Technische Universität München
TUM School of Computation, Information and Technology
Chair of Sicherheit in Information Technology

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: *tim.music@tum.de*

2015-2016 – Oscar M. Guillén Hernández

2016-2017 – Martha Johanna Sepulveda Florez

2017-2021 – Michael Gruber

2021-now – Tim Music

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 several phases, which you as a team of two will work on. Depending on the tasks you are working on, you either put on a "black hat" and become the attacker or a "white hat" and be a security analyst. As the attacker, your goal is to extract the secret key stored within the device by attacking the implementation using a Differential Power Analysis (DPA). Later you use the extracted key to clone the original card by programming your own Smart Card emulator. As the security analyst you implement, test and improve different countermeasures aimed to protect the Smart Card that you programmed. You test the resistance of the provided reference card, as well as your own implemented countermeasures by improving the attack scripts. Whenever working on security improvements, 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 Milestones

This laboratory consists out of six work phases with two major milestones.

Phase 1 - Theoretical Background

Get familiar with the hardware and theoretical background. Read up on the ISO7816 protocol, AES block cipher, STM32F051 microcontroller, debugger and logic analyzer hardware. This phase also includes the introductory lectures.

Phase 2 - Analysis

Basic DPA implementation and validation on prerecorded traces. Later on perform measurements on the reference card and infer the key.

Phase 3 - Clone

Implement a smart card operating system supporting all necessary basic commands. Development can be aided by using the provided protocol specification, communication scripts from PC side and logic analyzer traces from the reference card communication. Further, implement the AES decryption, validate it with the test vectors provided and integrate it into the smart card OS. Please choose to optimize your smart card implementation either on code size or speed.

Milestone 1 - Intermediate Presentation

The fist major milestone will close all phases before. During the intermediate presentation, all groups will be required to be present in the student room, ready to showcase the work achieved so far. Please prepare some plots and benchmarks from your DPA, as well as code size and performance from your smart card OS. And of course, we will watch the SiglTV video with your cloned card.

Phase 4 - Hardening of the AES and DPA Improvements

As the reference implementation has been broken by the DPA, it is time to target one of the reference card's hardened AES. Starting with hiding countermeasures, you are to improve your DPA implementation. Then, implement your own hiding countermeasures and assess their effectiveness.

Phase 5 - Hardening of the AES and DPA Improvements II

Another way to harden AES is to employ masking. Implement masking on your smart card and show that even with a high number of traces you can not break the reference card's masked implementation. In contrary, how does this change if you select one of the implementations with biased masks? Then, implement your own masking and verify that you can not break it with a first order DPA. Finally, pick an AES key at will, flash the clone card with your masked implementation and hand the hardware back to the department. If it does not break within a certain number of traces, you will receive a bonus point. *Optionally: If you wish, you can implement a second order DPA and show its effectiveness. However, this is outside the scope of this lab.*

Phase 6 - Lab Report

Per team, write a short laboratory report, with a limit of four PDF pages in which you shortly outline the following:

1. The performance of your DPA and your thoughts you put into the implementation. Also any trace preprocessing done.

2. The workings of your smart card OS, state machine graph, program size and speed of all AES implementations. Also include a comparison to the reference card.
3. Which aspects of the DPA did you improve, how did the runtime change for various levels of AES hardening. How many traces were required to break the implementations in the end.
4. Shortly, how the countermeasures in your AES have been implemented and the overall overhead in comparison to the unprotected one.

Please include a rough time plan and mark which team member worked on each task. The report is to be handed in roughly one week before the final exam, details will be in Moodle.

Milestone 2 - Oral Exam

During the oral exam you are going to be asked questions based on general theoretical background from the lecture slides and the work you described in your lab report.

1.2.2 Laboratory hours

You are free to work on the lab tasks where and when you please. The laboratory room **N2947** 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 Equipment

For the completion of the lab assignments your team will receive the following hardware kit.

- Two Smart Card
- Programming and Debug Helper (PDH)
- Saleae 8-channel Logic Analyzer

In addition to the hardware your repository provides you a set of Python scripts. Further, you will have access to several tools already preinstalled on the lab computers. This section aims to provide a reference on how to make the most out of this.

1.3.1 Smart Card Hardware

The two SCs are the central component of the lab and are centered around a modern Cortex-M0 microcontroller by ST Microelectronics, the STM32F51. Selected controller features 64 KB of flash memory, as well as 8 KB of SRAM, giving you plenty of room for your implementations.

The first Smart Card (SC) 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. The figure Fig. 1.1 shows the top side of the provided hardware.

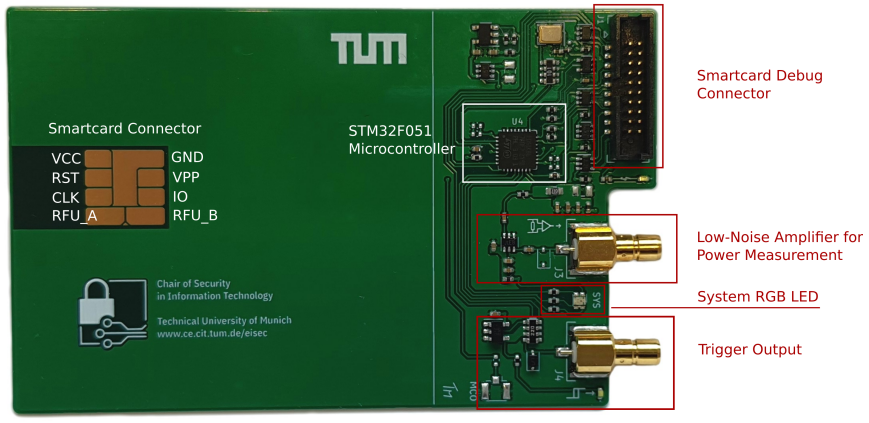

Figure 1.1: Smart Card: Annotated Front View

SC Debug Connector

In Fig. 1.1 on the upper right, the SC connector can be seen. Typically the SC is connected to its debugger through a 20-pin IDC cable. This connection serves multiple purposes, such as:

Card Power The connector allows supplying the SC with power independently of the SC reader. As soon as the SC debug cable connects the SC to the PDH, the power switch on the PDH takes precedence. For more information please see Section 1.3.2.

Serial Wire Debug Interface The microcontroller exposes a Serial Wire Debug (SWD) interface through which firmware may be uploaded and the ARM-Cortex debugged. For more information please see Section 1.3.2.

Debug UART Interface If additional debug output is required, the card exposes one of its UART interfaces onto the connector.

Smart Card Terminal Pins & Trigger Pin For reverse engineering the reference card, the SC terminal pins (left side of Fig. 1.1), as well as the trigger pin are directly routed onto the SC debug connector and can be comfortably accessed on the PDH's pin headers.

Power Measurement

In the process of performing a DPA, it is crucial to measure the controller's power consumption. Selecting a suitable shunt resistor is a sensitive topic. Dimensioning its resistance too high results in a high voltage drop, leading to brown-out effects on the chip. If the value is too low results in a low measurable voltage, decreasing the SNR by using less of the oscilloscope's dynamic range.

A sensible solution is to use a wideband low-noise amplifier, such as the BGA2801, which is internally matched to 50 Ohms and may directly drive an oscilloscope input. This way, a smaller shunt may be placed, whilst at the same time utilizing most of the oscilloscopes dynamic range. Explicitly for the current shunt configuration of the smart cards, the average 25 mV shunt voltage drop is amplified by around 23 dB to 300 mV, to be then measured by the picoscope.

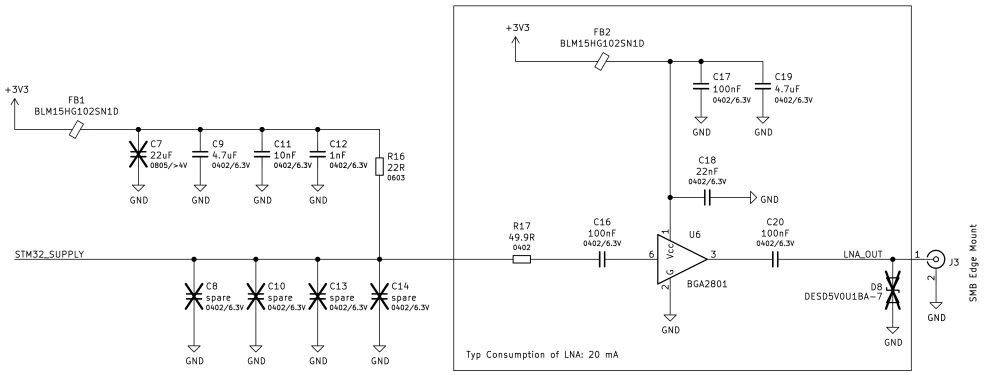

Figure 1.2: Smart Card Power Measurement Schematic

Clock Source

The microcontroller on the SC is able to derive its processor clock from a variety of sources. For the scope of this lab we consider three relevant clock sources that are selected through several multiplexers. In addition to the brief explanations in this lab manual, please consider the *RCC - Reset and Clock Control* chapter of the STM32F051's datasheet and reference manual.

The first clock source is the internal 8 MHz oscillator, named *HSI*. After each system reset, the processor core clock is driven by it. Due to its nature, this oscillator is not very accurate and generally drifts quite substantially over temperature, making it unsuitable for most communication aspects.

The second and third clock source both are being externally fed into the microcontroller through its external clock input pin *PF0*, driving the so called *HSE*. If the *CLKSEL* GPIO is asserted with a logic high, external crystal oscillator drives supplies an accurate 8 MHz clock. In the case of a system reset, the *CLKSEL* pin is high impedance, the pull-up resistor R1 automatically enables this configuration. Finally, pulling *CLKSEL* low, the clock supplied by the card reader can be routed onto the microcontroller's clock input.

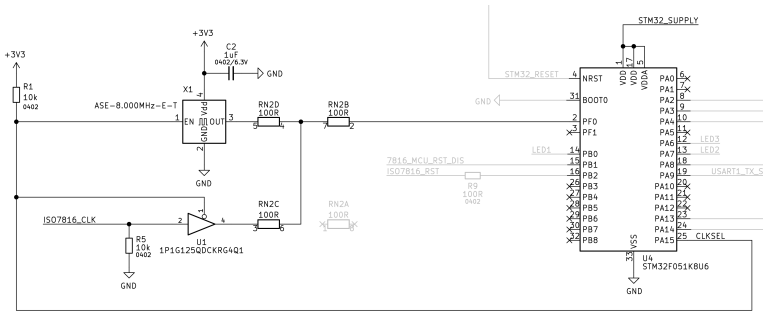

Figure 1.3: Smart Card: External Clock Circuit

Inside the SC demo program inside your repository, the function `void sys_set_clksource(uint8_t src)` can be used to comfortably switch between the 8 MHz crystal oscillator and the card reader

clock. Due to the unstable nature of the *HSI* it is not intended to use it inside the application, and thus there is no functionality provided to switch back to it.

Even though most of the clock tree configuration and switching procedure is already provided to you, there are some caveats you should keep in mind, especially while debugging the SC.

The card reader disables the card clock after a few seconds if no further transaction is following. This means, your smartcard will become unresponsive and the debugger connection will break. If you have not yet implemented the ATR yet, do not expect to get a clock signal for more than a few milliseconds. In most cases using the external crystal oscillator clock is your best bet.

By specification, the card reader can supply any clock frequency it desires, in a range of 4 MHz to 8 MHz. As the ISO7816's USART bit duration (ETU) is derived from the clock, the only way you can be certain to exactly match the baud rate of reader and SC, is to use the card reader clock.

The SC clock output by the reader has a varying duty cycle and overall high amounts of jitter. Even though you want to secure the SC also by several countermeasures, do your DPA team a favor and utilize the crystal oscillator clock whilst computing the AES. This helps align the traces, facilitating the DPA measurements.

Reset Source

The SC controller's reset behavior is controlled by the circuit in Fig. 1.4, supporting two reset sources. Having the *STM32_RESET* net exposed on the SC debug connector, any attached debugger may unconditionally trigger a system reset.

Additionally, a system reset can be conditionally triggered by pulling the *ISO7816_RST* signal to low, which is controlled by the card reader. By default, the gate of *Q2A* is pulled high, disabling this reset path. If this behavior is not desired anymore, e.g., the card reader is allowed to reset the card, the *7816_MCU_RST_DIS* signal can be pulled low. This forces *Q2A* into a low-impedance state, relaying the card reader's reset line to affect the controller's reset pin. During debugging it is generally advised to keep the *ISO7816_RST* signal isolated and to read its state through the provided GPIO pin. Depending on your programming style it can be feasible to have this reset path activated.

The demo program provided to you will readily disable the reset functionality as one of the first functions in the main routine. This is done by using the function `sys_set_reset_source(...)`.

Recommendations (aka how to make your life easier)

- You can find the schematics, as well as the data sheet plus reference manual of the microcontroller in the lab materials in moodle. Before you start the lab, take some time to get familiar with the hardware.
- The card readers used in this lab output a fixed frequency of around 4.75 MHz, allowing you to calculate the suitable baud rate divider whilst running on the crystal oscillator clock. This allows you to have a stable communication. For the scope of this lab, no points are deducted in any way, if you only use the crystal oscillator clock.

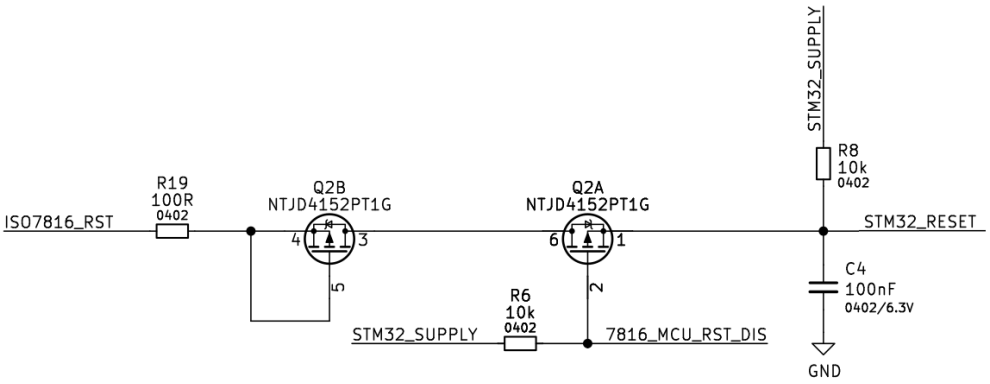

Figure 1.4: Smart Card External Reset Circuit

- Do utilize the PDH to be independent of the card reader's supply during development of your SC operating system. Generally, once the communication has ended, the card reader only keeps the SC active for a few more seconds.
- Do decide, whether you want the card reader to be able to reset the microcontroller. Often during debugging you have an easier time querying the *ISO7816_RST_SNS* pin and putting your state machine in the desired reset state.

1.3.2 Programming and Debug Helper

The PDH does what its name implies. It is the centerpiece for developing the SC operating system, connecting the SC, PC and potentially the logic analyzer. When connected to the PC, the PDH registers as an ST-Link debug adapter and is later used by openOCD to establish a connection to the target microcontroller. Its status is indicated by a set of LEDs on its left PCB edge.

Connectors

The top 20-pin connector, as seen in Fig. 1.5 connects the SC to the debugger, establishing a connection through a provided 20-pin IDC cable. Most signals running through the connector, are conveniently exposed on the pin header bar, spanning the PDH's right PCB side. It is designed for easily attaching the logic analyzer to sniff the communication during the firmware development phase, as well as for benchmarking purposes.

The signals exposed are the following ones:

- Signals from the SC chip card connector (VCC, RST, CLK, VPP, IO, C4, C8, GND)
- Trigger Signal
- Debug UART interface
- SWD Debug interface

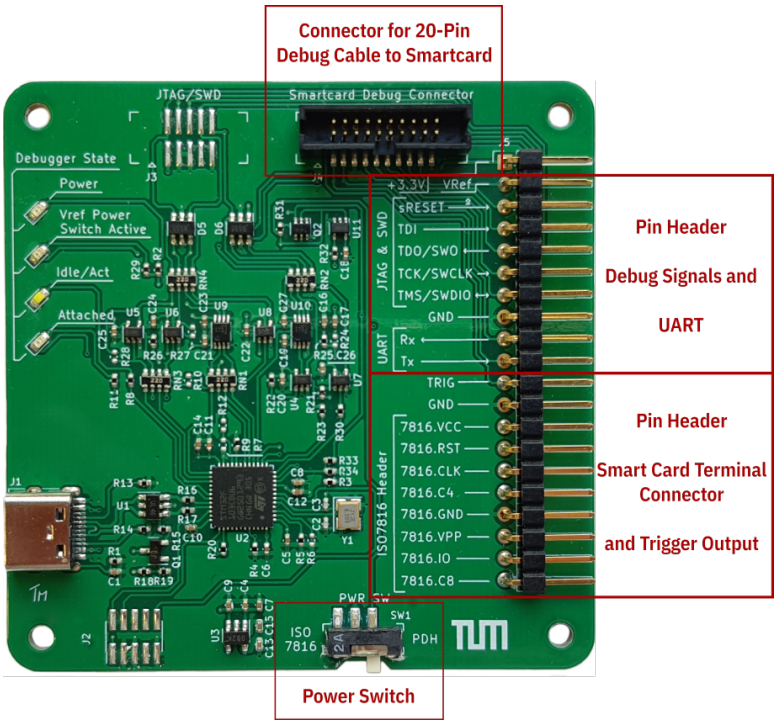

Figure 1.5: Programming and Debug Helper: Annotated Front View

Power Switch

Having a steady power supply is the base for a good debug setup. Whilst the ISO7816 specification supplies detailed card activation and deactivation procedures, these may become somewhat of a general annoyance during firmware development. Out of safety, if a SC is unresponsive, it is disabled after few clock cycles, resulting in the reader pulling CLK, RST and VDD low. As soon as this happens, the debugging session will crash.

To keep the debugging session active and the card alive, these three signals had been given special care. Most of the time your card should be independent of the card reader clock, as the internal crystal oscillator can be used. At the same time the card reader reset can be gated, the only relevant signal remaining is the power supply.

Allowing the card to be supplied steadily, the PDH contains a power switch, as seen on the bottom of Fig. 1.5. In its left position, the PDH does not influence the SC's power supply. This means the card reader supplies the power for its operation. In its right position, the SC's on-board voltage regulator is unused. Its 3.3 V output is cut off and the PDH's 3.3 V supply voltage used instead. An active power switch is also indicated by the *Vref Power Switch Active* LED on the top left.

1.3.3 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. For

visualizing the waveforms the software suite, either the vendor-software *logic* or *pulseview* (part of the open source suite *sigrok*) can be chosen.

In case you are working in our student labs, *pulseview* is already installed on each computer. If you wish to use the *logic* suite, you may download the 64-bit latest version from the vendor's webpage (`http://downloads.saleae.com`).

In order to see the data values on the card's IO line in addition to just the pure waveform, use the built-in protocol decoders. With the known frequency of the card reader clock of around 4.75 MHz, the baudrate can be determined.

- Async Serial Analyzer (*logic*) / UART (*pulseview*)
- Baudrate: 4.75 MHz / 372 clocks = 12900 (bit/sec)
  values between 12700 and 12900 work well
- Data Length: 8 bits + 1 Parity (even)
- 1.5 stop bits
- LSB first
- Non inverted

If you have not used a logic analyzer before, take some time to familiarize yourself with the hardware. Always a good starting point is the documentation found on the Saleae website[1] or in the *sigrok* wiki[2][3].

1.3.4 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 and a short search on the web will yield several cheat sheets. 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
  ```

[1] `https://www.saleae.com/`
[2] `https://sigrok.org/wiki/Getting_started_with_a_logic_analyzer`
[3] `https://sigrok.org/wiki/Protocol_decoders`

- 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
  ```

1.3.5 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. Please note that only the reference implementation on your reference card is compatible with the siglTV video feed. To verify the key for the protected AES versions, use the test vectors.

- 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 are 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
  ```
  ```

1.4 Smart Card Firmware Development

This section will give a short introduction into how to develop the firmware for your smartcard, necessary associated libraries as well as how to program it onto the card.

1.4.1 Building the Firmware

Developing firmware for the SC's Cortex-M0 microcontroller requires a library which provides the necessary hardware support. This is where the libopencm3 project comes into play. It is a fully open-source lowlevel hardware library for mainly ARM Cortex-M3 microcontroller, further supporting a range of M0 and M4 cores. This way, all processor core specific header files, platform peripheral drivers (USART, SPI, timers, ...), linker scripts and a Makefile template are given. For your SC operating system, you can decide to either use as many libopencm3 lowlevel drivers[4] or directly access hardware registers, as named in the microcontroller's reference manual, CMSIS and libopencm3 headers.

Your git repository already embeds the libopencm3 library as a git submodule and put it into the `demo_os_stm32f051/libopencm3` directory. In the toplevel directory of your repository follow these steps to clone the library before continuing:

1. Initialize the submodule:
   ```
   $ git submodule init
   ```
2. As soon as the submodule is initialized, fetch it:
   ```
   $ git submodule update
   ```
3. Build the necessary libopencm3 libraries for the stm32f0 to be linked to your binaries:
   ```
   $ cd demo_os_stm32f051/libopencm3
   $ make TARGETS=stm32/f0
   ```

To compile your code, the ARM GNU toolchain needs to be installed on your development systems. If you are using the student PC's in the lab rooms, it is already pre-installed. For your own computers you can either install the toolchain through your distribution's package manager or download the pre-built packages from ARM directly.[5]

At this point you should be able to compile the demo firmware by calling *make* in the `demo_os_stm32f051` directory. This call should generate a suitable linker script, put all object files in a build subdirectory and yield a *demo_os_smartcard.elf* and *demo_os_smartcard.bin*.

1.4.2 Debugging the Firmware

As soon as the firmware has been built, testing on the SC can begin. The ARM-compatible version of the popular GNU debugger (GDB) loads the compiled binary, but needs another tool to access the microcontroller. The following section gives a short overview how to use them in this lab. All features you probably will be using are solely scratching the surface of the

[4] See the `libopencm3/lib/stm32` and `libopencm3/include/.../stm32` directories
[5] `https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain`

tools' capabilities, so if you are looking for an extended set of features, make sure to check their documentation.

OpenOCD

The Open On-Chip Debugger (openOCD) aims to provide debugging, in-system programming and boundary-scan testing for embedded target devices. It utilizes the PDH as a debug adapter to connect to the target microcontroller. For the scope of this laboratory, you should not need to change any configuration parameters in the openOCD scripts. The user manual provided can give you some guidelines if some problems should surface.[6]

If you work on the university's student PCs, please use the openOCD binary from the TUM SEC NAS, as the one shipped by Ubuntu may be too old for your debugger firmware. To use the latest release, enter the following commands on each console you intend to use openOCD.

```
module use /nas/sec/tools/modulefiles
module load openocd/latest
```

In the `demo_os_stm32f051` directory inside your git repository, you find a folder called scripts. This folder contains two different scripts, named `openocd.sh` and `openocd_flash.sh`.

In advance of each debugging session, the openOCD GDB server needs to be started in a separate terminal. If the SC is connected and powered, the following output should appear. This indicates, that openOCD has detected the Cortex-M0 processor and is now listening for a GDB connection on localhost port 3333.

```
$ ./openocd.sh
Open On-Chip Debugger 0.12.0-rc1
Licensed under GNU GPL v2
For bug reports, read
       http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd".
      To override use 'transport␣select␣<transport>'.
Info : The selected transport took over low-level target control.
      The results might differ compared to plain JTAG/SWD
DEPRECATED! use 'adapter␣speed' not 'adapter_khz'
adapter speed: 8000 kHz

srst_only separate srst_nogate srst_open_drain connect_deassert_srst

Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 8000 kHz
Info : STLINK V2J42M27 (API v2) VID:PID 0483:3752
Info : Target voltage: 3.223265
Info : [stm32f0x.cpu] Cortex-M0 r0p0 processor detected
Info : [stm32f0x.cpu] target has 4 breakpoints, 2 watchpoints
Info : starting gdb server for stm32f0x.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

[6] `https://openocd.org/doc/pdf/openocd.pdf`

Should the processor not be found, openOCD will exit with an error message. In this case, verify your cable connection, as well as the power supply. Please re-read the SC hardware Section 1.3.1 and the recommendations Section 1.3.1.

GDB

The moment openOCD has attached to the target, you may start GDB to load and start debugging the program. If you have never used GDB for debugging before, there are plenty of resources including specific examples for ARM controllers, on the web. Debugging with GDB together with its terminal UI (TUI) is quite generic, so this section will be kept brief.

In the same scripts directory as for openOCD are two scripts named `debug_openocd.sh` and `flash_openocd_stlink.sh`. For a standard debugging session, use the `debug_openocd.sh`, which is explained in the following. The script will prefer using the `arm-none-eabi-gdb` binary or fall back to `gdb-multiarch`.

In order to always use an up-to-date binary, make is invoked before the debug session If it terminates without an error, GDB is launched and receives the following parameters:

- Start GDB with the terminal UI active *-tui*
- The binary to use, in this case *demo_os_smartcard.elf*
- After reading the binary, execute the commands in *-x ./scripts/gdb_commands_dbg*

By default, the *gdb_commands_dbg* looks as follows: At first GDB is told to connect to the openOCD's debug server on port 3333 (line 1). Then, the target microcontroller is forced into reset (line 2). The load command erases the device's flash, extracts all relevant sections from the ELF binary and programs it to the appropriate sections (line 3). A breakpoint is then set at the beginning of the main function (line 4). Lastly, execution of the program is started by a continue instruction (line 5). Using such GDB scripts is advantageous, as you do not need to manually type in the commands for each invocation of a debugging section.

```
1 target remote localhost:3333
2 monitor reset halt
3 load
4 b main
5 c
```

Listing 1.1: GDB Commands for basic debugging

Over the course of your debugging adventures, you may want to modify this GDB script. If you would, for example, like to not break at the start of the main function, remove or comment the line (# is the comment character). Alternatively to supplying the *break* command with a function name, the file name together with the line number can also be used, e.g. *break main.c:111*. Please see the GDB manual for more information.[7] Some commands like *continue* or *break* have short aliases, such as *b* or *c*. Any other commands can be shortened as long as the resulting abbreviation is unambiguous.

[7] `https://www.sourceware.org/gdb/documentation/`

The other `gdb_flash.sh` script works familiar, but skips enabling the terminal UI, not setting any breakpoints and detaching the debugger after starting the program. In case you do not want to debug right now, it may be simpler to just use the `openocd_flash.sh` script, which does not require a running openOCD session.

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 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

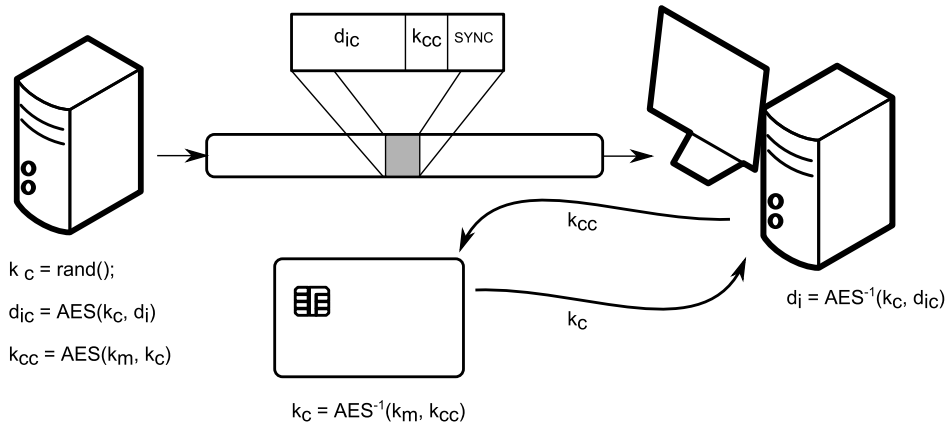

Figure 2.1: Structure of the PayTV-System

  - 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]

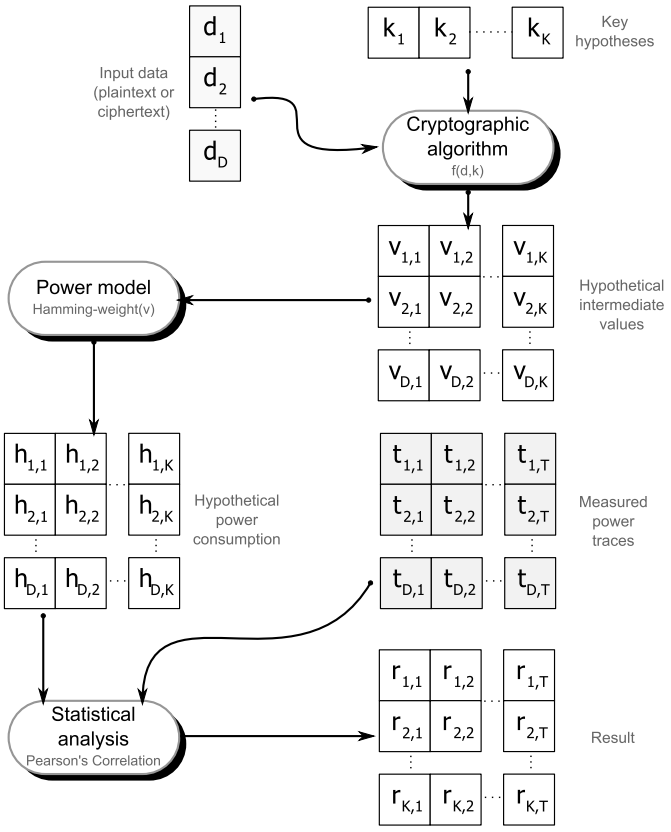

Figure 3.1: Block diagram of the steps of a DPA attack. The blocks in light gray correspond to the inputs to your script. The blocks in white correspond to the values which are generated within the script.

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.

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

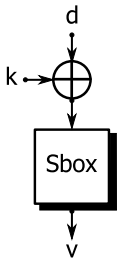

Figure 3.2: AES S-box transformation during the first round. k is an unknown, constant value, while d is a known, variable value.

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

3.2 Measurements with the 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. A conceptual setup for the DPA attack is shown in Fig. 3.3.

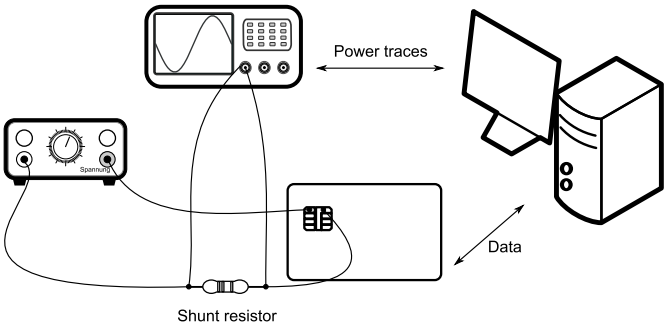

Figure 3.3: DPA Setup

The hardware for the Smart Card emulator includes a 22 Ohms shunt resistor. Its voltage drop is then amplified through a low-noise amplifier and accessible on an SMB coaxial connector. See Fig. 1.2 for the schematic. Use one of the BNC to SMB cables from your hardware set to connect this output to the oscilloscope's Channel A input. To capture each decryption at the same time, which is essential for a DPA, the firmware needs to set the trigger pin at the start of each decryption and reset it when finished. Do connect the trigger output with another BNC to SMB cable to the oscilloscope's external trigger input, named EXT.

To automatically record your power measurements (i.e., power traces) you can use the provided measuring script `scripts/measurement/trace_measurement_picosdk.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 backwards in time. The Fig. 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 quickly. You can use the "hdf5viewer" to inspect the organization of the output file. This program is already installed on all lab computers. Please note that HDF5 uses row-major ordering to store matrices, which matches the behavior of Python, loading 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_picosdk.py
```

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

```
sample_rate = psc.F_250_MHZ # Sample rate [S/sec]
n_samples = 1250000 # 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 sample rates and record lengths to get used to taking measurements and to find out the one that suits you best.

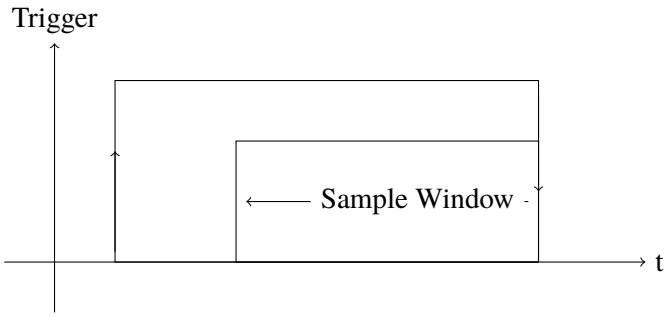

Figure 3.4: Dependency of the trigger and the sample window

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.

3.4 DPA countermeasures

DPA attacks work because the power consumption of the SC 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.

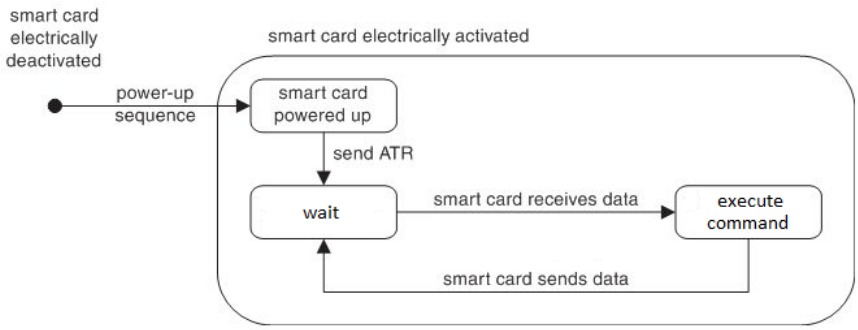

Figure 4.1: Flow chart of the Smart Card states

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 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].

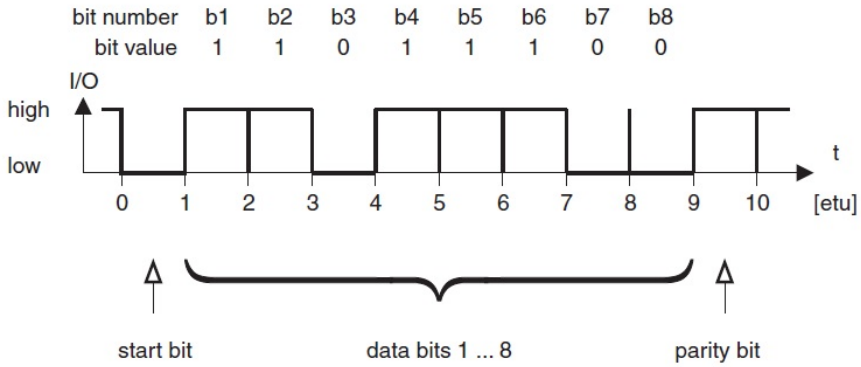

Figure 4.2: ISO 7816 character framing showing the initial character TS with the direct convention, which is indicated by the value $3B_{16} = 00111011_2$

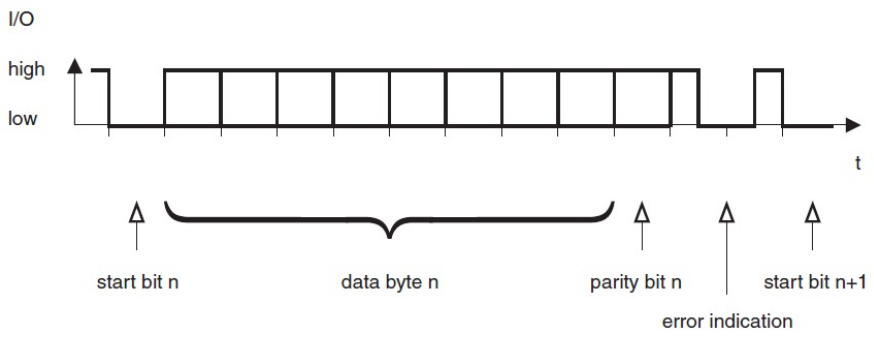

Figure 4.3: Signaling a transmission error with the T = 0 protocol by setting the I/O line low during the guard time

> **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 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.

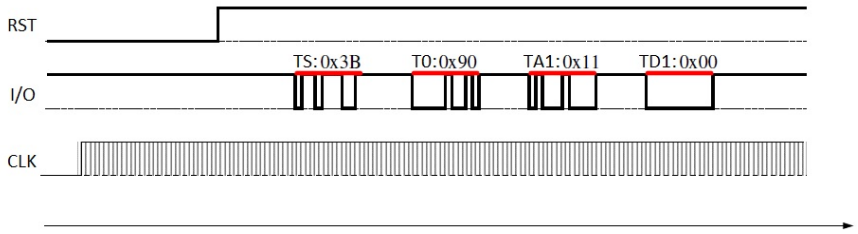

Figure 4.4: Answer to Reset

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.

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]

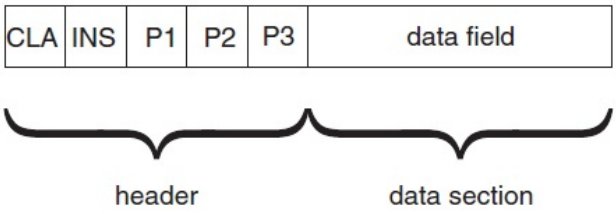

Figure 4.5: Command structure with the T = 0 protocol [5, p. 255]

- 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 1.3.3 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

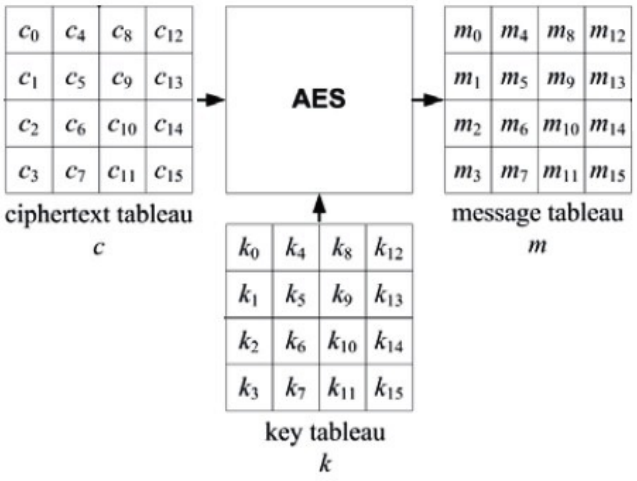

Figure 4.6: Inputs and output for an AES decryption

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].

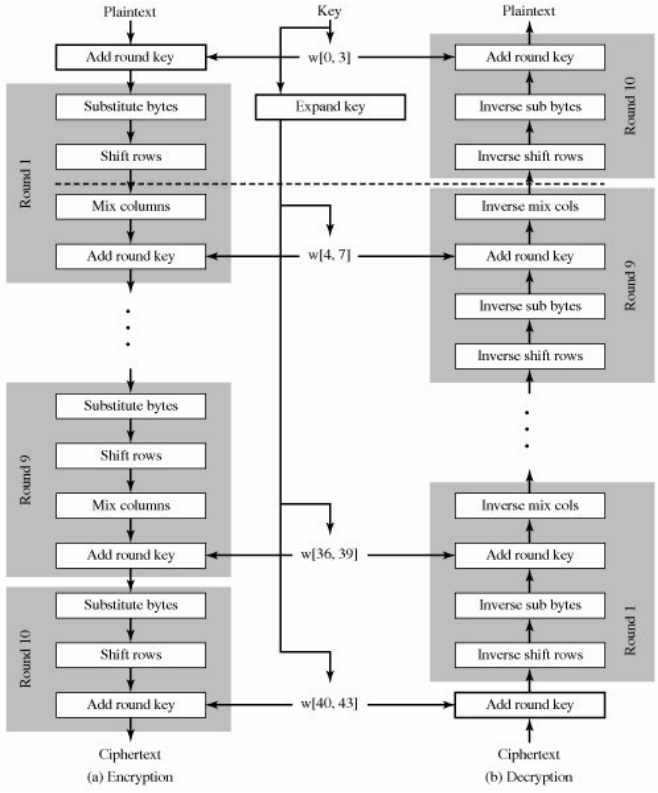

Figure 4.7: Block diagram of the AES encryption and decryption

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 output 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.

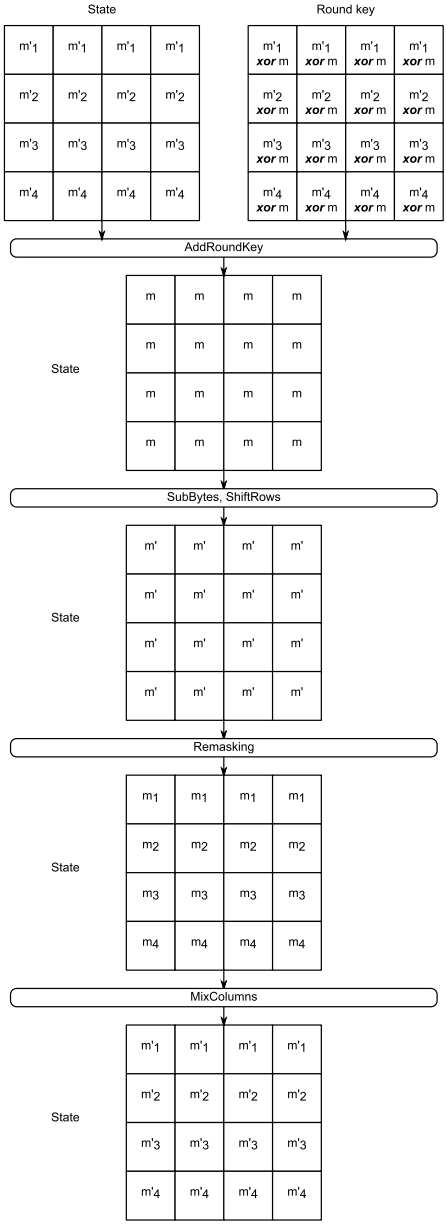

Figure 4.8: AES states with the different masks [3, p. 230]

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.
- **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].

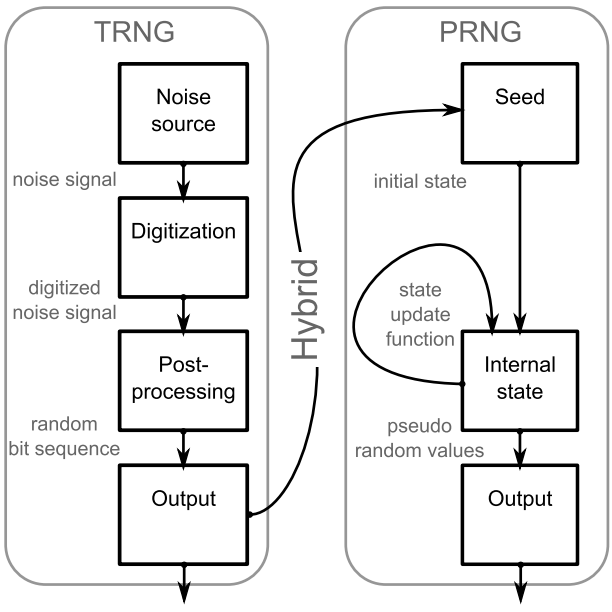

Figure 4.9: General structure of Cryptographic RNGs

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.