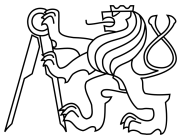
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Software Countermeasures Against Differential Power Analysis
Student:	Bc. Alena Sochůrková
Supervisor:	Ing. Jiří Buček
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2017/18

Instructions

Study existing software countermeasures against Differential Power Analysis (DPA) for the AES-128 algorithm. Implement AES-128 for encryption in the ECB mode on a smart card (Atmel AVR microcontroller) and use DPA to acquire the secret key. Design, implement, and test several countermeasures and assess their efficiency and quality based on increased difficulty of the original attack.

References

Will be provided by the supervisor.

L.S.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 23, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Master's thesis

Software Countermeasures Against Differential Power Analysis

Bc. Alena Sochůrková

Supervisor: Ing. Jiří Buček

6th May 2016

Acknowledgements

I would like to thank my supervisor for the introduction to this interesting topic and his help. I would also like to thank my partner, family and friends for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 6th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Sochůrková, Alena. *Software Countermeasures Against Differential Power Analysis.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Tato práce se zabývá možnými programovými prostředky k ochraně proti diferenciální odběrové analýze algoritmu AES-128 na AVR mikrokontroleru. Implementace několika možných protiopatření je porovnána s nechráněnou verzí z pohledu bezpečnosti a složitosti.

Klíčová slova šifrování, DPA, útoky postranními kanály, AES-128

Abstract

This work explores possible software countermeasures against differential power analysis for AES-128 algorithm implemented on an AVR micro-controller. The implementation of several countermeasures is compared with unprotected version in both security and efficiency.

Keywords encryption, DPA, side-channel attacks, AES-128

Contents

List of Figures

List of Tables

Introduction

Cipher design is a never-ending struggle. The perfect cipher is never complete and for every new one there is a proof-of-concept attack that exploits even the smallest design or implementation flaw. Advanced Encryption Standard (AES) described in 1.2 is considered to be mathematically strong and resistant to cryptanalysis, but its implementation in hardware might create many vulnerabilities. Side-channel attacks are one of them and they are introduced in 1.1 with emphasis on power consumption channel that's exploited in the demonstrated attack.

The countermeasures listed in 1.5 are crucial for any device that could expose secret information through the side channel. An ordinary AVR ATMega163 smart card was used for the implementation of the encryption algorithm and countermeasures. All considered countermeasures are mentioned in the chapter 2 with more detailed description and examples of the ones that were actually implemented on the smart card. The implementation details, especially the specifics of the ATMega163 environment and AVR 8-bit instruction set, can be found in the chapter 3.

The implementation is evaluated and compared with the original unprotected one using two power consumption models in the last chapter 4. The software countermeasures usually degrade the performance because they introduce extra operations. This issue is discussed in the chapter as well and possible solutions are proposed to alleviate the effects of applied countermeasures on both performance and memory consumption.

Chapter 1

Analysis

This chapter introduces the theory that was needed for AES-128 implementation, side-channel attack execution and countermeasure design.

1.1 Side-channel attacks

Side-channel attacks (SCA) exploit information obtained from the physical implementation of algorithm e.g. power consumption or electromagnetic radiation of circuit during execution. The exposure of the inner state or secret value stored on a piece of hardware is called leakage and we say that the device is leaking information.

Side-channel attacks require the attacker to know the implementation details of attacked algorithm and device specification. This requirement motivates the manufacturers to hide and obfuscate their design, but it was proved multiple times that this approach can never lead to truly secure product[1].

There are various side channels that can be exploited:

- **Power consumption**

  It exploits the fact that different operations and even processing of different values draw different amount of current. This side channel will be described in more detail in 1.1.1. Power consumption side channel was exploited by Kocher in his original paper on differential power analysis targeting DES implementation[2].

- **Electromagnetic field**

  The circuit emanates the EM signals while it is executing the algorithm. This side channel is very popular for attacks on contactless cards or in cases we don't have direct access to measure the power consumption.

- **Time**

  Duration of the execution is also considered to be a side channel. Attacks exploiting this channel are called timing attacks and they are very successful on algorithms with some degree of data dependency in execution.

In this work, we are only interested in power consumption side channel.

1.1.1 Power consumption channel

In our work, we use a micro-controller based on CMOS technology. CMOS power consumption depends on the logic circuit activity e.g. computations, storing and loading of values. The power consumption is influenced by the type of operation the circuit is doing and on processed values[3]. When switching between logical 0 and 1, a short leakage occurs before the value settles and this leakage is then exploited for differential power analysis.

Simple power analysis (SPA) is the easiest and least powerful of approaches. The attacker simply measures the power consumption during the algorithm execution, examines the traces and searches for patterns that would give away what values were processed. There are cases when this analysis can succeed because of the nature of the algorithm e.g. square and multiply algorithm in RSA encryption. The square and multiply algorithm leaks the information whether square or multiply operation is executed in every step that might even be visible to human eye.

Differential power analysis (DPA) is a more sophisticated method that makes use of statistics. The measured power consumption for multiple inputs is compared with chosen power consumption model for all possible values of the secret key and we choose the value whose modeled consumption matches the actual one best. In this work, the first order DPA with correlation coefficients (CPA) is used.

High order differential power analysis (HO-DPA) refers to advanced DPA that might use different data inputs and offsets in the measured traces. For example, if we know that the secret value is used twice in the algorithm, we can mount second order DPA, because there are two offsets in which the value will match our power consumption model for the value[3]. In case of AES-128 algorithm, the second order DPA could exploit the fact that the secret key is used for the key expansion and later for encryption.

1.1.1.1 Power consumption models

The measurement on its own is not enough to obtain any information in most cases and we need a power consumption model. It is a hypothetical model

for all possible values we create to make assumptions about the power consumption during the algorithm execution. If our chosen power consumption model fits the real consumption, it enables us to test all possible key values and determine which one was probably used, by correlation.

The model must contain the value that we are guessing, that's why we examine the attacked design and pick operations that process the secret value directly e.g. in case of AES-128, we might choose *AddRoundKey(0)* and *SubBytes* from the first round, because the very first *AddRoundKey* uses the original secret key while *SubBytes* from the first round depends only on the original key and plaintext.

When the attacked operation is chosen, we must select the model of consumption. We expect that when certain value is processed on the device, the power consumption depends on it and changes accordingly. The following models were used in the analysis.

Hamming weight sums number of 1s in the binary representation of the processed value. We collect traces for encryption of 100 plaintext inputs and we want to get the first byte of the secret key. The resulting model will be a 100×256 matrix with hypothetical consumption for all 100 inputs in case of 256 possible values of the byte. Each element of the matrix contains number of 1s in the result of *AddRoundKey(0)* and *SubBytes* operation.

Hamming distance uses number of bits in which two values differ - Hamming weight of the exclusive OR of the two values. For our AES-128 algorithm, we can compare the value after *AddRoundKey(0)* with value after *SubBytes*.

1.1.1.2 DPA with correlation coefficients

Differential power analysis becomes a powerful tool when combined with correlation coefficients. Most dependencies in power traces that could otherwise go unnoticed can be revealed by correlation coefficients.

For two sets of data e.g. measured power traces and our consumption model, correlation shows how much values from one set change when values from the other one change. Correlation does not imply causation and the two sets might actually be independent, it only claims that statistically the two sets appear to be dependent. The degree of correlation can be measured by correlation coefficients.

The correlation coefficient used for correlation of the two datasets in this

work is the most common *Pearson's correlation coefficient*. It is defined as:

$$\rho_{X,Y} = corr(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E\left[(X-\mu_X)(Y-\mu_Y)\right]}{\sigma_X \sigma_Y}$$

where X, Y are random variables with standard deviations σ_X, σ_Y and expected values μ_X, μ_Y. This correlation coefficient is particularly effective for revealing linear relationship between the two variables. The value of Pearson's correlation coefficient lies in the interval $[-1; +1]$, the border cases imply perfect direct linear relationship $(+1)$ or perfect inverse linear relationship (-1). If the value equals 0, there is no linear dependency between the two variables. For the sake of correlating the power consumption model and real power consumption measurements, we want to find the byte from the 256 possible values with the highest correlation coefficient output.

The measured traces are represented by matrix of m rows where each row represents one measurement for different plaintext input. Number of columns d is defined by the time interval and the precision of measurement depends on the configuration of our oscilloscope. The time interval should be the smallest possible or the computation will take considerably more time (there are 256 more correlations to compute for every extra column).

When we correlate columns in matrices:

$$traces = \begin{pmatrix} t_{0,0} & t_{0,1} & \cdots & t_{0,nr} \\ t_{1,0} & t_{1,1} & \cdots & t_{1,nr} \\ \vdots & \vdots & \ddots & \vdots \\ t_{p,0} & t_{p,1} & \cdots & t_{p,nr} \end{pmatrix}; \quad model = \begin{pmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,255} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,255} \\ \vdots & \vdots & \ddots & \vdots \\ h_{p,0} & h_{p,1} & \cdots & h_{p,255} \end{pmatrix},$$

we obtain the matrix of dimensions $256 \times nr$:

$$correlations = \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,nr} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,nr} \\ \vdots & \vdots & \ddots & \vdots \\ c_{255,0} & c_{255,1} & \cdots & c_{255,nr} \end{pmatrix}$$

The row containing the highest correlation coefficient is our most probable key byte guess and the column marks the time offset when the value leaked. Depending on the executed operations, the same value might leak multiple times, then we would see that multiple high coefficients appear in the same row.

1.2 AES

The encryption algorithm used in this implementation is AES-128. The original name of the underlying algorithm of AES is Rijndael and it was designed by Joan Daemen and Vincent Rijmen. In 2001, Rijndael became the new Advanced Encryption Standard to be used for encryption of electronic data. Rijndael and other 14 designs entered the contest for DES successor held by National Institute of Standards and Technology(NIST) and underwent extensive testing and cryptanalysis. Rijndael was voted the best one of the finalists because of its resistance to cryptanalysis and efficiency.

The side-channel attacks among other cryptanalysis approaches were also considered for all finalist designs and their vulnerabilities and possible countermeasures were discussed by Daemen and other experts in [4] and later in [5]. Daemen suggested complicating the exploitation of correlations through desynchronisation of the executed operations as software countermeasure and power consumption randomization by dedicated hardware module. The software countermeasures proposed by Daemen will be described later in section 1.5.

Description of winning AES Rijndael design as presented in [6] is given in this section and the important fundamental operations are mentioned. Fundamental operations used by encryption design determine possible vulnerabilities that can be exploited.

1.2.1 AES-128 overview

AES is a symmetric block cipher with block size of 128 bits and supported key lengths of 128, 192 and 256 bits. The original Rijndael cipher could use three different block sizes - 128, 192 and 256 bits, but this feature was not included in AES. The key length determines only the length of the expanded key and total number of rounds, while the rest of the algorithm stays the same for all lengths.

The input data is divided into 16 byte long blocks sometimes referred to as state or data path.

Every block is processed separately and according to chosen encryption mode, it might use initialization vector. For the sake of this work, we only describe Electronic Codebook (ECB) encryption mode with a single block which does not need any initialization vector and each data output is independent of the previous ones, because the smart card implementation only encrypts one block of data.

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

1.3 Mathematical structures and operations

AES design uses relatively small number of fundamental operations. If Rcon, Rijndael field multiplications and S-box are stored in memory and used as look-up tables, the algorithm does not contain any arithmetical operations.

1.3.1 Rcon

Operation $rcon(i)$ is defined as:

$$rcon(i) = x^{i-1} mod \quad x^8 + x^4 + x^3 + x + 1$$

The exponentiation is performed in Rijndael finite field $GF(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

The Rijndael finite field consists of 256 elements and the derived $rcon(i)$ values for $i \in [1; 256]$ can be stored in memory as a look-up table to speed-up the execution. *rcon* is essential to computation of the S-box. The multiples of Rijndael field elements are used in *MixColumns* part of the algorithm.

1.3.2 S-box

S-box is a non-linear transformation defined as:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}^{-1} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Following example demonstrates computation of an S-box value without direct usage of matrix multiplication in more human readable format:

1.3.2.0.1 S-box computation example We will get the S-box output for `0x53`.

1. Find the multiplicative inverse of `0x53` in Rijndael finite field using Extended Euclidean Algorithm.

$$(\texttt{0x53})^{-1} = \texttt{0xCA}$$

2. The matrix multiplication can be described by the following pseudocode:

```
s = 1100 1010  // s <- 0xCA
x = 1100 1010  // x <- 0xCA

for i in 0..3
 s = s << 1  // shift 1 bit left
    x = x xor s
end
return x // 0x8E
```

3. Xor with `0x63`:

$$\texttt{0x8E} \oplus \texttt{0x63} = \texttt{0xED}$$

Now we have the output:

$$sbox(\texttt{0x53}) = \texttt{0xED}$$

The non-linearity of S-box is a good trait for the security of the algorithm, but it introduces extra steps and memory usage for masking countermeasures. Any countermeasure that modifies the processed value must convert the S-box look-up table or algorithm.

1.4 Design

AES-128 uses 128 bit long key and takes 10 rounds to process one block of input data. Each round consists of the same sequence of basic operations - *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. The last round skips *MixColumns* step. *AddRoundKey* uses different round key in every round. The round keys are derived from the original 128-bit key during *KeySchedule* - either all of them before we start to encrypt or one at a time before its respective round.

1.4.0.1 KeySchedule

The secret key used in both encryption and decryption is expanded either one at a time before its respective round or all at once. Every 32bit word of an expanded key depends on words from the previous round key. The complete key schedule is illustrated in 1.1.

There are 10 rounds in AES-128, but round keys contain one extra key, the original one, that's mixed with plaintext before the first round. It's named *RoundKey 0*.

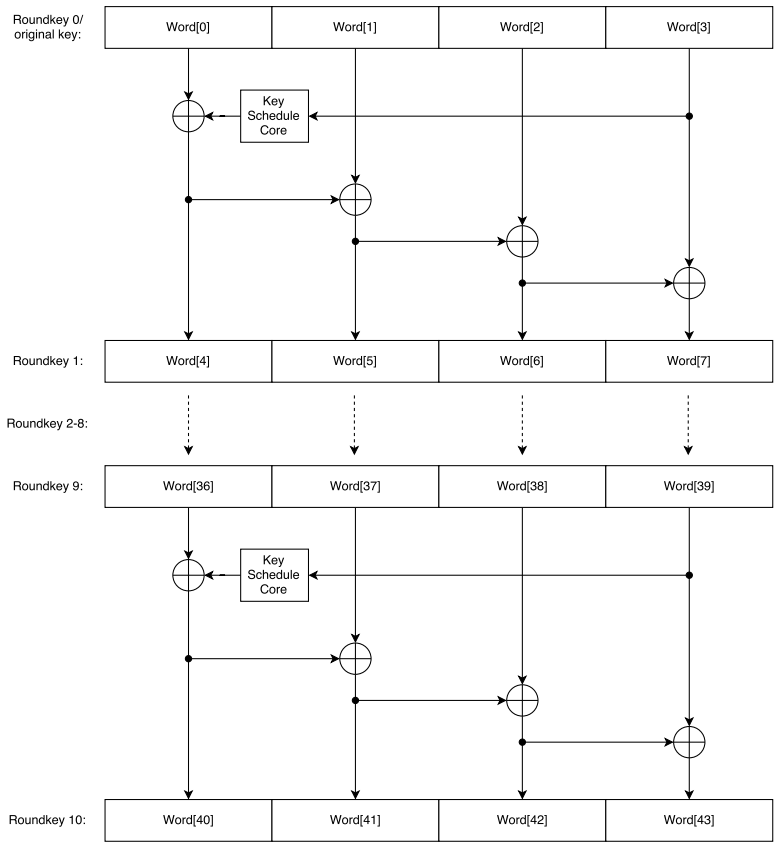


Figure 1.1: AES-128 key schedule

The last word of each round key is modified by *key schedule core* before it's exclusive ORed with the first word and used as the first word of the new round key.

1.4.0.1.1 Key schedule core The picture 1.2 describes the key schedule core in detail. The input 32bit word is rotated by 8 bits left, S-box transformation is applied on each byte and first byte is exclusive ORed with *rcon*.

The argument of *rcon* is derived from the round number. For the first round $i = 2$, for next rounds i is always incremented by 1.

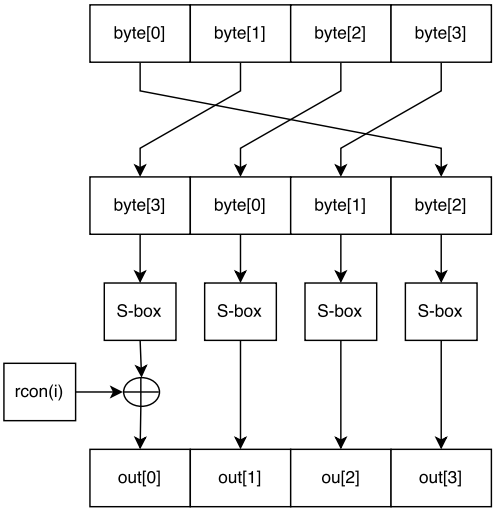


Figure 1.2: Key schedule core

1.4.0.1.2 Used operations and possible vulnerabilities

The key schedule routine uses logical operations exclusive OR of two 32bit words and logical left shift of 32 bit word by one byte. If both S-box and *rcon* are implemented as an algorithm instead of in-memory look-up table, it executes several arithmetical operations in Rijndael finite field - exponentiation with exponents from 1 to 10 in *rcon* and computation of multiplicative inverse, logical left shift of 8 bits by one bit and addition in S-Box.

The original key value might be exposed during the expansion or it could help in execution of high order DPA, but the secret key never changes because it serves as its identifier, therefore the full expanded key can be stored in memory and does not need to be computed every time.

1.4.0.2 AddRoundKey

AddRoundKey(i) mixes in the round key for the i-th round using exclusive OR logical operation. The first execution of *AddRoundKey(0)* before the first round is the most vulnerable one. If no countermeasures are applied the whole original 128-bit key is exposed through the power consumption of the logical operation or even by loading its bytes into cleared registers.

1.4.0.3 SubBytes

Depending on the implementation, *SubBytes* executes the whole S-box element computation or uses the look-up S-box table stored in the memory. The first round *SubBytes* is the easiest one to attack because the processed value contains only plaintext and the key we are trying to guess.

The following two operations *ShiftRows* and *MixColumns* are both part of the diffusion layer. The diffusion layer does not add any additional secret information.

1.4.0.4 ShiftRows

The rows in the *ShiftRows* refer to the rows of the table representation. The first row is not shifted, the second one is shifted one position left, the third one is shifted two positions left and the last one three positions left.

B_0	B_4	B_8	B_{12}
B_5	B_9	B_{13}	B_1
B_{10}	B_{14}	B_2	B_6
B_{15}	B_3	B_7	B_{11}

1.4.0.5 MixColumns

Unlike *ShiftRows* the *MixColumns* operation applies the same transformation on all columns. The first column of the new block is computed as:

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} * \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}$$

The same rule applies on the other three columns. The multiplication is performed in Rijndael finite field and again we can choose between the algorithm and multiplication look-up tables.

The *MixColumns* is skipped in the last round of AES.

1.5 Countermeasures

The side-channel attacks are not very frequent compared to other cybersecurity threats such as malware infections of user computers, but because of

the nature of affected devices they pose greater threat if actually executed. The manufacturers of hardware keys, encryption modules, access cards or pay cards cannot afford to ignore this threat.

The countermeasures can be hardware or software. Hardware countermeasures tend to be more powerful, but the device must be already designed with them and they usually introduce extra cost. Software countermeasures are employed to heighten the security on devices that lack the hardware ones or they can complement the hardware ones. Most described countermeasures are mentioned in [3].

1.5.1 Hiding

Hiding countermeasures attempt to break the link between the processed value or executed operation and the power consumption. There are multiple ways how to do that and most of those countermeasures are implemented in hardware. Hiding countermeasures do not change the processed values and they are insensitive to the algorithm that's executed on the device.

There are two approaches on how to hide the dependency in power consumption – by randomizing the consumption or making it constant for all operations and all values. Both approaches will cause that the attacker cannot obtain any exploitable information from his measurement of the consumption. The perfectly random or constant consumption cannot be achieved, but the attempts to create the best hiding design can be divided in two categories according to the dimension they use - time or amplitude.

1.5.1.1 Time

Differential power analysis exploits the fact that the encryption algorithm operations are executed in the same order and therefore the wanted values appear at the same offset in the measured traces. That can be avoided by inserting dummy operations at random points of the execution or by shuffling the order of the algorithm's operations.

1.5.1.1.1 Shuffling If there are operations whose execution can be interchanged without changing the final output, they can be shuffled. AES-128 always processes one block of data which means there are always 16 bytes to be exclusive ORed with 16 bytes of the expanded key in *AddRoundKey* and 16 bytes to be substituted using S-box in *SubBytes*. The order of those two operations cannot be changed, but the order in which the 16 bytes are processed can be randomized each time.

1.5.1.1.2 Dummy operations We can break the alignment of the power traces with randomly inserted dummy operations. The total number of dummy operations must stay the same for each program execution to prevent the timing attacks.

1.5.1.2 Amplitude

The power consumption can only be exploited if the attacker can measure it. Hiding the consumption in amplitude means that we manipulate the signal-to-noise ratio by either increasing the noise or reducing the signal, therefore the attacker cannot measure the consumption correctly or there is no exploitable information in it.

1.5.1.2.1 Noise The noise can be added by switching on an unrelated noise generator built in the device. This generator can do some useful operations or it can create random signal. The major disadvantage of dedicated noise generator is the extra power consumption that might be undesirable.

Another way to increase the noise is by executing several operations in parallel. For this purpose either multiple processor cores can be used or independent operations of the algorithm can be executed. In case of AES-128, if we had 16-bit wide ALU, we could compute two bytes from the block at once and the attacker would be forced to model the consumption for 2^{16} values of the two bytes instead of usual 2^8 for one byte.

1.5.1.2.2 Signal One of the possible countermeasures proposed by Daemen in [4] was designing the device with special invertor that would write or read the inverse of the value that's actually written or read at the same time. This would cause the power consumption of the read and write operation always appear constant for all values.

Designing the device in the way that would draw the same amount for every operation is not trivial. The attempts are made either to build the cells that would have constant power consumption or to filter the actual consumption.

1.5.2 Masking

While hiding attempts to hide the link between the processed value and the consumption, masking modifies the processed value itself. Masking can be implemented in software much easier than hiding and the masking scheme implementation can be simply changed later if new type of side-channel attack occurs without the need to change the hardware.

The main goal of masking schemes is to combine the intermediate value with mask to change its characteristic power trace. The value is still leaking, but it's not the real value and if we choose the mask carefully, the attacker cannot analyze the measured traces easily.

$$v_m = v \circ m$$

The mask m is generated on the device and it should be different for each execution of the algorithm. The operation $\circ$ can be an arbitrary operation, but we have to know its inversion to obtain the original intermediate value in the end or we have to compute a compensation mask along with the algorithm execution. Depending on the type of the operation, we have to adjust the executed algorithm. For example if the algorithm contains any non-linear operations, the usage of the mask must be taken into account so that the final output does not change.

1.5.2.1 Arithmetic vs. Boolean operation

The type of operation used for the masking is usually determined by the executed algorithm. When applying a mask we must make sure the operations that process the transformed value are linear:

$$f(x) + f(y) = f(x + y)$$

$$(v_1 \circ m) + (v_2 \circ m) = (v_1 + v_2) \circ m$$

Otherwise the mask would interfere with the calculation and we would not be able to obtain the desired output after unmasking. If we use more masks and the algorithm mixes them at some point, we either have to make sure the mixing cannot remove the masks or we have to choose the masks in a way that will make such interfering impossible.

Algorithms using arithmetic operations - modular addition and multiplication, mix in the mask with arithmetic operation and algorithms using Boolean operations - exclusive OR, use the exclusive OR for masking. There are algorithms that combine both types of operations and they require switching between the masking types which introduces extra resources and is not very efficient.

If we use more masks, we might need to compute a compensation mask that will be applied on the masked output to obtain the actual output.

1.5.2.1.1 Arithmetic The most common arithmetic operations used in encryption algorithms are modular addition and multiplication:

$$v_m = v + m \ (mod\ n); \quad v_m = v \cdot m \ (mod\ n)$$

The biggest disadvantage of arithmetic masking with multiplicative masks is the inability to mask the zero element because of the nature of the operation:

$$\forall m: \quad m \cdot 0 = 0 \cdot m = 0 \quad \leftarrow \text{ zero element for the } \cdot \text{ operation}$$

This property would leave all zero values unmasked and could be exploited by the attacker.

1.5.2.1.2 Boolean The most popular Boolean operation in encryption is exclusive OR and it is also used for masking:

$$v_m = v \oplus m$$

We will refer to the masking scheme used in our AES implementation as Boolean, but because the Boolean operation $\oplus$ is the same as arithmetic + in the Rijndael finite field, it is actually both arithmetic and Boolean.

1.5.2.2 Masking methods

1.5.2.2.1 Blinding Blinding refers to the masking technique used in asymmetric cryptography e.g. RSA. The arithmetic mask can be applied by additive or multiplicative operation. In case of RSA, it is used either as message blinding or exponent blinding. Message blinding masks the message with m^e by multiplication where m is the random mask and e is the public key. Exponent blinding takes advantage of adding the mask $m \cdot \phi(n)$ that does not change the output because $v^{d+m\cdot\phi(n)} \equiv v^d (mod\ n)$.

Using the masking like this is possible because of the arithmetic nature of the algorithm.

1.5.2.2.2 Secret sharing The intermediate value v is divided into number of shares and only when the attacker knows all the shares he can get the secret value itself. Secret sharing with two shares (m, v_m) is achieved by applying a mask on the intermediate value $v_m = v \oplus m$. Dividing the intermediate value into more shares increases the security.

Dividing the secret into more shares can even protect it against HO-DPA – n masks can prevent up to n-th order DPA according to [7].

1.5.2.3 Mask generation and usage

1.5.2.3.1 Random The mask can be generated randomly and then we have to ensure that the generator we used to obtain the random numbers is not predictable. If the device has any peripheries or for example temperature or voltage on components can be measured, we can use the measurements as a source of entropy and use true random number generator. However mask generation should not take too long, otherwise, especially in masking schemes that use multiple random masks, the efficiency of the encryption will drop significantly.

Random masks in AES implementation mean an extra S-box look-up table conversion for each used mask, because non-linear S-box operation must fit the rule:

$$\text{S-box}(v \oplus m) = \text{S-box}(v) \oplus m$$

S-box conversion itself is very straightforward, but all 256 values have to be read, modified and written each time it is converted – with the used smart card AVR ATMega163's 8-bit instruction set the S-box conversion means 2 clock cycles for each read and write operation or 3 clock cycles if the look-up table is stored in the program memory. The exclusive OR operation costs another extra clock cycle and the loop counter decrement adds another cycle. Converting one S-box look-up table takes at least $256 * (2 + 1 + 1) = 1024$ extra clock cycles. The cost of the S-box conversions for random masks led to introduction of fixed masks and low entropy masking schemes (LEMS) with precomputed converted S-boxes stored in memory.

1.5.2.3.2 Fixed The encryption algorithms must balance the security and performance and randomly generated masks degrade the performance with repeated conversions of non-linear operations. Precomputing all possible S-boxes in AES would be possible but for one byte mask, we would need 256 masks $\times$ 256 S-box elements $=$ 64KB of memory. For smart cards, 64KB in extra look-up tables is not the best option because of memory constraints.

The memory and time constraints caused the researchers to look for other viable options and design lightweight countermeasures against selection of the most important and powerful attacks. In low entropy masking scheme, we don't use the full randomness and instead we select a subset of masks with good properties, e.g. they don't cancel out each other during execution. The small set of possible masks can then be rotated randomly before being applied. This method called Rotating Sbox Masking (RSM) was implemented and tested in DPA contest v4 for AES-256 implementation on ATMega163 smart card[8].

Chapter 2

Design

When designing any countermeasures to be used on such a small device as smart card, the memory and other constraints must be taken into account, therefore this chapter introduces the platform of implementation - Atmel ATMega163 smart card with customized Simple Operating System for Smartcard Education (SOSSE).

This chapter also describes the countermeasures designed for protection.

2.1 ATMega163 smart card

The smart card used for DPA and countermeasures design is AVR ATMega163 with 24C256 EEPROM. ATMega163 is not a state of the art technology, but it is very popular for practical DPA demonstration and software countermeasure design, because it lacks any hardware protection from DPA, does not produce much additional noise unrelated to executed instructions and it's also used in DPA contest v4 [9].

ATMega163 described in [10] is an AVR RISC micro-controller manufactured by Atmel Corporation. The micro-controller is equipped with ALU that can process 8-bit values, Data bus that's able to transport 8-bit values, instruction set with 130 instructions and RISC architecture described in Figure 2.1.

2.1.1 Data memory

On Harvard architecture, the Program memory and Data address Space is not shared.

General purpose registers

ATMega163 uses 32× 8-bit general purpose registers named `r0-r31`. Those registers constitute the Register File which is the range `$0000-$001F` in Data Address Space and they can be accessed in only one clock cycle.

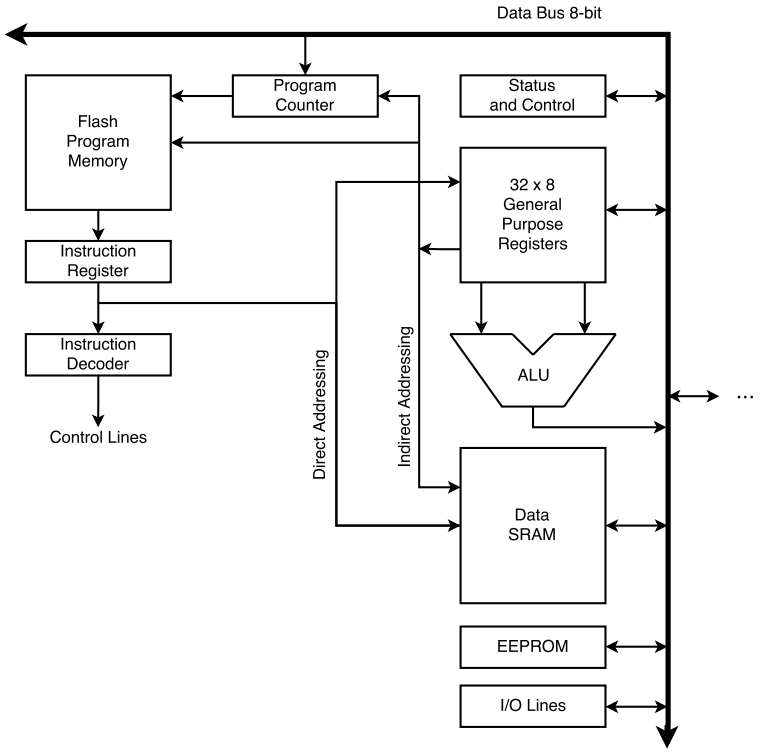


Figure 2.1: ATMega163 RISC architecture

The six highest registers `r26-r31` can be used as three 16-bit indirect address register pointers and one of these extended registers can be used as address pointer for look-up tables in Flash Program memory.

I/O registers

The I/O registers `$00-$3F` can be accessed from `$0020-$005F` in Data Address Space. I/O Space consists of Status register SREG at `$3F($5F)`, Stack Pointer High and Low parts at `$3E($5E)` and `$3D($5D)` and other I/O registers.

The Stack Pointer points to the first available storing location and it must be set to point above `$60`. It should be set by the programmer at the beginning of his main routine and it usually points to `RAMEND` that marks the last available address of Data Space. Stack is descending, as Stack Pointer is decremented by 1 or 2 when 8-bit or 16-bit value is pushed onto it and incremented by 1 or 2 when it's popped.

It's important to limit the depth of nested subroutines calls on AVR microcontrollers and match all `PUSH` and `POP` instructions, because stack underflow will rewrite the I/O registers under `$60`, keep Stack Pointer High and Low parts and Status register in tact(according to AVRStudio Simulator) and continue to rewrite all registers until it exhausts the Data Address Space. If we

use too much internal SRAM memory in our code for example for storing look-up tables and there is not enough space left for stack to grow, they will be rewritten by stack underflow and the program will not return correct output.

Internal SRAM
Internal SRAM is the main data memory and it starts at `$0060` in Data Address Space. Read and write operations are fast in this memory, but it's volatile and its size is limited. Internal SRAM contains the variables of the executed program and it shares the space with stack as mentioned before. We cannot use the internal SRAM as storage for multiple look-up tables and other large data structures - either the program will not run at all if we use more memory than we have or if we fit in but don't leave enough space for the stack, its behavior will not be defined as the resulting stack underflow will rewrite our variables.

2.1.2 Flash Program memory

The programmable flash memory is where the program instructions are stored. It's considerably bigger and it's non-volatile. If we need to store any large data structures such as S-box look-up tables, we have to store them in Flash Program memory. However, storing the data in program memory will cause that all read and write operations on such data will take one extra clock cycle.

The Program memory is executed with a two stage pipeline - one instruction is executed while the next one is pre-fetched. This enables the execution of instructions in every clock cycle. The Program memory is divided in two sections - the Boot program section and the Application Program section. Those two sections have separate Boot Lock bits and user can select the level of protection for these sections. The combination of the bits will determine the size of the Boot program section and it can be set up to 1024 bytes.

The main difference between the two sections is the ability to execute Store Program Memory(`SPM`) instruction as it can rewrite any address in the Program memory but can only be executed from the Boot program section. The Boot program section can use the `SPM` instruction to update both the Boot and the Application sections but they can only be modified page by page (128 bytes). This leads us to the realization that the Program memory can be used for storing additional data that do not fit into SRAM, but those data must be read-only and must be saved in memory before the program execution with the rest of the Flash Program memory.

In practice, if we want to place any converted masked S-box look-up tables into the Program memory, we have to precompute them, mark them with the `PROGMEM` macro in `C` source code and program the AVR Flash memory with them. In case we generate only one mask randomly, the look-up table will fit into the SRAM and can be recomputed on every execution of the algorithm, but if we wanted to use two or more masks scheme, we would quickly use up

the space in SRAM (each S-box takes 256 bytes from the total 1024 bytes of SRAM).

For randomly generated multiple masks, we would have to precompute and save all 256 possible S-box look-up tables which would cost us 64KB in total and would exceed even the Flash memory limit of ATMega163. On the other hand, if we only use a fixed subset of all possible masks, we can fit their look-up tables in the Flash memory. DPA contest v4 [9] chose this path and used only subset of 16 masks with desired characteristics.

2.1.3 EEPROM

Additional Electronically Erasable Read-Only Memory [11] is usually used for device identification data that do not change frequently. Storing or loading any data from EEPROM during the algorithm execution would take too much time - EEPROM is only accessed via data bus unlike SRAM or Flash memory. Despite being read-only, it can also be rewritten by the program instructions, but its latency is too high and the size not sufficient for storing big amounts of data.

2.1.4 Instruction set

Instruction set described completely in [12] contains 130 instructions, most of them can be executed in a single clock cycle. One instruction is usually executed and the next one pre-fetched.

ATMega163 is designed as RISC load/store architecture where the instructions are strictly divided in two categories - memory access (load value from Data space into register or store register content into Data space) and ALU operations. ALU operations are performed only on registers, never on register and immediate value.

Total Execution Time of one ALU operation consists of Register Operands Fetch, ALU Operation Execute and Result Write Back phases described in [10] and illustrated by Figure 2.2:

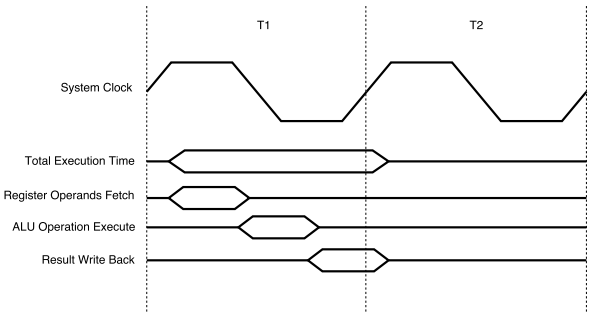


Figure 2.2: Single Cycle ALU Operation

If we measure the power consumption of such operation, we can see not only the consumption of the ALU Operation Execute phase but also change in register value (number of flipped 1s) during Result Write Back. This fact leads us to realization that writing the software countermeasures in `C` language is not sufficient and we must implement the critical parts in AVR assembly language.

2.1.4.1 Data transfer instructions

Load and store instruction in Data space take 2 clock cycles and load from Program memory with `LPM` instruction takes 3 clock cycles. Loading any data from Program memory will cost an extra cycle and storing is not used in practice. In our implementation, the Program memory is used for the original static S-box look-up table and Rijndael field multiplication look-up tables.

Data space load and store instructions can use either direct or indirect addressing. Direct addressing loads or stores one byte from or to given address in Data space, indirect addressing loads or stores one byte from or to address in Data space pointed by X-,Y- or Z-pointer register.

2.1.4.1.1 Load Loading value from SRAM or Flash memory will expose the Hamming weight of the value if the destination register Rd was cleared before the load operation. In order to prevent this leakage, we can precharge the register with random value before we load the secret value.

2.1.4.1.2 Store Storing the value from register to Data space can either leak the Hamming weight of the secret value in case the memory was cleared before storing (this would happen if we initialized the variable with `0x00` and copied the secret value into it) or it can leak the Hamming distance of the last stored value and the new one for example the difference between the plaintext value before adding the round key and after.

2.1.4.2 Logical instructions

AES encryption algorithm with look-up tables does not use many logical operations. The most frequently used instruction on ATMega163 platform is `EOR`. Instruction `EOR` executes an exclusive OR on registers Rd, Rr and stores the result into Rd[12]:

$$\texttt{Rd} \leftarrow \texttt{Rd} \oplus \texttt{Rr}$$

There are multiple ways to exploit the `EOR` instruction. The easiest one is using the Hamming distance of the two operands that leaks during the instruction execution:

$$HD(\texttt{Rd}, \texttt{Rr}) = HW(\texttt{Rd} \oplus \texttt{Rr})$$

This leakage can be prevented if only one of the operands is known to the attacker and he cannot use plaintext and key combination in his power consumption model. If either the plaintext value or key value are masked with random masks, the Hamming weight of their combination is not visible in the traces and the power consumption of their exclusive OR is also masked.

Another way to exploit the `EOR` instruction is by targeting the change of the destination register. The power consumption depends on number of 1s that need to be flipped during the write back phase. The difference between old and new value of register `Rd` is `Rr`, which can be exploited for DPA if we used the operation with the secret key loaded in `Rr`:

$$\texttt{Rd} \leftarrow \texttt{1001 0101} \quad \texttt{plaintext 0x95}$$

$$\texttt{Rr} \leftarrow \texttt{1010 1010} \quad \texttt{key 0xAA}$$

$$\texttt{Rd} \leftarrow \texttt{1001 0101} \oplus \texttt{1010 1010}$$

$$HD(\texttt{Rd}, \texttt{Rd} \oplus \texttt{Rr}) = HW(\texttt{Rr})$$

The Hamming weight of the secret key can be leaked with wrong operands order. Only when we write the code in assembly, we can be absolutely sure that operands will be used in the right order.

2.2 Used countermeasures

2.2.1 Boolean masking

Masking scheme using Boolean operation can be implemented by exclusive OR-ing a byte of mask to byte of key or plaintext. For linear operations in AES, this is very straightforward, but for non-linear S-box substitution done by `SubBytes()` converted masked S-box look-up tables must be computed.

2.2.1.1 Single mask

The first attempt to mask the power consumption of `AddRoundKey(0)` and `SubBytes()` was made with one pseudorandomly generated mask. This approach was successful in masking the Hamming weight of the output of `AddRoundKey(0)`→`SubBytes()` result.

After measuring the power consumption and correlating the traces with the consumption model based on Hamming weight, none of the 16 bytes was retrieved successfully and it did not even occur in top 3 guesses. Single mask scheme worked on Hamming weight model, but was not able to mask the Hamming distance of the value before `SubBytes()` and after because `SubBytes()` performs substitution:

$$\texttt{p} \oplus \texttt{k} \oplus \texttt{m} \rightarrow \texttt{S-box(p} \oplus \texttt{k)} \oplus \texttt{m}$$

where p one byte of plaintext, k one byte of secret key and m is random mask. The difference between the input and output of unmasked S-box is defined as:

$$(\mathtt{p} \oplus \mathtt{k}) \oplus (\mathtt{S\text{-}box}(\mathtt{p} \oplus \mathtt{k}))$$

If we use the same mask for input and output, the difference will be:

$$(\mathtt{p} \oplus \mathtt{k} \oplus \mathtt{m}) \oplus (\mathtt{S\text{-}box}(\mathtt{p} \oplus \mathtt{k}) \oplus \mathtt{m})$$

and the single mask will not mask the actual difference done by substitution.

2.2.2 Using different registers

This countermeasure was originally designed to mask the Hamming distance of S-box substitution when it only used single mask masking scheme. The original value was stored into register `r19` and the output of the look-up operation for this value was stored into register `r24`.

This countermeasure prevented the leaking of difference between the old and new value in register, but was later abandoned because it did not solve the issue with storing the substituted value back into SRAM. Instead, the flaws of single mask masking scheme were solved with upgrade to better masking scheme using input and output masks.

2.2.2.1 Input and output masks

Better way to convert the S-box look-up table that does not leak Hamming distance of input and output, is generating two distinct masks `m_in` and `m_out`. S-box substitution with two different masks is defined as:

$$\mathtt{p} \oplus \mathtt{k} \oplus \mathtt{m\_in} \rightarrow \mathtt{S\text{-}box}(\mathtt{p} \oplus \mathtt{k}) \oplus \mathtt{m\_out}$$

Now the difference between the old value and substituted value changed to:

$$(\mathtt{p} \oplus \mathtt{k} \oplus \mathtt{m\_in}) \oplus (\mathtt{S\text{-}box}(\mathtt{p} \oplus \mathtt{k}) \oplus \mathtt{m\_out})$$

and the difference of S-box substitution is mixed with $\mathtt{m\_in} \oplus \mathtt{m\_out}$. The input and output masks must be always distinct, otherwise they would remove each other from the equation, and they should have different Hamming distance each time the algorithm is executed.

2.2.3 Random register precharging

The registers we work with should be handled as if they contain the worst possible value – `0x00`. Previous routines might have cleared the registers and loading any secret value into cleared register equals leaking the Hamming distance of the value from `0x00` which is Hamming weight of the value itself.

The Hamming weight itself does not always give away the contained value but it can help the attacker to narrow down the list of possible candidates.

To prevent the exposition of loaded value, we precharge the destination register with random value that will be later overwritten. The leaked Hamming distance is different for each algorithm execution.

Similar countermeasure could be applied to SRAM store instruction, but it would slow down the algorithm even more and the change in SRAM stored values did not prove to be as visible as change in register stored values.

2.3 Other countermeasures

There are also masking countermeasures for operations `ShiftRows` and `MixColumns`. These operations are more difficult to analyze because they don't work with one byte but with combination of 2 to 4 bytes.

The countermeasures described in this section were designed and considered but were not implemented either due to their expected inefficiency or because they targeted parts of algorithm that were not analyzed with used power consumption models.

2.3.1 ShiftRows

This operation's leakage depends on implementation of the shifts. The `ShiftRows` operation rotates one byte of data from its position in row to left. For the first row, there is no shift, for the second row the values are rotated one byte left:

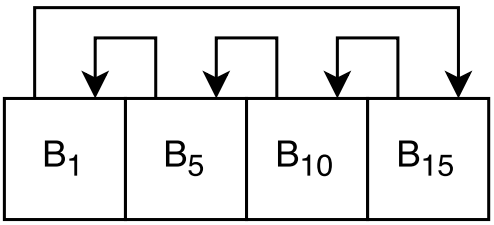


Figure 2.3: Rotate second row

The value stored originally on B_1 position is defined as:

$$B_1 = \texttt{S-box}(\texttt{k}_1 \oplus \texttt{p}_1)$$

With applied mask it's:

$$B_1 = \texttt{S-box}(\texttt{k}_1 \oplus \texttt{p}_1) \oplus \texttt{m}$$

The neighbour shifted on B_1 position is defined with applied mask as:

$$B_5 = \texttt{S-box}(\texttt{k}_5 \oplus \texttt{p}_5) \oplus \texttt{m}$$

When the value on position B_1 changes to B_5, the leaked Hamming distance will be the Hamming weight of their difference:

$$HD(B_1, B_5) = HW(\texttt{S-box}(\texttt{k}_1 \oplus \texttt{p}_1) \oplus \texttt{m} \oplus \texttt{S-box}(\texttt{k}_5 \oplus \texttt{p}_5) \oplus \texttt{m})$$

We can clearly see that the masks on B_1 will interfere with the mask on B_5. If we wanted to model the power consumption, we would need to create the hypothetical consumption for the combination of B_1 and B_5, that's $2^{16} = 65536$ possible combinations, but we would be able to guess two keys at once. Compared to $2 * 2^8$ cost of acquiring two bytes with DPA on operations that only work with one byte, it's more difficult.

For the third and fourth row, we should implement the rotation as direct displacement of two values and avoid the two or three consecutive shifts. This will lower the number of leaking values and also speed up the execution.

If we wanted to mask the `ShiftRows` operation, we would need four distinct random masks for the four values of each row $\texttt{m}_0, \texttt{m}_1, \texttt{m}_2, \texttt{m}_3$. Because no mixing between rows happens in this operations, we can use the same set of 4 masks on all three rows. The first row does not need to be masked, because it's not shifted.

With new masks, the difference between B_1 and B_5 would be:

$$HD(B_1, B_5) = HW(\texttt{S-box}(\texttt{k}_1 \oplus \texttt{p}_1) \oplus \texttt{m}_0 \oplus \texttt{S-box}(\texttt{k}_5 \oplus \texttt{p}_5) \oplus \texttt{m}_1)$$

and the masks $\texttt{m}_0$ and $\texttt{m}_1$ would stay in tact. Of course, the Hamming distance of every pair of masks must change with each execution to keep the consumption random.

This countermeasure was not used in practice because our attack targets the operations that use only one byte of the key.

2.3.2 MixColumns

The last operation left to analyze and protect is `MixColumns`. This operation mixes the values from each column. After `ShiftRows`, the byte C_1 is defined as:

$$C_1 = \texttt{S-box}(\texttt{k}_5 \oplus \texttt{p}_5) \oplus \texttt{m}$$

and with `MixColumns` it changes to:

$$C_1 = 01 * B_0 + 02 * B_5 + 03 * B_{10} + 01 * B_{15} + \texttt{m}$$

To shorten the definition, we define $\texttt{S-box}_i = \texttt{S-box}(\texttt{k}_i \oplus \texttt{p}_i)$, where $\texttt{i}$ is the position of byte in the block.

$$C_1 = 01 * (\texttt{S-box}_0 \oplus \texttt{m}) + 02 * (\texttt{S-box}_5 \oplus \texttt{m}) + 03 * (\texttt{S-box}_{10} \oplus \texttt{m}) + 01 * (\texttt{S-box}_{15} \oplus \texttt{m})$$

First, we will show that the operation `MixColumns` is linear and the mask can be easily removed from the output:

$$C_1 = \texttt{S-box}_0 + 02 * \texttt{S-box}_5 + 03 * \texttt{S-box}_{10} + \texttt{S-box}_{15} + \texttt{m} + (02 + 03) * \texttt{m} + \texttt{m}$$

$$C_1 = \texttt{S-box}_0 + 02 * \texttt{S-box}_5 + 03 * \texttt{S-box}_{10} + \texttt{S-box}_{15} + 01 * \texttt{m}$$

The difference between the original C_1 and the new one is defined as:

$$HD(C_1, C_1^{'}) = HW(\texttt{S-box}_5 \oplus \texttt{m} \oplus \texttt{S-box}_0 \oplus 02 * \texttt{S-box}_5 \oplus 03 * \texttt{S-box}_{10} \oplus \texttt{S-box}_{15} \oplus \texttt{m})$$

For the creation of power consumption model based on Hamming distance of value on specific position, we would need a combination of 4 bytes, which is $2^{32} = 4294967296$ possible values for 4 bytes of the key compared to $4 * 2^8$ with operations using only one byte.

The operation `MixColumns` is divided into multiple multiplication and addition operations that can be attacked separately to avoid the combination of multiple bytes.

To mask this operation, we would use similar masking scheme as `ShiftRows`, but it would be applied on columns. Each column would have 4 distinct masks with random Hamming distance of all pairs for each execution.

2.3.3 Dummy cycles

One of briefly considered but abandoned countermeasure was insertion of dummy cycles. This is a hiding countermeasure described before and it has few downsides. Firstly, it adds empty operations that do not contribute to the algorithm execution and secondly, it can be filtered from the traces with some pre-processing. With pre-processing, we could fix the misaligned traces because there would be parts that occur in all traces randomly interleaved with traces of dummy cycles.

Number of inserted dummy cycles and their total duration must be the same for every execution, otherwise there would be an opportunity for timing attack because of different lengths. If the operation in dummy cycles is still the same, it can be easily detected in the traces because of the repeating pattern.

Inserting dummy cycles as software countermeasure was not used in the end because it would degrade the performance significantly and can be invalidated by pre-processing the traces.

Chapter 3

Implementation

3.1 Programming languages

The implementation uses two languages – the main language of the project is C and few important functions are written in AVR assembly language. C code is more readable when we need to get a grasp of the program functionality, assembly language enables us to define the executed instructions exactly as we want them.

3.1.1 C and assembly mixing

With `avr-gcc`, we can either use the inline assembler functionality in C or combine C and assembly source codes. Combining C and assembly source codes enables us to write the whole functions in assembly instead of chunks of codes inside existing C functions. The combination of C and assembly code in AVRGCC project is described in Atmel Application Note [13]. Our implementation only uses calling of assembly routines, passing variables to assembly and sharing global variables.

The C and assembly code is separated into two different files with extensions `.c` for C source code and `.S` for assembly code.

3.1.1.1 Using registers

The convention of `avr-gcc` dictates certain usage of registers that must be complied with, otherwise the behavior of code generated by `avr-gcc` from C can be unpredictable and incorrect. The complete summary of registers usage is depicted in table 3.1.

If we want to use some of the "call-saved" registers in our assembly routine, we must `push` them on stack before working with them and then `pop` the value back before we return from the routine. `avr-gcc` will expect that all "call-saved" registers and `r0` will stay in tact and `r1` will contain `0x00`. All the

other registers are available freely to the assembly code, but their content is not defined.

Register	Description	Usage in assembly
r0	Temporary	save and restore
r1	Zero	clear before returning
r2-r17	"call-saved"	save and restore
r28		
r29		
r18-r27	"call-used"	use freely
r30		
r31		

Table 3.1: Register usage in assembly with `avr-gcc`

As we can see from table 3.1, assembly code can use two of three extended registers freely – X-pointer and Z-pointer.

3.1.1.2 Sharing global variables

Only global variables from C can be made visible to assembly as the rest of them is defined in local context of their functions. The global variable must not be declared as `static` because it makes it invisible to other object files.

```
uint8_t mask; // in .c          .extern mask  ; in .S
```

The value of global variable `mask` can then be loaded into register `r25` accessing the passed address indirectly via extended Y-pointer:

```
.extern mask ; at 16-bit address 0x0061
lds r31, mask+1
lds r30, mask
ld r25, Z
```

3.1.1.3 Calling assembly routines from C

The routine called from C code has to be declared in C and assembly as follows:

```
    // in .c                    ; in .S
extern void subBytes(void);     .global subBytes
                                subBytes:
                                    ; routine
                                    ret
```

In this example, the function does not take any arguments, therefore we don't need to acquire them in assembly. If it accepted any arguments, they would be passed according to `avr-gcc` convention.

Arguments in fixed argument list are passed in registers `r8-r25`. This range differs in Atmel manual [13] and Atmel Libc reference manual [14], however it was tested with AVRStudio4 and the full range `r8-r25` is used before storing the arguments to stack.

Each argument consumes an even number of registers. The unused byte in case of one byte values is not touched. If argument does not fit into available registers, it is passed on the stack. Arguments of function with variable argument list are also passed on stack but in the right to left order. The order of assignment is from left to right as depicted in Figure 3.1.

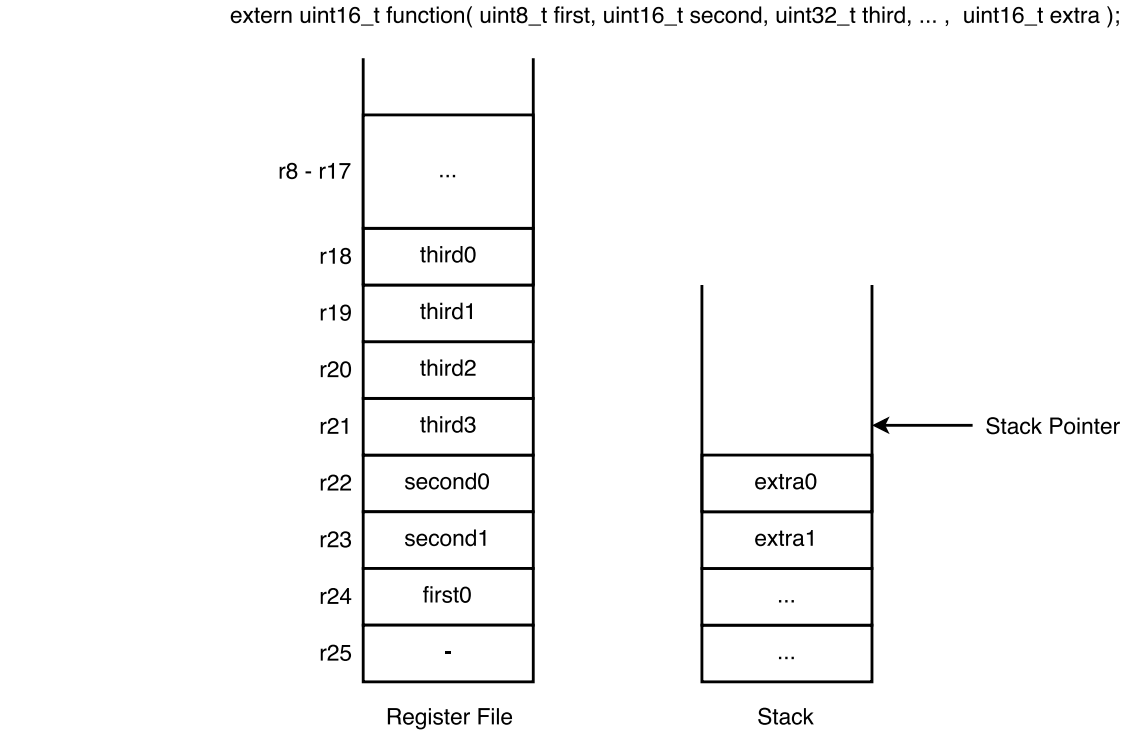


Figure 3.1: Parameters mapping

The return value must be stored in the same manner – starting from `r25` and maintaining the same byte order as passed arguments.

Passing the real key value as function argument would be source of leakage generated by the compiler, therefore we declare the variable as global and only use associated pointer.

3.1.2 avr-gcc compilation and optimization issues

The project in C is compiled to AVR platform executable code using `avr-gcc` compiler. The optimization level is set to `-Os`, the fastest and smallest code which is good for the efficiency and size of the executable but it might lead to unwanted changes in the implemented countermeasures.

3.1.2.1 Dummy cycles

One of the affected piece of code was the attempted dummy cycles insertion that was later abandoned because of reasons stated in chapter Design 2.3.3. The compiler is very effective in detecting the unnecessary pieces of code and optimizing them away. The following code is not transferred into machine code because it modifies a single variable(on top of the iterated variable) that is not used later in the execution:

```
uint16_t cycle;                 ; generated assembly,
uint8_t i;                      ; optimization
for(i = 0; i < 20; i++){        ; removed everything
        cycle++;
}
```

If we wanted to force the creation of the variable and all code that modifies it, we would have to declare it as `volatile` in C code or implement the dummy cycle as an assembly routine.

3.1.2.2 Operation `EOR`

Instruction `EOR` is not negatively affected by optimization, however its transition into assembly code causes a major leakage described previously in chapter Design 2.1.4.2.

The order of operands of `EOR` enables the leakage in the write back phase of the instruction execution in ALU, because the order of operands in `avr-gcc` generated assembly code does not change according to the order in the original C code but it rather depends on the destination variable in C.

The behavior of `avr-gcc` can be observed on the following example of three different C versions of exclusive-OR operation usage that turned out to have the exact same assembly code in the end:

```
uint8_t mask = 0x55;
```
```
ldi r24, 0x55
std Y+1, r24
```

```
uint8_t key = 0x22;
```
```
ldi r25, 0x22
std Y, r25
```

```
key = key^mask;
```
```
ldd r25, Y   ; load key
ldd r24, Y+1 ; load mask
eor r24, r25
std Y, r24 ; store key
```

```
key = mask^key;
```
```
ldd r25, Y   ; load key
ldd r24, Y+1 ; load mask
eor r24, r25
std Y, r24 ; store key
```

```
key ^= mask;
```
```
ldd r25, Y   ; load key
ldd r24, Y+1 ; load mask
eor r24, r25
std Y, r24 ; store key
```

We cannot count on the compiler that it will transfer our designed countermeasure in C into the same countermeasure in assembly.

Furthermore, we can see that value of the key is loaded directly from memory pointed by Y-pointer into register `r25`. The difference between the old value of `r25` and the new one in this example equals `0x00` because of the previous `LDI` instruction. This instruction is not used in the real implementation because the key is already stored in memory and loaded into register from there. The content of register that will load the key value is the same on each algorithm execution unless some random value was loaded into it.

This issue can be solved with random register precharging:

```
.extern precharge ; generated by rand() in C
.extern key

lds r31, precharge+1
lds r30, precharge
ld r23, Z

lds r27, key+1
lds r26, key
ld r23, X
```

3.2 Random numbers

ATMega163 itself does not have internal support for generating random numbers with true random number generator. In the designed countermeasures, only pseudo random numbers are used as an output of standard C function

`rand()`. Modulo by 256 is applied on the result of `rand()` to get it into one byte range mask.

The random number generator in the AVR version of standard library `avr-libc` is the standard linear congruential generator defined in [15]. For this platform, `RAND_MAX` is not defined as the highest $2^{31}-1$, but only $2^{15}-1 = 32767$.

The initial seed is not set and it defaults to `1`. If we disconnect the smart card and start again, the state of pseudo random generator will reset and start from the same initial state.

The linear congruential random number generator from the standard library is sufficient for this level of DPA, because it does not aim to exploit the weaknesses of the generator. If we wanted more randomized mask generation, we would need a source of entropy available to the smart card such as using built-in real-time clock or the most noisy bits of ADC. Another possible way to keep the information about number of reboots would be saving an initial state into EEPROM and incrementing it on every start-up and using it as new seed. This would guarantee different numbers in each restart, but it would still be predictable as we wouldn't have a way to truly randomize the increments.

3.3 Card firmware

The smart card runs a modified version of Simple Operating System for Smartcard Education (SOSSE)[1]. The card needs to be re-programmed with a programmer each time we change the executable. Another possible way to update the implemented code would be writing the Boot program code that would be able to update the Application program code section, but using the programmer is more straightforward.

The modified version of SOSSE is used and further developed for the purposes of the course Security and Hardware (MI-BHW) taught at Faculty of Information Technology, Czech Technical University in Prague.

3.3.1 APDU

SOSSE mediates the communication between our custom implemented functions and smart card interface. The communication is implemented using application protocol data unit (APDU). APDU is the communication unit between smart card and the reader defined in ISO/IEC 7816 [16]. The communication between the card and reader is divided into APDU command and APDU response.

[1] `http://www.mbsks.franken.de/sosse/html/index.html`

3.3.1.1 Command

The command is sent by reader to card. The length is at least 4 bytes of mandatory header containing the instruction class, instruction code and and two bytes of instruction parameters. The command might contain the body with length of sent data, data itself and expected maximum length of response data.

The class byte `CLA` says to what extent the command and response comply with the definition in ISO/IEC 7816 and the format of secure messaging and the logical channel number. Our `CLA` is defined as `0x80`, the first nibble says that the structure of command and response is used according to the definition except for features defined by the second nibble. The second nibble contains the information about secure messaging in two high bits and about logical channel number in two low bits. Our second nibble indicates that no secure messaging and no logical channel is used.

The parameters `P1` and `P2` might indicate offset for writing, but they are set to `0x00` in our case and not used.

The instruction byte `INS` indicates which instruction implemented by the card we request to be used. Our SOSSE supports several instructions and our implementation of AES-128 is invoked by `0x60` instruction code.

The length of sent data `Lc` can be defined as 0, 1 or 3 bytes. One byte length may contain values from 1 to 255. The maximum data length is indicated by 3 bytes, but the first byte must be set to zero for 3 bytes length, therefore it can only contain values from 1 to 65535. Zero length is only allowed for 0 byte length.

Our `Lc` is defined as `0x10` for one 16 byte block of plaintext data to be encrypted by the card.

The length of expected maximum length of response `Le` is defined by 0, 1, 2 or 3 bytes. Zero byte length defines zero bytes length of expected response. One byte defines the length in the range 1 to 256 where 0 means 256. Two bytes is used if the length of sent data was defined in the command. It's in the range 1 to 65536 where 0 means 65536. Three bytes is used when the length of sent data was not defined in the command. It's in the range 1 to 65535 with the first byte equal to 0.

Our `Le` is defined as `0x10` for one 16 byte block of encrypted data returned by the card.

3.3.1.2 Response

Sent by card to reader, contains the response data and two bytes `SW1-SW2` with command processing status.

The command processing status is the return code that indicates either success `0x9000` or one of possible errors, e.g. `0x6800` for unsupported instruction code and `0x6a00` for unexpected length of the command or of the expected response.

3.3.2 Instruction `0x60`

Our instruction `0x60` encrypts 16 bytes of data from input and returns 16 bytes on output. We expect that the length always matches 16 bytes and ignore any extra bytes. The encryption function receives allocated `input`, `output` and `key` buffers and saves the encrypted block into `output`. The implementation of the AES-128 algorithm is described in detail in the following section 3.4.

3.4 AES-128

The implemented AES-128 function encrypts one block of input using an embedded key. The operations

3.4.1 Key expansion

The key expansion took place at once before the algorithm execution in early implementation. It was later replaced with constant array containing the full expanded key, mainly because the key was not expected to change and the expansion taking place before each algorithm execution slowed down the measurement significantly.

The results of the simulation show that each round key except the original one takes 5386 clock cycles to compute and `expandKey()` in total takes 55410 clock cycles. The function is still present in the C source code, but it's never called.

The clock cycles were measured for implementation that used both S-box and `rcon` look-up tables stored in the Program memory, not computed each time.

3.4.2 Key addition

`AddRounKey(round_nr)` function implements the key addition layer of AES. The original unprotected key addition simply exclusive-ORed the key with the plaintext byte after byte.

This function was later rewritten in assembly language using the same interface. The enhanced `AddRoundKeyMasked(round_nr)` loads the random

`mask_in` and random `precharge` value from the global context. The random values are generated in C code using the Libc `rand()` function.

Before loading the round key, the destination register is precharged with the random `precharge` value. The register is precharged with the same value every time we load next byte of the key.

Next step is adding the `mask_in` value. Instruction `EOR` is executed with key value in the destination register and mask in the other one. The value leaking its Hamming distance during the register write back phase is the mask and not the key. The masked key byte is then exclusive-ORed with corresponding byte of plaintext.

3.4.3 SubBytes

The function `subBytes()` uses one byte of block as index into S-box look-up table stored in the Program memory and substitutes the byte for the value on that index.

3.4.3.1 S-box masking

S-box masking implementation is done by function `maskSbox()` that generates `mask_in` and `mask_out`, iterates over the S-box look-up table and fills in another look-up table with masked S-box. Masked S-box is created as:

```
// generate input and output masks
do{
    mask_in = rand()%256;
} while( mask_in == 0x00 );

do{
    mask_out = rand()%256;
} while( mask_out == 0x00 || mask_out == mask_in );

// fill in the masked S-box
uint16_t i = 0;
for( i = 0; i < 256; i++ ){
    sbox_masked[i^mask_in] = sbox(i)^mask_out;
}
```

The `mask_in` and `mask_out` are generated so that they are not same and do not equal zero. If the input mask equaled zero, the key value would not be masked at all in key addition operation and the output mask would be applied in S-box substitution.

If the output mask equaled zero, the key addition would be masked, but the S-box substitution would do mask removal and the value obtained by S-box substitution would be the unmasked product of key addition and S-box substitution. This value would be easily detected by the simplest Hamming weight consumption model and for the rest of the execution, the values would stay completely unmasked.

If the input and output masks equaled each other, we would be using the single mask masking scheme that was abandoned because of its inability to mask the Hamming distance between key addition and S-box substitution product.

These constraints might seem unnecessary and paranoid at first, but if our random generator had the same probability of generating each number in range 0 to 255, the probability of generating zero would be 1/256 and probability that the values will be equal would be 1/65536, assuming that two consecutive values are not dependent. Then for example, if we allowed the zero value for the output mask, we would obtain unmasked power traces leaking the Hamming weight of S-box substitution product once in 256 algorithm executions. If we measured more traces e.g. 5120, there would be a column in the measured traces that would contain 20 values directly dependent only on the plaintext and key while the rest of the values would be fairly random. If no other column achieved so many correlated values randomly, the correct key could be retrieved - the correlation coefficient would be probably low but it would be enough to beat the other key values.

3.4.4 ShiftRows

The function `ShiftRows` rotates each row to left by zero, one, two or three bytes. The rotation is done using a temporary variable and moving each byte directly to its new position instead of byte by byte. This speeds up the execution.

Function `ShiftRows` is masked with `mask_out` Boolean mask.

3.4.5 MixColumns

The multiplication by `02` and `03` polynomials from Rijndael finite field in `MixColumns` can be implemented either as function or look-up table. Both possibilities were examined and look-up tables were used for the final implementation. Although they take 2×256 bytes of additional memory, they speed up the execution.

Function `MixColumns` is masked with `mask_out` Boolean mask.

3.4.6 Removing mask

The output mask is removed at the end of the first round before adding the first extended round key. The full masking scheme of the first round is depicted by Figure 3.2.

3.4.7 Other rounds

The other rounds are not currently masked by the masking scheme. In early stages of implementation, all rounds were masked to test whether the output

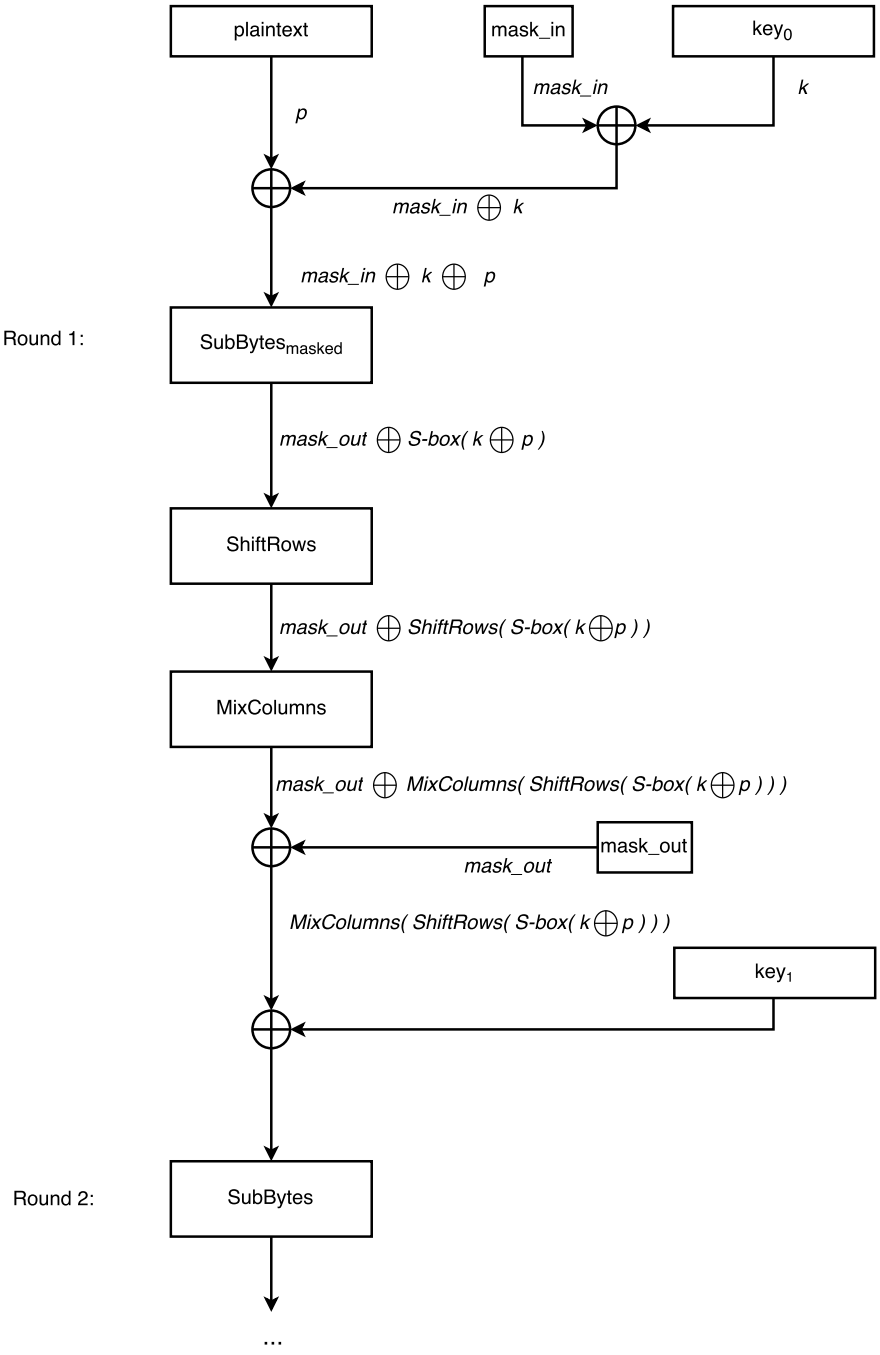


Figure 3.2: Masking scheme

will be successfully unmasked at the end of the execution. The output mask is removed at the end of the first round before `SubBytes` operation. The second

`SubBytes` operation is using the original unmasked S-box.

If we wanted to keep the operation masked, we could remove the output mask and apply the input mask again at the end of each round. This would enable us to use the same masked S-box for all rounds and avoid computing 10 different masked S-boxes. Or we would precompute 10 different S-boxes with previous output mask as the new input mask. Another possibility would be switching between two S-boxes – `mask_in` → `mask_out` and `mask_out` → `mask_in`.

In the final implementation, only the first round is masked because the attack is not targeted at the rest of them.

3.5 Differential power analysis

The differential power analysis was implemented using Wolfram Mathematica. The measured traces are loaded and examined visually to determine the best range for DPA.

3.5.1 Range

The chosen range should be wide enough for the key addition and S-box substitution in the first round to fit in, but the complexity of later correlation computation should be taken into account. If we miss the part where the targeted operations occur, we won't receive any results for the consumption model. On the other hand, too wide range will take longer to correlate with the model.

Figure 3.3 shows the plotted power consumption of one execution of algorithm without key expansion and without any masking scheme applied. Only five rounds fit into the trace with length of 1 million samples and we can clearly see that one round is represented by approximately 200000 samples.

For DPA on this unprotected implementation the samples from range $[50000, 150000]$ were used, which match the beginning of the first round, and the correct values were successfully acquired for all 16 bytes of the secret key using the Hamming weight model.

Finding the correct range for DPA starts to be more difficult when the power consumption is masked because the repeating round patterns tend to dissolve. If we want to be sure that DPA got the relevant samples, we can try to set all used masks and precharges to `0x00` and find the range on this unmasked version or measure the offsets of functions in simulator in clock cycles and map them to samples. Otherwise, we have to expand the range.

3.5.2 Used models

Two models were used for DPA on the unprotected implementation, on the implementation using single mask and on the implementation using input and

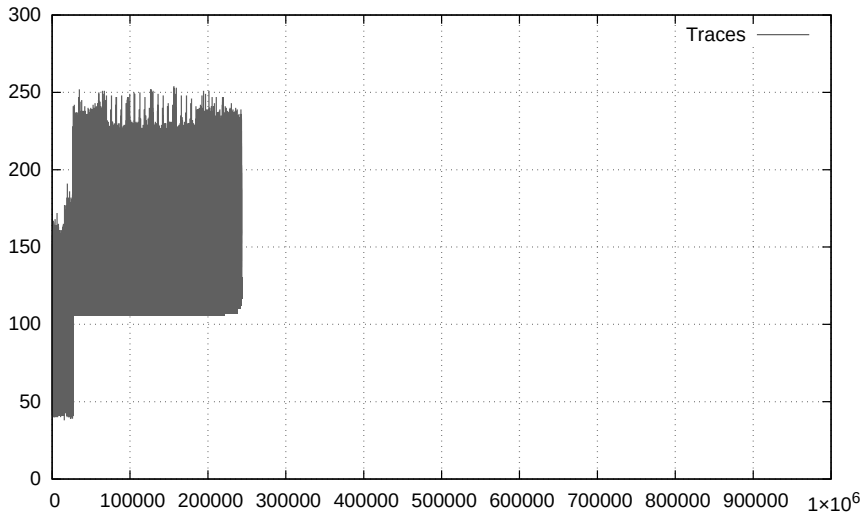


Figure 3.3: Traces of algorithm without masking and key expansion

output masks. For the attack on the first round only plaintexts were used, ciphertexts would be more useful when attacking the last round.

3.5.2.1 Hamming weight

The original DPA of unprotected implementation used the Hamming weight model and the key was acquired one byte at a time. The hypothetical model is computed as follows:

1. Generate all 256 possible byte values.
2. Apply `AddRoundKey(0)` on all pairs of keys × plaintexts.
3. Apply `SubBytes()` on the combinations of keys and plaintexts.
4. Compute the Hamming weight of each combination.
5. Correlate the resulting matrix with selected number of power traces and selected range.
6. Search for the highest correlation coefficient in the matrix.
7. The row containing the highest coefficient is the winning key value and the column marks the offset in trace when the value leaked. This can occur in more offsets because the Hamming weight will leak multiple times after `SubBytes` operation.

This model was successful in retrieving all bytes of the unprotected implementation with 200 measured plaintext inputs and the range of 100000 samples. The highest scoring key value always scored approximately twice as high correlation coefficient as the second best one.

This model was not successful against single mask masking scheme anymore because the value leaking the Hamming weight was already masked.

3.5.2.2 Hamming distance

The Hamming distance model makes a slight change to the computation in step 4:

1. Generate all 256 possible byte values.
2. Apply `AddRoundKey(0)` on all pairs of keys × plaintexts.
3. Apply `SubBytes()` on the combinations of keys and plaintexts.
4. Compute the Hamming distance of previous two operations.
5. Correlate the resulting matrix with selected number of power traces and selected range.
6. Search for the highest correlation coefficient in the matrix.
7. The row containing the highest coefficient is the winning key value and the column marks the offset in trace when the value leaked.

This model was able to break the implementation masked by the single mask masking scheme because of its design flaw - unmasked S-box difference. The acquired winning key bytes also scored significantly higher correlation coefficients than other bytes, but the difference wasn't as high as with the unprotected implementation. Number of input plaintexts and the range pf samples was not changed.

When applied on input and output mask masking scheme with register precharging, this model was not able to acquire any key byte correctly and the highest correlation coefficients of all bytes were similar. Adding few hundreds of measured traces was not enough to acquire the key. The real secret key did not even appear in top three guesses. More about key guessing entropy and implementation evaluation will be described in the chapter Evaluation 4.

3.5.3 Used tools

For AES and countermeasures implementation, AVRStudio 4.19 (Build 730)[2] with WinAVR 20100110 [3] tools and GCC compiler were used . The hardware device hosting the implementation was ATMega163 smart card.

For measurement of power consumption, Agilent DSO-X 3012A oscilloscope was used. The input and output were controlled by SC Power Measurement program. The program uses the linear congruential random number generator from the standard library and the generated plaintexts sequence is always the same for each session.

For DPA of measured traces, Wolfram Mathematica 10.3.0.0[4] was used. Although it provides many features that make the DPA implementation easier, it shows great difficulties in processing larger amounts of data. For the purpose of quick data visualization, gnuplot[5] command line utility was used.

[2] http://www.atmel.com/tools/studioarchive.aspx
[3] http://winavr.sourceforge.net
[4] https://www.wolfram.com/mathematica/
[5] http://www.gnuplot.info

Chapter 4

Evaluation

This chapter will sum up the achievements of DPA and the ability of countermeasures to prevent some of them. The performance of protected implementation is also evaluated and compared with the original unprotected implementation.

4.1 Aspects of evaluation

When assessing the characteristics of the implemented countermeasures, we care about both their quality and their efficiency.

4.1.1 Quality

One of the most important quality of DPA is the ability to obtain the secret key. In case the DPA was able to acquire the correct secret key, we care about the correlation coefficient that the value received according to the power consumption model.

4.1.1.1 Guessing entropy

If one or more bytes of the secret key were guessed incorrectly, we want to determine what rank the correct key received. In the latter case, we need to examine whether the distance of the correct key from the top is consistent throughout all bytes or its position appears random.

4.1.1.2 Correlation coefficients

We also care about the correlation coefficient of the winning value and its distance from the other best scoring values. If the distance is greater, e.g. the first one scored 0.7 and the second one and the rest only scored less than 0.3, we are more confident that the obtained value is correct. If the first one scored a smaller (< 0.5) correlation coefficient similar to other top guesses,

it's more likely that it's incorrect. This behavior was observed in the executed DPA consistently.

4.1.2 Efficiency

In encryption, the security and the speed of implementation must be balanced, especially if working with devices such as smart cards. If the smart card is supposed to be used as an entry card into building, the encryption algorithm implemented on it that processes the challenge-response cannot afford to take too much time.

4.1.2.1 Clock cycles

To compare the changes in speed of implementations with and without applied countermeasures, the clock cycle counter in AVRStudio simulator for ATMega163 was used.

4.2 Results

4.2.1 Quality

4.2.1.1 Unprotected implementation

First, the unprotected implementation was measured and analyzed. For Hamming weight and Hamming distance models targeting the output of `SubBytes` operation in the first round or the difference from `AdddRoundKey(0)`, the results in Table 4.1 were obtained.

	HW	**HD**
Number of inputs	300	300
Range	200000	200000
Correct bytes	16/16	16/16
Average coefficient of #1	0.837705	0.779386
Average coefficient of #2	0.367906	0.313598
Average distance #1-#2	0.469799	0.465789

Table 4.1: Unprotected implementation quality

Both models were successful in modeling the power consumption and obtaining all bytes correctly, but Hamming weight got slightly better correlation coefficients of the winning key value on average. After examining the correlation matrix highest coefficients, it was discovered that with Hamming weight

model the matrix contained multiple high scores of the same byte, meaning that the value leaked multiple times during the execution. This was caused by the fact that in ideal case, the output of `SubBytes` appears in the trace at least four times, twice right after `SubBytes` and then twice shortly after at the beginning of `ShiftRows`. Hamming distance leakage only happens twice in total during the `SubBytes` operation - first when updating the value in register and then when updating the value in SRAM.

4.2.1.2 Single mask

The single mask masking scheme DPA quality assessment can be seen in Table 4.2. The results show that with single mask applied, the Hamming weight model was no longer successful while the Hamming distance model targeting the `AddRoundKey(0)` and `SubBytes` operations was still able to retrieve all bytes of the key correctly. The guessing entropy of Hamming weight model was fairly random and the correct key did not place consistently on similar rank. The small distance in correlation coefficients of Hamming weight model's top two guesses shows that DPA was not able to find any correlated sample and therefore the highest of all the low coefficients won.

	HW	**HD**
Number of inputs	300	300
Range	200000	200000
Correct bytes	0/16	16/16
Average coefficient of #1	0.302922	0.640829
Average coefficient of #2	0.289554	0.304652
Average distance #1-#2	0.0133679	0.336177
Maximal guessing entropy	244	1
Minimal guessing entropy	9	1
Average guessing entropy	130.812	1

Table 4.2: Single mask masking scheme quality

Originally, two versions of single mask masking scheme existed – one with S-box output register write back masked by changing the destination registers in assembly and the one used for this evaluation.

4.2.1.3 Input and output mask

When evaluating the input and output mask masking scheme, neither Hamming weight nor Hamming distance models were successful in obtaining any

bytes of the secret key and the correct key never appeared on similar position. Detailed results can be seen in Table 4.3.

	HW	**HD**
Number of inputs	300	300
Range	200000	200000
Correct bytes	0/16	0/16
Average coefficient of #1	0.301019	0.298037
Average coefficient of #2	0.292493	0.291232
Average distance #1-#2	0.00852519	0.00680513
Maximal guessing entropy	251	230
Minimal guessing entropy	10	44
Average guessing entropy	121.375	138.188

Table 4.3: Input and output masks masking scheme quality

Number of inputs was chosen to be the same as with the previous unprotected and single mask implementations to illustrate the difference in DPA difficulty.

4.2.2 Efficiency

The number of clock cycles was first measured for unprotected implementation with all functions in C. The original implementation was compared with clock cycles measured for the implementation with input and output mask masking scheme applied. The single mask masking scheme was not added into measured implementations in Table 4.4, because it uses almost the same functions as the input and output mask with the only exception of drawing only one random mask. This difference was addressed by adding clock cycle measurement for operation `rand()%256` to show its total cycles contribution.

From the measured clock cycles, it's clearly visible that the masked implementation runs longer. However, there are functions that were made faster by rewriting them in assembly - `AddRoundKeyMasked(nr)` and `SubBytesMasked()`. The first one is made faster by omitting longer offset computing in `avr-gcc` generated code. `avr-gcc` compiled code tend to use more general arithmetic operations for computation of offsets so that the emitted code works in all cases. We can afford to use more simplistic and straightforward instructions, e.g. we know that we will always read and write 16 bytes in the routine therefore we can use immediate value for the loop and read consecutive bytes from the memory in one loop.

The most clock cycles added by the masked implementation are from the `maskSbox()` function. This function iterates over 256 values and executes exclusive-OR twice for each value. Computation of masked S-boxes, as was discussed in previous chapters, introduces significant memory requirements and degrades performance, therefore we have to limit the number of computed masked S-boxes.

	Unprotected	**I&O masks**
`encrypt()`	77,661	103,491
`rand()%256`	not implemented	1,691
`maskSbox()`	not implemented	9,060
`AddRoundKey(nr)`	1,063	
`AddRoundKeyMasked(nr)`[6]	not implemented	229
`SubBytes()`	357	
`SubBytesMasked()`[6]	not implemented	228
`ShiftRows()`	4,875	
`MixColumns()`	1,293	
`unmaskText()`	not implemented	417

Table 4.4: Measured clock cycles

Possible improvements to implementation that would alleviate the added operations cost could be achieved by rewriting `AddRoundKey(nr)` in assembly or switching to fixed set of masks. However, rewriting the functions in assembly makes them incompatible with other platforms while `C` code can usually be compiled into machine code without change.

[6]The function is implemented as an assembly routine, not C function.

Conclusion

Summary

In this thesis, we've described the power side-channel attack, differential power analysis methods and possible ways to prevent the exploitation of AES-128 encryption implementation on real-life hardware devices. The most discussed countermeasures against DPA belonged among masking countermeasures and this work dealt exclusively with the software ones.

A simple masking scheme using single mask was designed, evaluated and deemed insufficient after introducing better consumption model based on Hamming distance instead of Hamming weight. This masking scheme's flaws were described and a better one superseded it – input and output mask. The new masking scheme was successful in protecting the secret key values against first order DPA of operations in the first round of algorithm execution.

Meeting our expectations, the masking scheme in `C` language was not secure, because we've lacked the control of how the values are actually handled and which instructions are executed by the processor. Compiler-emitted code did not keep our intended countermeasures intact and it was necessary to rewrite the most critical functions as AVR assembly routines.

Our final masking scheme is complemented by the suitable language choice. Other countermeasures were considered, some of them such as random register precharging were added to the implementation while others were abandoned because of their inefficiency.

The unprotected, single mask and input and output mask implementations were evaluated in the last chapter. The DPA analyzed the first round of the algorithm and focused on the operations `AddRoundKey(0)` and `SubBytes` using both Hamming weight and Hamming distance model of power consumption. The inevitable performance degradation was illustrated by measured clock cycles for the final masked implementation. Random number generation and S-box conversion turned out to add the extra costs the most. On the other hand, some functions could be optimized by rewriting them completely in

assembly and giving up on portability.

What the implementation platform lacks the most is a strong random number generator. ATMega163 offers very limited options for generating random numbers and creating a true random generator on this platform is out of scope of this thesis.

Future work

The design and implementation of the final masking scheme could be further improved. Firstly, it could cover `ShiftRows` and `MixColumns` operations, secondly, it also could be extended to more rounds. The possible application on more rounds was discussed, but because the original attack targeted only first round and the two specific operations, the countermeasures focused on them as well.

Extending the masking scheme on all rounds would not degrade the performance as much as the S-box conversion computation, but it was not deemed necessary because of the DPA used.

The final implementation divides the secret into two shares, which means it's effective against the first order DPA. There is definitely space for DPA of higher orders that can demonstrate the high order DPA abilities.

The tool used for DPA implementation was Wolfram Mathematica, but when processing large data inputs, it had difficulties with own memory-management and exporting visualizations. The large data inputs processing could be automatized to enable traces larger than 1GB to be correlated without running out of memory. An interesting option would be processing the measured 1MB traces, knowing which time offsets are useful for the DPA, and importing the pre-processed traces into Mathematica instead.

Bibliography

[1] Liu, J.; Yu, Y.; Standaert, F.-X.; et al. Small Tweaks Do Not Help: Differential Power Analysis of MILENAGE Implementations in 3G/4G USIM Cards. In *Computer Security–ESORICS 2015*, Springer, 2015, pp. 468–480.

[2] Kocher, P.; Jaffe, J.; Jun, B. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, Springer, 1999, pp. 388–397.

[3] Stefan Mangard, T. P., Elisabeth Oswald. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007, ISBN 978-0-387-38162-6, 338 pp.

[4] Daemen, J.; Rijmen, V. Resistance against implementation attacks: A comparative study of the AES proposals. *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, volume 82, 1999.

[5] Messerges, T. S. *Fast Software Encryption: 7th International Workshop, FSE 2000 New York, NY, USA, April 10–12, 2000 Proceedings*, chapter Securing the AES Finalists Against Power Analysis Attacks. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, ISBN 978-3-540-44706-1, pp. 150–164, doi:10.1007/3-540-44706-7_11. Available from: `http://dx.doi.org/10.1007/3-540-44706-7_11`

[6] Daemen, J.; Rijmen, V. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002, ISBN 3-540-42580-2, 238 pp.

[7] Chari, S.; Jutla, C. S.; Rao, J. R.; et al. *Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings*, chapter Towards Sound Approaches to Counteract Power-Analysis Attacks. Berlin,

Heidelberg: Springer Berlin Heidelberg, 1999, ISBN 978-3-540-48405-9, pp. 398–412, doi:10.1007/3-540-48405-1_26. Available from: http://dx.doi.org/10.1007/3-540-48405-1_26

[8] Bhasin, S.; Bruneau, N.; Danger, J.-L.; et al. *Security, Privacy, and Applied Cryptography Engineering: 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings*, chapter Analysis and Improvements of the DPA Contest v4 Implementation. Cham: Springer International Publishing, 2014, ISBN 978-3-319-12060-7, pp. 201–218, doi:10.1007/978-3-319-12060-7_14. Available from: http://dx.doi.org/10.1007/978-3-319-12060-7_14

[9] DPA contest v4 [online]. 2015, [Cited 2016-04-22]. Available from: http://www.dpacontest.org/v4/index.php

[10] Atmel Corporation. *8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash ATMega163 [online]*. 2009. Available from: http://www.atmel.com/images/doc1142.pdf

[11] Atmel Corporation. *Two-wire Automotive Temperature Serial EEPROMs [online]*. 2012. Available from: http://www.atmel.com/Images/doc5121.pdf

[12] Atmel Corporation. *Atmel AVR 8-bit Instruction Set [online]*. 2014. Available from: http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf

[13] Atmel Corporation. *Mixing Assembly and C with AVRGCC [online]*. 2012. Available from: http://www.atmel.com/images/doc42055.pdf

[14] AVR Libc Reference Manual [online]. 2016, [Cited 2016-04-29]. Available from: http://www.atmel.com/webdoc/AVRLibcReferenceManual/FAQ_1faq_regbind.html

[15] Park, S. K.; Miller, K. W. Random number generators: Good ones are hard to find. *Communications of the ACM*, volume 31, 1988.

[16] ISO/IEC 7816-4:2005. 2013, [Cited 2016-04-30]. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=36134

Appendix A

Acronyms

AES Advanced Encryption Standard

APDU Application Protocol Data Unit

CMOS Complementary Metal–Oxide–Semiconductor

DPA Differential Power Analysis

ECB Electronic Codebook

EEPROM Electrically Erasable Programmable Read-Only Memory

GCC GNU Compiler Collection

LEMS Low Entropy Masking Scheme

RISC Reduced Instruction Set Computing

RSM Rotating Sbox Masking

SCA Side-Channel Attack

SOSSE Simple Operating System for Smartcard Education

SPA Simple Power Analysis

SRAM Static Random Access Memory

Appendix B

Contents of enclosed DVD

- `readme.txt` ..... the file with DVD contents description
- `hex` ..... the directory with hex files
- `dpa` ..... the directory with DPA examples
- `src` ..... the directory of source codes
  - `firmware` ..... card firmware sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format