not detect a parity error, it waits for the next start bit after the guard time. If an error is detected, the receiver must indicate this by setting the line low for one etu starting at half an etu of the stop bit. The transmitter must check the I/O line during the stop bit and if it is low re-send the character. The signal for a transmission error is shown in Figure 4.3.[5, p. 256].

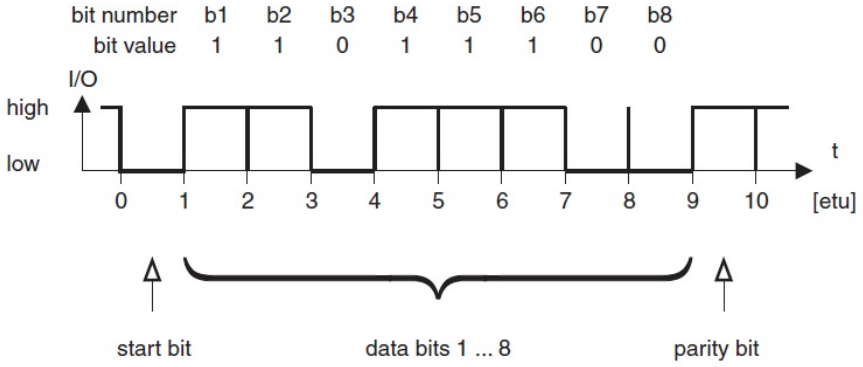


Figure 4.2: ISO 7816 character framing showing the initial character TS with the direct convention, which is indicated by the value $3B_{16} = 00111011_2$

Tip: To avoid errors caused by spurious signals, sampling may be improved using multiple samples in conjunction to decision rules. Take a look at [5, p. 247] for more information.

The duration of a bit is called an elementary-time-unit (*etu*) (see Figure 4.2).[5, p. 206]. The etu

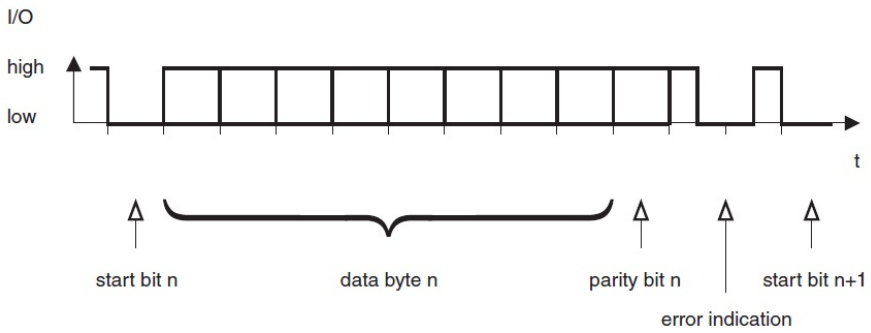


Figure 4.3: Signaling a transmission error with the T = 0 protocol by setting the I/O line low during the guard time

is equal to the count of F/D clock cycles on the CLK pin. Where F and D are the transmission parameters: F is the clock rate conversion integer and D the baud rate adjustment integer [1, p. 13]. The default values are $F = 372$ and $D = 1$.

$$1\ etu = \frac{F}{D} \cdot \frac{1}{f}$$

Tip: The communication between the card and the terminal occurs asynchronously. This means there is no shared timing signal between them. To avoid errors during sampling, synchronization between the clock within your Smart Card emulator and the communication signal is needed. Think about how you could use the start bit to do this.

4.2.1 Answer-to-Reset

After plugging in the Smart Card into the terminal, the card receives power (VCC) and a clock signal (CLK) and the controller has time for initialization. The terminal opens the communication by setting the reset signal (RST) to high. As a result, the card sends the so called "Answer to Reset" (ATR) over the I/O-pin. The ATR sequence contains information about the card and establishes the communication parameters. For the lab, we use a short ATR sequence to indicate the bit convention, clock rate division factor and bit rate adjustment factor. The characters which compose this ATR sequence are the initial character TS, the format character $T0$ and the optional interface characters TA_1 and TD_1.

- **The initial character TS**
  The TS is the first byte of the ATR and must always be sent. It specifies the convention used for the communication protocol and also contains information to determine the divisor value. There are only two codes allowed for this byte: '3B' with the direct convention (which is used in our case) and '3F' with the inverse convention. Figure 4.2 shows the timing of the bit sequence of the initial character.[5, p. 206]
- **The format character T0**
  T0 is the second byte of the ATR sequence and specifies which interface characters follow it. Equal to the initial character TS, the T0 is mandatory in the ATR. [5, p. 207] In the case of SiglTV the coding of the format character is 0x90.

- **The interface characters**
  The interface characters consist of TA_i, TB_i, TC_i and TD_i bytes and specify all the transmission parameters of the protocol. There are default values defined for all the transmission protocol parameters because these bytes are optional. In case of the ATR used in the lab, the interface characters $TA_1 = 0x11$ and $TD_1 = 0x00$ are used to complete the sequence. [5, p. 207]

Figure 4.4 shows the reset, I/O and clock signals during an ATR sequence. You should be able to recognize timing diagrams like this one using the provided logic analyzer.

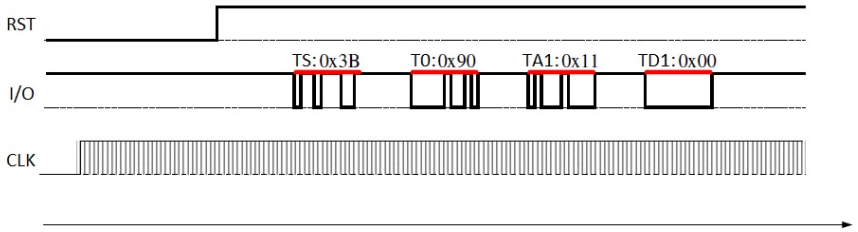


Figure 4.4: Answer to Reset

Note: If a valid ATR sequence is not received, the terminal will shut down the clock signal after a timeout period is reached. Keep in mind to use the provided "programm helper" instead of the Smart Card terminal when flashing the microcontroller.

4.2.2 T=0 Protocol

After the conclusion of the ATR, the card waits for instruction from the terminal. As shown in Figure 4.5, the command structure consists of a header containing a class byte (CLA), an instruction byte (INS) and three parameter bytes (P1 to P3) [5, p. 255]. It can be optionally followed by a data section.

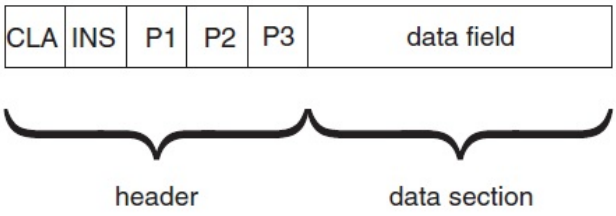


Figure 4.5: Command structure with the T = 0 protocol [5, p. 255]

After transmitting the header as a string of five bytes, the terminal waits for a procedure byte. Table 4.1 shows the three types of procedure bytes.[1, p. 23]

- If the value is '60', it is a NULL byte. It requests no action on data transfer. The interface device shall wait for a character conveying a procedure byte.[1, p. 23]
- If the value is '6X' or '9X', except for '60', it is a SW1 byte. It requests no action on data transfer. The interface device shall wait for a character conveying a SW2 byte. There is no restriction on SW2 value.[1, p. 23]
- If the value is the value of INS, apart from the values '6X' and '9X', it is an ACK byte. All remaining data bytes if any bytes remain, denoted D_i to D_n, shall be transferred subsequently. Then the interface device shall wait for a character conveying a procedure byte.[1, p. 23]
- If the value is the exclusive-or of 'FF' with the value of INS, apart from the values '6X' and '9X', it is an ACK byte. Only the next data byte if it exists, denoted D_i, shall be transferred. Then the interface device shall wait for a character conveying a procedure byte.[1, p. 23]
- Any other value is invalid.[1, p. 23]

Byte	**Value**	**Action on data transfer**	**Followed by**
NULL	'60'	No action	A procedure byte
SW1	'6X($\neq$'60'), '9X'	No action	A SW2 byte
ACK	INS	All remaining data bytes	A procedure byte
	INS $\oplus$ 'FF'	The next data byte	A procedure byte

Table 4.1: Procedure bytes [1, p. 23]

Tip: Use the *logic analyzer* to eavesdrop the communication between the reference card and the terminal to get more information about the instructions and the protocol (cf. Section 1.3.4 for information on how to configure the logic analyzer).

4.3 Cryptographic Core

The Advanced Encryption Standard (AES) is the cryptographic algorithm used in the laboratory. This block takes as input the ciphertext corresponding to the encrypted chunk-key k_{cc}, decrypts it using the master-key k_m and returns the plaintext chunk-key k_c. Therefore you only need to implement the *decryption* function. Specifically, decryption using AES-128 in ECB mode.

4.3.1 Basic AES Implementation

The AES decryption block takes as inputs 16 bytes corresponding to the ciphertext and the 16-byte master key, and returns as output the 16-byte plaintext. Within the decryption block bytes are arranged in a state matrix, as shown in Figure 4.6. The state matrix is updated by applying several transformations. Figure 4.7 shows the block diagrams which describe the order of transformations used during encryption (a) and decryption (b) for AES. The details of each transformation can be found in [4].

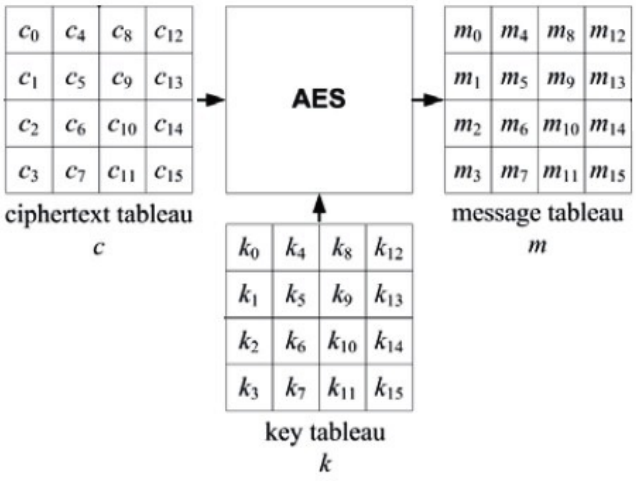


Figure 4.6: Inputs and output for an AES decryption

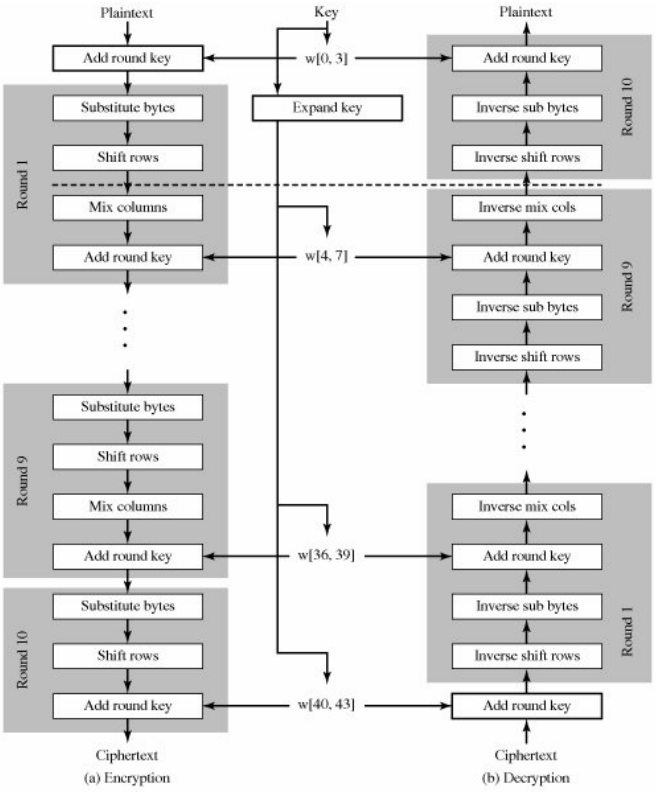


Figure 4.7: Block diagram of the AES encryption and decryption

Debugging your AES implementation may be complicated without a reference. Therefore, one of the most common ways to test the correctness of the implementations of cryptographic algorithms is making use of test vectors and compare inputs and outputs with known values. You can obtain

the vectors for AES from the NIST website:
Test Vectors for AES (ECB Mode)
`http://csrc.nist.gov/groups/STM/cavp/documents/aes/KAT_AES.zip`

> **Tip:** Set the trigger pin $JP5$ ($PB4$ in the AVR) when an AES decryption starts and clear it once it finishes. This pin is used to align the power traces while taking the measurements with the oscilloscope and to measure the run-time duration of your algorithm with help of the logic analyzer.

4.3.2 Hardened AES Implementation

In this section two basic techniques to protect the AES are discussed and an introduction to random numbers is given.

Hiding

As mentioned in Section 3.4.1, the main possibilities to apply hiding in software are restricted to randomizing the current consumption over the time axis. Shuffling executes the critical operations randomly generated order and dummy operations moves the execution time of the critical operations over the time axis by inserting additional operations. More information can be found in [3, p. 167]

Masking

For the implementation of masking, you need six independent random variables for the masks m, m', m_1, m_2, m_3 and m_4. Figure 4.8 shows which masks are applied on which state matrices in an AES encryption process. The masks m and m' are the input and output masks for the SubBytes operation and the masks m_1, m_2, m_3 and m_4 are the input masks for the MixColumns operation. Therefore the two following precomputations are needed [3, p. 229ff]:

- Masked S-box table S_m such that $S_m(x \oplus m) = S(x) \oplus m'$.
- Output masks for the MixColums operation by applying this operation to (m_1, m_2, m_3, m_4) to generate the masks m'_1, m'_2, m'_3 and m'_4.

Please take a look at [3, p. 229] to get a better understanding of this masking algorithm.

4.4 Random numbers generator

The random source is an important element of the implementation of countermeasures. Therefore this section gives an introduction to random numbers. Following random number application exists:

- **Pseudorandom Numbers (PRN)**
  Deterministically generated sequence which cannot be discriminated from true random numbers by several statistical tests. In general repeatable, not necessarily unpredictable.

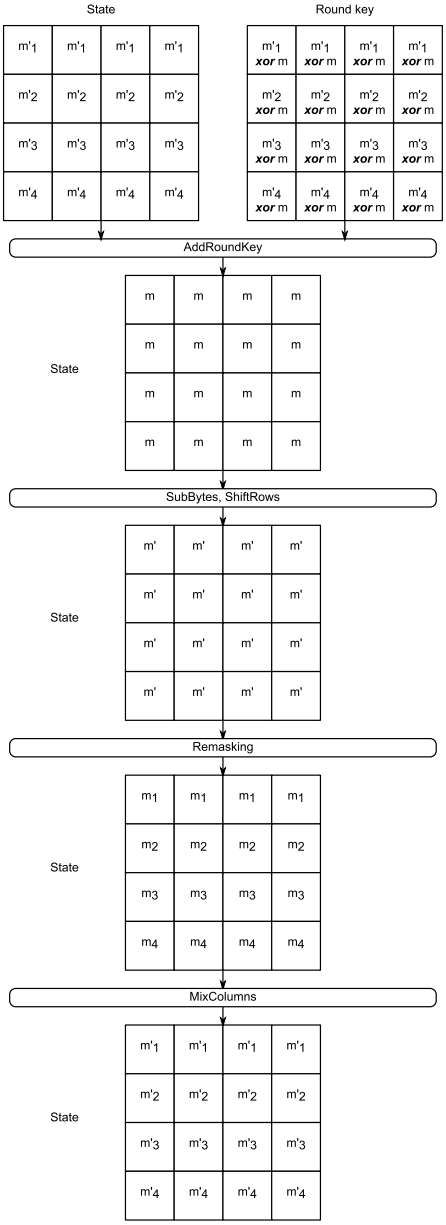


Figure 4.8: AES states with the different masks [3, p. 230]

- **True Random Numbers (TRN)**
  Sequences generated from physical random processes. In general unpredictable and unrepeatable.
- **Cryptographically Secure Pseudorandom Numbers (CSPRN)**
  Unpredictability is required, not necessarily unrepeatability.

The general structure of cryptographic random number generators is shown in Figure 4.9. A reliable noise source is the basis for a TRNG. The output of the noise source is digitized and post-processed to produce a balanced distribution of "0" and "1" bits ($e.g.$ making use of an XOR or a Von Neumann corrector). The output after post-processing is a random bit sequence which may be used directly. However, since this process may be computationally intensive, instead of generating all the random numbers this way, a hybrid approach is followed. The TRNG is used at start up and a PRNG is seeded with the random bit sequence. Subsequent values are obtained at the output of the PRNG.
There are different options which may be used generate random numbers. You are encouraged to search and compare solutions to find the one that better suits your needs. A good reference to get you started is [2].

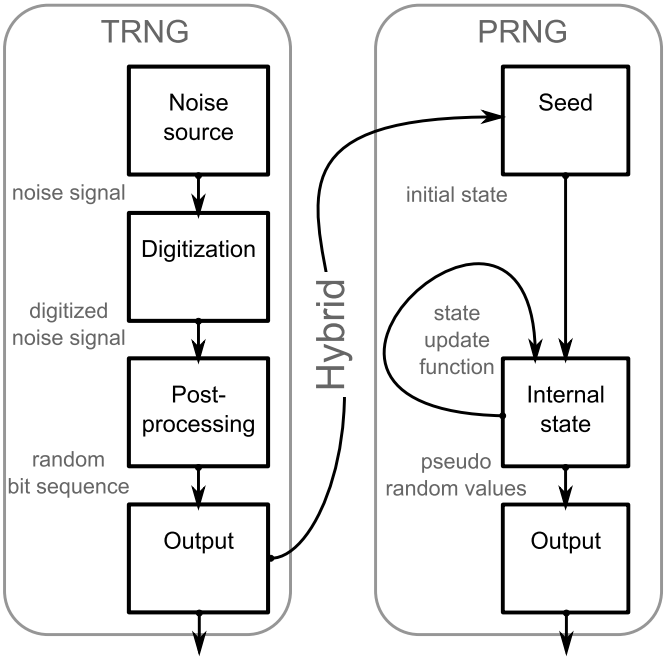


Figure 4.9: General structure of Cryptographic RNGs

Bibliography

[1] ISO/IEC. 7816-3:2006 - identification cards – integrated circuit cards – part 3: Cards with contacts – electrical interface and transmission protocols, 2006.

[2] Benjamin Jun and Paul Kocher. The intel random number generator. *Cryptography Research Inc. white paper*, 1999.

[3] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks*. Springer, 2007.

[4] NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197:441–0311, 2001.

[5] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. John Wiley and Sons, 2010.

Acknowledgements

Thanks to Thomas Zeschg, tutor during the semesters SoSe15, WiSe15/16, and SoSe16, for his help during the creation and improvement of the laboratory script.