FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Masked AES cipher on a microcontroller and Second-order DPA
Student:	Abdullah Bhatti
Supervisor:	Ing. Jiří Buček
Study Programme:	Informatics
Study Branch:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	Until the end of summer semester 2018/19

Instructions

Study the Differential Power Analysis (DPA) attack method and countermeasures against it by masking, including attacks on masking using Second-Order DPA. Program a masked AES implementation on an AVR microcontroller. Perform the first-order and second-order DPA attacks on the masked AES implementation and evaluate the complexity and success rate of the attacks.

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 15, 2018

Czech Technical University in Prague

Faculty of Information Technology

Department of digital design

Master's thesis

Masked AES cipher on a microcontroller and Second-order DPA

Bc. Abdullah Bhatti

Supervisor: Ing. Jiří Buček, Ph.D.

17th December 2018

Acknowledgements

I would like to start off by thanking God, dad, mom, Usama and Zainab for their unconditional love and support during my thesis. Furthermore, I would like to express my gratitude towards Ing. Jiří Buček, Ph.D. for giving me the chance to work on this project and helping me out every step of the way. His knowledge and patience has helped me immensely.

Besides my advisor, I would like to thank the Czech Technical University in Prague for giving me the opportunity to study in this institution in Europe. I would also like to thank Bilal Ahmed, Kříž Sergio and Jirka Kocum, amongst other friends for their help and support.

Declaration

In Prague on 17th December 2018 ....

Citation of this thesis

Bhatti, Abdullah. *Masked AES cipher on a microcontroller and Second-order DPA*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

Práce je zaměřena na možné útoky pomocí diferenciální analýzy výkonu na maskované implementaci AES-128 na mikrokontroleru AVR. Implementace je rozdělena do tří částí. První část obsahuje nemaskované verze AES-128. Druhá část používá konstantní hodnotu masky nastavené v celém kódu. Třetí část generuje náhodnou masku, která je použita na 10 kol. Nakonec jsou DPA prvního řádu a DPA druhého řádu a DPA druhého řádu použity k pokusu rozbití šifry a nalezení klíče.

Klíčová slova Differential power analysis, masked, cipher, attacks, AES-128, AVR microcontroller.

Abstract

The thesis is focused on possible attacks using differential power analysis on a masked AES-128 implementation onto an AVR microcontroller. The implementation is divided into three parts. The first part comprises of an unmasked version of AES-128. The second part uses a constant value of mask used throughout the code. The third part generates a random mask which is then applied to the 10 rounds. Finally, first order DPA and second order DPA are used to try and break the cipher and find the key.

Keywords Differential power analysis, masked, cipher, attacks, AES-128, AVR microcontroller.

Contents

List of Figures

Introduction

Encryption is a part of cryptography where information is secured such that only you and the people who the information is intended for can read it. The original text, referred to as the plaintext is converted into a ciphertext after undergoing some operations. For the project, we have used the Advanced Encryption Standard (AES).

Advanced Encryption Standard (AES), is a symmetric encryption algorithm that was designed to be a secure cipher being mathematically strong and resistant to cryptanalysis. However, like most ciphers, attacks emerge that exploit the smallest flaws within the design. One of these attacks include the Power analysis attacks that present a grave threat to the implementation of these algorithms. Over the years, countermeasures have been developed to protect the implementation against differential power analysis attacks.

A masked implementation of AES has been frequently used as a countermeasure against these attacks where the intermediate results are randomized during the computation of the algorithm to make the power consumption independent of the intermediate values of the algorithm.

Chapter 1

Analysis

1.1 What is the Advanced Encryption Standard (AES)?

Advanced Encryption Standard (AES) is a symmetric block cipher which officially came into existence in 2001 by the US National Institute of Standards and Technology (NIST). It was the much-needed successor of the Data Encryption Standard (DES) which was found to be vulnerable to brute force attacks [4]. The algorithm chosen for AES was submitted by two Belgian cryptographers, Joan Daemen and Vincent Rijmen .

AES has three distinct types of block ciphers, namely AES-128 [5], AES-192 and AES-256 where the numeric represent the bit-length of the key.

1.2 Working of AES

AES algorithm involves several computations which are done on bytes and operations which involve various substitutions and bits shuffling. These operations are done repeatedly in rounds. The number of rounds differ with respect to the type of block cipher being used. AES-128, which has been used in this project, comprises of a total of 10 rounds. AES-192 contains 12 rounds and finally, AES-256 has 14 rounds.

Figure 1.1 shows the general schematic of the AES-128 structure [6]. The left side of figure 1 shows the progression of 10 rounds and the 128-bit input round key which is generated from the original key. The round keys of all the rounds differ from each other in value. The right side shows the structure of each round and their four sub-processes. The last round, however, does not contain the MixColumns operation.

The 128 bits of the input data are processed as 16 bytes which are arranged in a 4×4 array shown in 1.2:

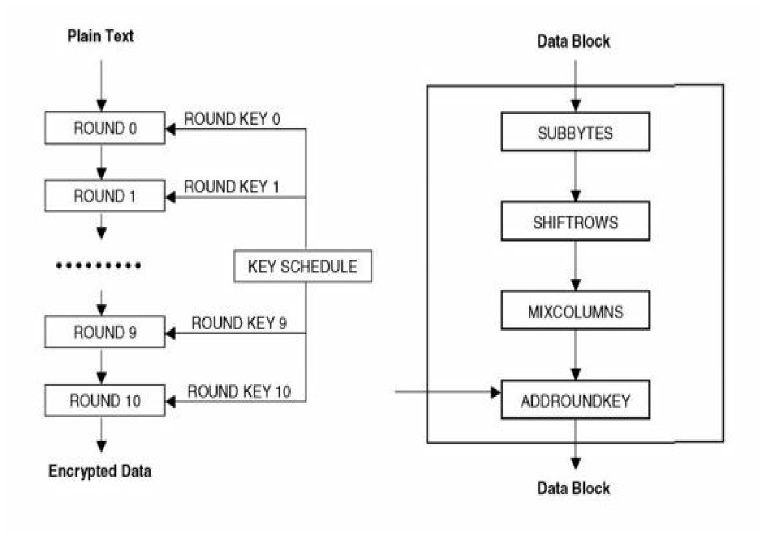


Fig. 1.1. Data path for AES-128 and the structure of one internal round

Byte_0	Byte_4	Byte_8	Byte_{12}
Byte_1	Byte_5	Byte_9	Byte_{13}
Byte_2	Byte_6	Byte_{10}	Byte_{14}
Byte_3	Byte_7	Byte_{11}	Byte_{15}

Fig. 1.2. Input data array for AES-128

These are referred to as the state or data path. The first four bytes are organized in the first column, the next four in the second, and so on. According to our project, we are going to use Electronic Codebook (ECB) encryption mode which makes the use of a single block.

Prior to the operations performed during the rounds, the first four words of the key are XORed with the input state array. During the encryption process, we use the operations mentioned in figure 1 respectively. The output of the first three steps are XORed with four words from the key schedule.

1.3 Mathematical operations and structures of AES

The operations involved in the AES design are simple. In case that round constant (Rcon), Rijndael field multiplication and S-box are stored in memory, there are no other arithmetical operations.

1.3.1 Round Constant (Rcon)

The operation of Rcon is defined as follows:

$$rcon(i) = x^{i-1} \bmod x^8 + x^4 + x^3 + x + 1 \tag{1.1}$$

Operations are performed in the finite field $GF(2^8)$.

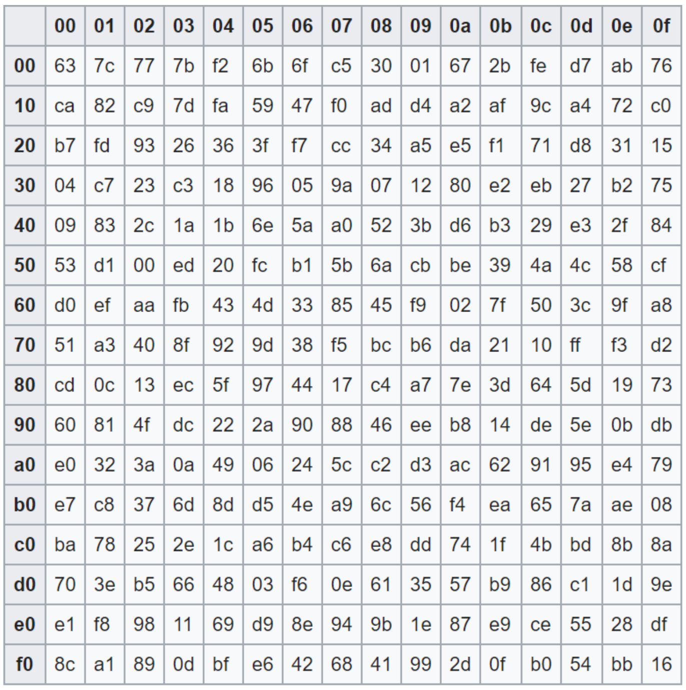
The round constants help in changing each round slightly than the other to avoid symmetry. This gives it protection from various attacks that try to exploit symmetry such as the slide attack.

The 32-bit input word of an expanded key is rotated to the left by 8 bits after which each byte is transformed using the S-box and finally, the first byte is XORed with rcon. For our case, we have a total of 10 rcon values as 11 round keys are required.

The derived rcon values can be stored in memory as a look-up table to speed-up the execution. Rcon is also essential in the the computation of the S-box which we are going to be talking about next.

1.3.2 S-box

The S-box is a fixed table which can be edited known as a substitution box or a look up table shown in 1.3:



	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0a**	**0b**	**0c**	**0d**	**0e**	**0f**
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig. 1.3. S-box for AES-128

The computation of the S-box involves a series of operations starting with the multiplicative inverse in:

$$GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1) \tag{1.2}$$

This is done using the Extended Euclidean Algorithm. Thereafter, the inverse is then transformed using the affine transformation. It is a non-linear transformation defined in 1.4.

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

Fig. 1.4. Affine transformation

Even though the non-linearity helps in securing the algorithm, it also means additional amount of memory usage and more steps needed for masking countermeasures.

1.3.3 SubBytes

SubBytes is one of the four sub processes that are performed in almost every round. It is a simple byte substitution which is looked up from a fixed table named the S-box. The bytes shown in figure 1.2 are each substituted according to the S-box. For instance, if we have a value of $0xa2$, the value substituted would be $0x3a$. Similarly, all the bytes in the matrix are replaced.

1.3.4 ShiftRows

The ShiftRows operation involves shifting of all the rows towards the left. The left most byte is reinserted on the right. However, the operation is slightly different for each row in terms of how much shifting is going to be carried out. The first row is not shifted or is shifted by 0. The second row is shifted one-byte position to the left, the third is shifted by two and the fourth row is shifted by three.

1.3.5 MixColumns

The MixColumns operation involves using each column as a four-term polynomial and performing modular multiplication whose coefficients are elements of $GF(2^8)$. Multiplication is carried out using the modulo $x^4 + 1$.

The first column of the new block is computed as follows:

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

Fig. 1.5. Computation matrix used during MixColumns

In the matrix shown in figure 1.5, C_0, C_1, C_2 and C_3 are the results after the transformation takes place. B_0, B_5, B_{10} and B_{15} are the first column of the new block. From the original block shown in figure 1.2, we can see the positions of these bytes before the transformations took place.

As opposed to ShiftRows, the same operation is performed on each column. MixColumns does not occur in the last round.

1.3.6 AddRoundKey

AddRoundKey uses the logical operation XOR. The 16 bytes of the 4 x 4 array are taken together as 128 bits and are XORed to the 128 bits of the round key. In the final round, this gives us the ciphertext.

The initial execution of the AddRoundKey operation is extremely vulnerable and in case that countermeasures have not been applied, the original key is exposed through the power consumption.

1.4 Attacks on Cryptographic Devices

Over the years, numerous kinds of attacks on cryptographic devices have come into existence which have the sole purpose of trying to retrieve the secret key that is being used in the encryption. These attacks have been categorized into passive and active attacks.

1.4.1 Passive and active attacks

Passive attacks make the use of the physical properties of a device to attain the secret key like the power consumption or electromagnetic radiation of circuit during execution. The device is operated within its specification.

Active attacks are fault injection attacks in which we inject the fault at the boundary of the chip. An active attack is carried out by tampering with the input or the environment of the device. The secret key is found out by exploiting the abnormal behavior of the device.

1.4.2 Invasive, semi-invasive and non-invasive attacks

An invasive attack is a reliable and strong attack that can be carried out on a cryptographic device. The attack involves direct access to the device and its various components using a probing station. In case where the probing is done only to observe signals, the attack falls into the passive attack category. However, if the signals in the device are altered in any way in attempts to change the functionality of the device, it becomes an active attack.

Like in invasive attacks, semi-invasive attacks start with the depackaging of the device. However, these attacks do not involve direct electrical contact onto the chip. The purpose of these attacks is to be able to read the content in the memory without using or probing the normal read-out circuits.

In non-invasive attacks, the attacker does not try to depackage the device or try to access various components of it. Only the directly accessible interfaces are used and nothing is changed. In doing so, the attacker does not leave any evidence of an attack.

We are going to be focusing on side channel attacks which are passive non-invasive attacks meaning we do not place anything on the chip that might alter its functionality or introduce glitches. We passively snoop the data and use it to find the secret key. The upcoming sections are going to talk about the different kinds of attacks that we can use.

1.4.3 Power consumption

This attack exploits the fact that the operations performed, and the processing of the data generates different amounts of current and this difference in power consumption is used to reveal the secret key [2]. This will be talked about in further detail in 1.5.

1.4.4 Electromagnetic emanation

There is a discharge of electromagnetic signals by the circuit during the processing of the algorithm. The attack is successful against implementations where there are several operations being performed. Since each type of operation emits a different amount of radiation, a trace of the EM signal may help in figuring out the operation that was performed and in turn, helping the attacker in getting the full or partial private key.

1.4.5 Timing attacks

This attack exploits the time required for the execution of the operations and the algorithm. These are successful on algorithms that are data dependent.

For our case, we are going to talk about the power consumption side channel attack.

1.5 Power analysis attacks

As mentioned before, the attack makes use of the differing power consumptions in the algorithm due to the operations that are performed and the data that is processed. This attack is perhaps the hardest to control.

The microcontroller chosen for this project, ATMega163, uses the CMOS technology which operates with digital data. The power consumption of CMOS circuits mainly comprises of two components: static power and dynamic power. Static power is dependent only on the design of the circuit since it is the current leakage of the transistors. Dynamic power, on the other hand, is dependent on the data that is being processed or the operations performed during the algorithm and occurs because of the switching between transistors. Dynamic power is what we are interested in as we need to look at intermediate values, preferably during the first round of the encryption in the AddRoundKey and SubBytes process. This is because, during these two processes, the original secret key and plaintext are being used. Furthermore, the variation of the total power is mostly due to the changes in the dynamic power rather than the static power since that remains somewhat constant and because of that, we can easily make use of the net power consumption for the attack.

For every attack, a power model is needed so that we can map the values during the processing of the algorithm by the smart card to power consumption values. We used the Hamming distance and the Hamming weight models. In order to understand what the Hamming distance and Hamming weight imply, lets look at the following example:

$$a = 11110110, b = 01110101 \tag{1.3}$$

$$HW(a) = 6, HW(b) = 5 \tag{1.4}$$

$$HD(a, b) = HW(a \oplus b) = 3 \tag{1.5}$$

Equation 1.3 shows two 8-bit numbers which are going to be used to calculate the Hamming weight and Hamming distance. The Hamming weight of a value is the number of 1s in its binary representation. Equation 1.4 gives us the result for the Hamming weight of the two binary numbers a and b. Finally, the Hamming distance, as shown in equation 1.5 is the Hamming weight of the XOR of the two binary values.

The Hamming distance model counts the number of transitions from 0 to 1 and 1 to 0 within the digital circuit. The observed number of transitions give us information about the power consumption of the device. Using these transitions, we can make a power trace that shows us the Hamming distance. The Hamming weight model is used while performing differential power analysis on the processed traces and will be talked about in 1.5.2.

The ATMega163, like most microcontrollers is built with registers, data bus, memory, arithmetic logic unit (ALU), and such other components. Typically, the data bus is connected to most of these due to which its capacitive load is quite high. Hence, it plays an important role in the power consumption of the device. Also, the loads on the individual wires of the bus are almost equal which is of concern since we assume that there is no difference between the wires or cells. Also, the registers, another component of the micro controller, work according to the clock signals and thus, only change their value after every clock cycle and the Hamming distance can then be calculated for consecutive clock cycles. The Hammming distance, however, is always tightly coupled to the hardware and since we do not know the internal micro-architecture of the controller, we can not be sure of the leakage. In case we have a register which would contain the result of AddRoundKey and then we loaded a value onto that register from the S-box, then the Hamming distance would be a good option to use.

1.5.1 Simple power analysis

In this approach, the attacker measures the power consumption during the cryptographic operations and uses those traces to find possible leakage points that may in turn be used to get some processed values. This technique is easy to execute but the least powerful approach because of which it is used more commonly in other algorithms like the square and multiply in RSA. The square and multiply algorithm is very easy to attack in most implementations since it squares and multiplies if the bit is 1 and in case the bit is 0, it only squares. These operations are clearly visible in a single power consumption trace, giving away the secret key.

1.5.2 Differential power analysis

The DPA is a method where we make the use of a set of power consumption measurements using different inputs by comparing it with a chosen measurement that contains all possible combinations of the secret key and then finding the one with the best correlation. This is especially useful where there is a lot of noise present since the large amount of measurements used help in finding the best correlation. DPA does not require detailed knowledge regarding the cryptographic device.

Over the course of time, masking has become a popular approach used to protect one from DPA. This works well against single order differential power analysis. However, an attacker may do multiple correlations of the power consumption in the scenario where the key is being used several times. In our case, second order DPA could be bought to use since the secret key is used for key expansion and then afterward, for encryption. On the other hand, these attacks are recognized as highly complex in nature and hard to execute because detailed knowledge regarding the cryptographic device is a prerequisite for its success.

DPA was the main method used to attack the key bytes. When taking a sample of say, 200 plaintexts, we can apply our power model on a matrix of intermediate values. We consider each byte separately and hence, take the first key byte which has a total of 256 possibilities. Using this information, we get a 200 x 256 matrix. Each element of the matrix contains the number of 1s in the result of the operations AddRoundKey and SubBytes in the first round of the algorithm. This is where the algorithm is applied and then further, used to find the correct key.

1.5.2.1 Higher-order DPA

Higher order DPA attacks are used when the implemented AES-128 scheme is not vulnerable to single order DPA. This is usually the case when the AES is masked. The attack exploits the leakage of two intermediate values which are related to each other because of the same mask. However, the two values may occur at different points in the power consumption since they may be found in two different operations of the algorithm because of which preprocessing of the traces is necessary so that we attain power consumption values that depend on both the intermediate values.

There are mainly three cases that occur practically when doing preprocessing. The first case is when the intermediate values are in different clock cycles. In this scenario, the two points in the trace are combined before the application of DPA. The second case is when the intermediate values are found in a single clock cycle. In this case, the preprocessing is applied to single points in the trace. The third scenario is when the values are found to be within one clock cycle and the power consumption characteristics are such that the preprocessing step may not be needed at all. Our case involved the first case, which is typical for software implementations.

The first step of a second-order DPA attack involves choosing two intermediate values u and v. Since we are looking at masked implementations when applying second-order DPA, we should know that these two intermediate values do not occur in the device. This is because only the masked intermediate values $v_m = v \oplus m$ and $u_m = u \oplus m$ are present in the device. Once the intermediate values have been chosen, we measure the power traces and start the preprocessing step. Usually, we are not aware of the exact points of where the two intermediate values, u_m and v_m occur, but we can make an educated guess about the interval that they might be in. The preprocessing function that is chosen is then applied to all points within that interval. Several kinds of preprocessing functions exist but the absolute difference preprocessing function is the most frequent one used. The next step involves calculating hypothetical values which are done so by computing the XOR of u_m and v_m. Supposing the result of the computation of $u_m \oplus v_m$ is w, we map w to the hypothetical power consumption values and compare it with the preprocessed traces.

1.5.3 DPA with correlation coefficients

Differential power analysis, combined with correlation coefficients, becomes a much stronger tool. 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 differ when values from the other one change. Correlation does not imply causation and the two sets might be independent, it only claims that statistically the two sets appear to be dependent. The degree of correlation can be measured by correlation coefficients.

1.6 Countermeasures

Instead of figuring out the weaknesses in the algorithm implemented, the attacks that are based on the information that is gathered from the implementation of the system itself, are called side channel attacks. They are not in abundance, however, due to their unique nature, their impact is higher than a significant other. The countermeasures to avoid these threats could be found either in hardware or software form. The software countermeasures are cheaper and are used to increase the level of security on devices that are not secure from hardware.

1.6.1 Hardware design

Hardware designing can include various techniques for countermeasures. Power consumption can be largely altered in hardware; hence it is a useful tool for countermeasures.

We may use masked logic gates which serves as an alternative to masking at algorithm level. No value that is stored in the hardware corresponds to the intermediate values. The idea is to never let the true secret value expose until we have the final output by masking the gates input and output signals.

For example, a regular 2-input AND gate ($y = a.b$) is transformed into a 5-input (a_m, m_a, b_m, m_b, m_y), 2-output (y_m, m_y) masked AND as shown below:

$$y_m = a_m.b_m \oplus m_a.b_m \oplus [m_b.a_m \oplus (m_a.m_b \oplus m_y)] \tag{1.6}$$

$$a_m = a \oplus m_a, b_m = b \oplus m_b, y_m = y \oplus m_y \tag{1.7}$$

The m_a, m_b and m_y are random 1-bit masks that can be equal or different.

Similarly, any basic gates can be converted into masked versions and hence, any circuit can be built using masked gates. When compared with algorithmic masking, at gate level, masking does not utilize very high-level functions by logic gates. In a cryptographic circuit consisting of regular gates, one can swap them for masked gates for form a version with lessor loopholes. However, gate counts will increase and thereby will consume extra power. Standard tools

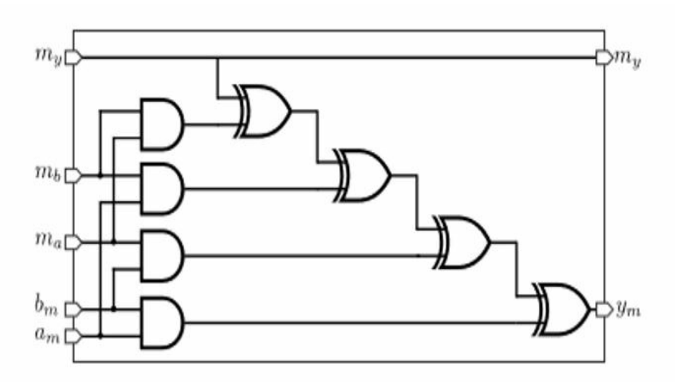


Fig. 1.6. Hardware design for using masked logic gates [1]

that synthesis logic does not support such a design flow. Their main goal is to optimize timing and area. Hence, more effort is required for functioning and timing verification of the masked gate netlist.

Disregarding the real hardware behavior changes, masking is a complete mathematically provable countermeasure and ends up securing the linkages between secret data and the final values of outputs. Attacks like the Hamming weight and Hamming distance are countered since these attacks utilizes only the final values.

On the other hand, a real design of hardware is complex. Input signals arrive to logic gates at different timings, hence the design overlooks the glitches in the system. In the figure above, the design contains imbalanced signal paths and variations may exist already before the signals enter the first masked layer level, thereby, some glitches end up appearing at the final stage before final value settles and end up correlating with the secret key values which are supposed to be hidden. Masked gates are hence vulnerable to attacks against those that use more precise toggle count power model.

Simple masking may not be enough to sustain attacks, however, combined with other counter measures, it increases the difficulty level.

1.6.2 Masking algorithm

When we attempting to hide the relations between processed values and secret values, masking as we have seen, ends up converting the processed values itself. On software, it is easier to implement masking rather than hiding. Moreover, if new attacks of side-channel attacks occur, rather than changing the hardware, masking scheme implementation can simply be changed. Main goal stays to combine intermediate values with masks to change the characteristic power traces. Even though, the value may leak, but it is not the real value but the masked value, which if carefully imbedded, the attacker cannot analyze the measured traces easily.

$$v_m = v * m \tag{1.8}$$

In equation 1.8, m stands for the mask generated on device and differs with each execution of the algorithm. m should be truly random and should not be such that it can be predicted. The operation "*" is an arbitrary one, while its inverse should be known to obtain the original intermediate value at the end, failing which, we would have to figure out a compensation mask along with the algorithm execution. We must adjust the algorithm execution with respect to each type of operation.

When we apply a mask on a device, we must keep in mind that the operations that our processes which transforms the values are linear:

$$f(x) + f(y) = f(x + y) \tag{1.9}$$

$$(v_1 * m) + (v_2 * m) = (v_1 + v_2) * m \tag{1.10}$$

If we do not keep the masks linear, they will disrupt our calculation, making it difficult to achieve the desired outcome after unmasking. While using more masks, the algorithms mix them at some stage. Consequently, we would either keep in mind that the mixing should not remove the masks, or we select the masks in such a way that interfering becomes impossible. We are going to talk about the masked algorithms in further detail in 2.2.

1.6.3 Dual Rail Precharge (DRP) Logic

Another method used to convert the result is that instead of integrating a mask at input level, we design DRP logic cells that make output switching activities independent of inputs and hence attaining constant power consumption ranges at gate level. In dual-rail gate, logic 0 and 1 are shown by complementary pairs such as (0,1) or (1,0). (1,1) is not a valid state of a circuit and before the charge, the state is defined as (0,0). The purpose is that the counts of transitions of 0 to 1 and 1 to 0 appearing at each gate per clock cycle remain the same, and not disturbed by the input values. Cross coupled inverters are also called Sense amplifiers, hence transistor level DRP designs are also known as Sense amplifier-based logic (SABL).

They hold the following advantages:

- Output changing processes are independent of input values.
- Glitches in the system do not affect the output changes.
- Balanced internal capacitance.

On the other hand, SABL based designs must balance external loads on routing wires. To relax this constraint, we have Three Phase Dual Rail Pre-Charge Logic (TDPL), which involves another discharge phase after evaluation. Hence, in TDPL, the differential outputs discharge once and charges once. Consequently, the unbalanced external loads have a lower effect.

Even though, SABL designs do achieve constant power usage while simulating the circuit, it still requires an expensive fully customized IC design flow, especially due to its dynamic logic. Hence, the later developments of DRP cells utilize static CMOS standard cells to enhance the switching activities of SABL cells. We also have the Wave Dynamic Differential Logic (WDDL) which is compatible with standard cell based semi-custom ASIC design. Due to the positive monotonic gates, pre-charged signals do not necessarily have to be distributed around the world to every combinational gate. In the initial stage, the inputs of any compound gate are pre-charged to 0s and any AND or OR gate output logic should be 0 too. Consequently, the AND gates at the output stage of a SDDL gate are removed.

Even though, WDDL designs are imbedded through existing standard cells, however, normal IC tools do not always support it. Assuming a cryptographic circuit is designed using hardware languages like Verilog, the designer initially narrows the cells to AND, OR and INV. Once we obtain a synthesized cell netlist, a custom script is executed. It replaces every gate by its WDDL alternative and hence removes the invertors, by exchanging the outputs. In the physical layout, strict designing barriers are applied. Those barriers that are used to rout the differential wires however, should be on the same metal layers, on adjacent routing paths, must have same lengths and breadths, must be protected with V_{dd} or ground lines to eradicate crosstalk, forming every other metallic layer into a ground plane to regulate the inter layer capacitance. Hence, the designers should enforce commercial IC design tools in several ways to meet the physical design barriers.

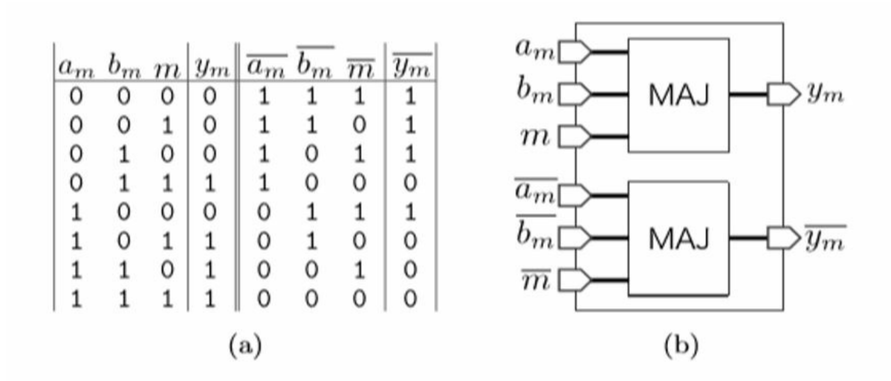
Theoretically, power consumption of SABL or WDDL gates are independent of the input data values and hence thwart the power analysis attacks, whereas practically, in the IC designs, the capacitances of the complementary outputs are impossible to be perfectly in balance because of the complexities in logic building, placing and routing. While the most powerful IC designing tools may make a desired balanced output when even the chips are being made-up, there still exist manufacturing variances that hence disturb the balance. Evidently, there are loopholes and leakages and when given sufficient magnitudes of power, it traces the secrets that are supposed to be extracted.

1.6.4 Masked Dual Rail Precharge Logic

Masked Dual-Rail Pre-charge Logic (MDPL), as the name suggests, combines masking and DRP method to stop DPA attacks. The MDPL has dual-rail design to access data-independent output switching activity and means to use a random mask bit to recompense for the loss imbalances in routing. An MDPL AND gate has six dual-rail inputs $(a_m, \overline{(a_m)}, b_m, \overline{(b_m)}, m, \overline{m})$ and gives two outputs $(y_{(m)}, \overline{(y_{(m)})})$. The truth table is in the given figure and the outputs are hence found as

$$y_m = ((a_m \oplus m).(b_m \oplus m)) \oplus m \tag{1.11}$$

$$\overline{y_m} = (((\overline{(a_m)} \oplus \overline{(m)}).(\overline{(b_m)}(\oplus)\overline{(m)})) \oplus \overline{(m)} \tag{1.12}$$



a_m	b_m	m	y_m	$\overline{a_m}$	$\overline{b_m}$	$\overline{m}$	$\overline{y_m}$
0	0	0	0	1	1	1	1
0	0	1	0	1	1	0	1
0	1	0	0	1	0	1	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	1	1	0	1	0	0
1	1	0	1	0	0	1	0
1	1	1	1	0	0	0	0

Fig. 1.7. Truth table with its hardware design [1]

From the truth table, we can observe that y_m and $\overline{(y_m)}$ may be obtained through the majority function (MAJ). If the inputs contain more 1s than the number of 0s, the result will yield an output of 1. However, if the inputs contain more 0s than 1s, then the output will yield an output of 0. A gate that proceeds with the majority function is meant to be found in a standard cell library. In an MDPL, all the signals are precharged to 0 and just like the wave dynamic differential logic, the precharge waves launch from registers and sweep all combinational logic. The majority gate is a positive monotonic function, therefore, the MDPL gates are protected from all the glitches. Moreover, since both the dual-rail outputs can be obtained using an identical majority function, MDPL has balanced current paths internally.

Like RSL, MDPL also utilizes a single-bit mask m (and its complementary $\overline{m}$) for every data byte encryption. MDPL, however, does not require balancing the loads and routing constraints. The imbalance is by all accounts rewarded by the random masks. The authors of this model imitated their project by utilizing a $0.35\mu m$ standard cell library, and then concluded, by using the strategy of mounting DoM assaults on the gate-level netlist power simulation that the secret key was not yet exposed. While comparing this with the regular but leaky standard cell design, the MDPL version is 4.5x larger in the area while having half the operating speed.

Even though SABL, WDDL, RSL and MDPL are all intended to be glitch-free, their execution still fails to completely prevent different arrival periods of signals that depend on cell and route suspensions. A lot of the usual Boolean logic gates contain the property which their logical outputs may be determined in a unique manner without essentially getting to know all of the inputs. Consequently, a gate often calculates its logical output earlier, before waiting for all of its logical inputs. Due to this changed time of calculation, which is called early propagation (or early evaluation), data-dependent power dynamics are exploited. It is explained by Suzuki and Saeki that the leakage using WDDL and MDPL as examples. They showed that a cell which evades early propagation has to postpone the calculation moment until all of the input signals have arrived, for e.g., all the inputs of MDPL have established to differential values. It is confirmed that the leakage on a model MDPL chip and proposed improved MDPL (iMDPL) to respond to early propagation. The proposed antidote in iMDPL remarkably surged the area compared to MDPL. For instance, one more NAND, three more OR and six more NOR gates were joined to the MDPL AND gate in the above figure 1.7.

It was discovered that another origin of outflow in RSL and MDPL that is the mask bit itself. For example, in RSL, the process of encryption with m = 0 consumed higher (or lower) power as compared to the encrypting process withm = 1. This is anticipated as the mask bit places the circuit in one of the two harmonizing forms. To find the time when the two distributions evidently distinguish from one another deserves some effort related to searching. Tiri and Schaumont recommend folding one portion around the nick on top of the other and then execute a standard DoM based attack. Thus, research works done later usually point toward this method as the folding attack. The authors used toggle count simulation and demonstrated the concept. Now the Gaussian curve shows us a probability density function, to use it to then filter the mask bit is called probability density function filtering. In case of MDPL, an alike approach is appropriate too. It was further found that leakages in iMDPL using the folding attack and a few other advanced mask detections techniques. Therefore, once the mask effect vanishes from RSL or MDPL countermeasures, the circuit demotes to simple and standard precharge or dual-rail logic without balanced load capacitance and is then found susceptible to attacks.

It is widely understood that one single countermeasure cannot be effective to secure the hardware that is meant for security against several side-channel attacks. Hence, a simple thought is to then merge different types of countermeasures. Sadly, some countermeasures are not appropriate. Therefore, to choose a mixture of appropriate countermeasures is rather a challenging and uphill task and requires a lot of work as practice.

Of course, by using multiple countermeasures simultaneously may further increase the circuit area, power, cost and makes the design flow very complex, however, to improve the level of security is never free and never optimal. There is always room for improvement. The cost and effort for the security measures should also be near to the value of the device itself. For a selected combination of countermeasures, it must also be used with equivalent key updating policies to make permissible power trace queries of the same key within its tolerance.

1.6.5 Hiding

Hiding countermeasures try to break the link between the processed value or executed operation and the power consumption. There are several ways how to do that and most of those countermeasures are applied in hardware. Hiding countermeasures do not alter the processed values and they are insensitive to the algorithm that's executed on the device. There are two methods on how to hide the dependency in power consumption - by randomizing the consumption or making it constant for all operations and all values. Both methods will stop the attacker from obtaining 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 separated in two categories according to the dimension they use - time or amplitude.

1.6.6 Randomization

Another method to counter side channel attacks is to randomize data that might leak through various side channels, such as power consumption, electromagnetic radiation, or execution time. The problem is to assure that an attacker may gather random information only, and therefore cannot gain any useful knowledge about the real initial and/or intermediate data involved in computations. In case of elliptic curve cryptosystem, randomized projective coordinates method is a practical countermeasure against side channel attacks in which an attacker cannot predict the appearance of a specific value because the coordinates have been randomized. The proposed scheme principally intends to resist the simple power analysis, not the differential power analysis. The standard DPA utilizes the correlation function that can differentiate whether a specific bit is related to the observed calculation. To resist DPA, we need to randomize the parameters of elliptic curves. However, if these randomization methods are simultaneously used, no attack is known to break the combined scheme. In other words, SPA-resistant schemes can be easily converted to be DPA-resistant ones using these randomization.

The mask can be generated randomly after which we should ensure that the generator we used to acquire the random numbers is not predictable. If the device has any peripheries or the 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 approaches 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 process must fit the rule: $Sbox(v \oplus m) = Sbox(v) \oplus m$. S-box conversion itself is very simple, but all 256 values have to be read, altered and written each time it is changed - 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 the introduction of fixed masks and low entropy masking schemes (LEMS) with precomputed converted S-boxes stored in memory.

1.6.7 Blinding

Blinding is originally a notion in cryptography that allows a user to have a provider compute a mathematical function y = f(x), where the user provides an input x and retrieves the corresponding output y, but the provider would neither learn x nor y. This concept is beneficial if the user cannot compute the mathematical function f all by himself because the provider uses an additional private input to calculate f efficiently. It is based on a homomorphic property of the RSA signing function. Blinding techniques are also the most effective countermeasure against remote timing analysis of web servers and against power analysis and/or timing analysis of hardware security modules.

The masking method used in Blinding is for asymmetric cryptography. 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 me by multiplication where m is the random mask and e is the public key. Exponent blinding takes benefit of adding the mask $m = m.\phi(n)$ that does not alter the output because $vd + m.\phi(n) \equiv vd(\bmod(n))$. Using the masking like this is possible because of the arithmetic nature of the algorithm.

1.6.8 Secret sharing

The intermediate value v is separated into a number of shares and only when the attacker categorizes all the shares he can get the secret value itself. Secret sharing with two shares (m, vm) is accomplished by applying a mask on the intermediate value $vm = v \oplus m$. Dividing the intermediate value into additional shares increases the security [7].

1.6.9 Fixed memory usage

The encryption algorithms should essentially balance the security and performance. Randomly generated masks reduce the performance with repeated conversions of all the non-linear operations. Precomputing all probable Sboxes 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 choice because of memory constraints. The memory and time constraints caused the researchers to look for other feasible 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 [8].

1.7 ATMega163 Smart Card

The ATmega163 is a low-power CMOS 8-bit microcontroller created on the AVR architecture described in [9]. By executing robust instructions in a single clock cycle, the ATmega163 achieves outputs approaching 1 MIPS per MHz allowing the system designer to enhance power consumption versus processing speed.

The AVR core combines the instruction set with 32 general purpose working registers. All the 32 registers are directly linked to the Arithmetic Logic Unit (ALU), allowing two independent registers to be read in one single instruction completed in one clock cycle. The subsequent architecture is more code efficient while realizing throughputs up to ten times quicker than predictable CISC microcontrollers.

The ATmega163 offers the following characteristics: 16K bytes of In-System Self-Programmable Flash, 512 bytes EEPROM, 1024 bytes SRAM, 32 general purpose I/O lines, 32 general purpose working registers, three flexible Timer-/Counters with compare modes, internal and external interrupts, a byte oriented Two-wire Serial Interface, an 8-channel, 10-bit ADC, a programmable Watchdog Timer with internal Oscillator, a programmable serial UART, an SPI serial port, and four software selectable power saving modes. The Idle mode halts the CPU while permitting the SRAM, Timer/Counters, SPI port, and interrupt system to remain operative. The Power-down mode saves the register contents but restricts the Oscillator, disabling all other chip functions until the next interrupt or Hardware Reset. In Power-save mode, the asynchronous Timer Oscillator remains running, permitting the user to uphold a timer base while the rest of the device is sleeping. The ADC Noise Reduction mode breaks the CPU and all I/O modules except asynchronous timer and ADC, to diminish switching noise through ADC conversions. The On-chip ISP Flash can be programmed over a SPI serial interface or a conventional programmer. By installing a Self-Programming Boot Loader, the microcontroller can be reorganized within the application without any external components. The Boot Program can use any interface to download the application program in the Application Flash memory. By combining an 8-bit CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega163 is a prevailing microcontroller that delivers a highly flexible and cost-effective result to many embedded control applications. The ATmega163 AVR is supported with a full suite of program and system development tools including: C compilers, macro assemblers, program debugger/simulators, In-Circuit Emulators, and evaluation kits.

1.7.1 Architecture

The fast-access Register File idea contains 32 x 8-bit general purpose occupied registers through a single clock cycle access time. This means that in one single clock cycle, one Arithmetic Logic Unit (ALU) process is performed. Two operands are output from the Register File, the operation is completed, and the result is kept back in the Register File, all in a single clock cycle.

Six of the 32 registers can be expended as three 16-bits indirect address register pointers for Data Space addressing - allowing effective address calculations. One of the three address pointer is also used as the address pointer for look-up tables in Flash Program memory. These additional function registers are the 16-bits X-, Y-, and Z-register. The ALU supports arithmetic and logic operations amongst registers or among a constant and a register. Similarly, single register operations are performed in the ALU.

Conventional memory addressing modes can be used on the Register File as well. This is permitted by the fact that the Register File is allocated the 32 lowest Data Space addresses ($00-1F$), allowing them to be accessed as though they were normal memory locations. The I/O Memory space comprises of 64 addresses for CPU peripheral functions as Control Registers, Timer/Counters, A/D-converters, and other I/O functions. The I/O Memory can be accessed directly, or as the Data Space locations following those of the Register File, $20-5F$.

The AVR makes the use of a Harvard architecture concept - with separate memories and buses for program and data. The Program memory is implemented with a two-stage pipeline. While one instruction is being performed, the next instruction is pre-fetched from the Program memory. This concept enables instructions to be executed in every clock cycle. The Program memory is In-System Re-Programmable Flash memory. With the jump and call instructions, the whole 8K word address space is directly accessed. Most AVR instructions have a single 16-bit word format. Every program memory address contains a 16- or 32-bit instruction. Program Flash memory space is divided in two sections, the Boot Program section (256 to 2,048 bytes) and the Application Program section. Both sections have dedicated Lock bits for write and

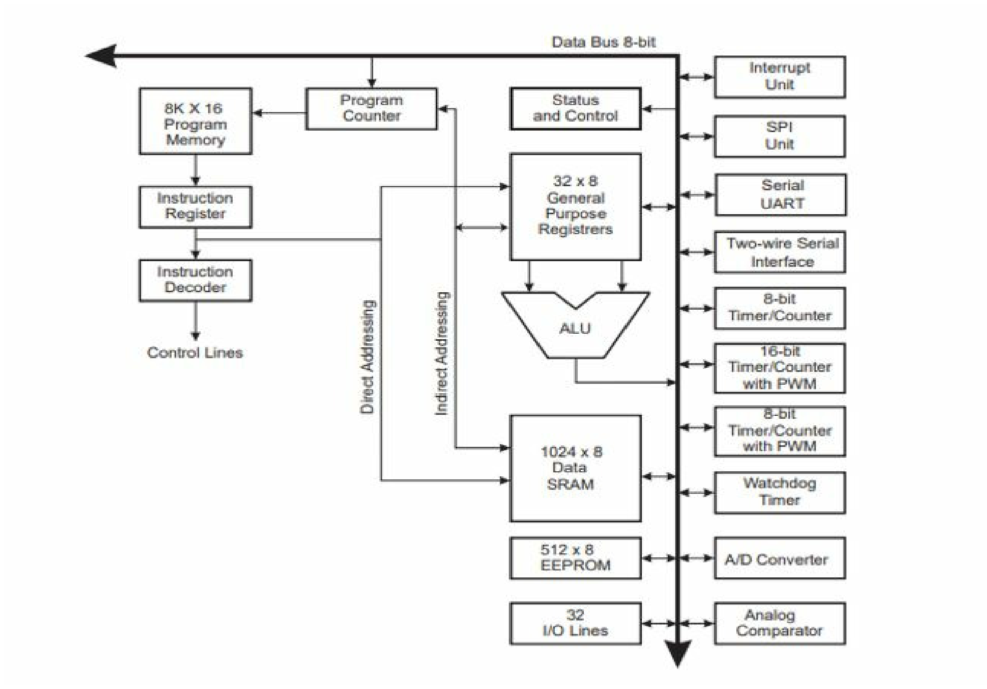


Fig. 1.8. ATMega163 RISC Architecture

read/write protection. The SPM instruction that writes into the Application Flash memory section is allowed only in the Boot Program section. During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the Stack. The Stack is effectively allocated in the general data SRAM, and consequently the Stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine (before subroutines or interrupts are executed). The 11-bit Stack Pointer SP is read/write accessible in the I/O space.

The 1,024 bytes data SRAM can be easily accessed through the five different addressing modes supported in the AVR architecture. The memory spaces in the AVR architecture are all linear and regular memory maps. A flexible interrupt module has its Control Registers in the I/O space with an additional Global Interrupt Enable bit in the Status Register. All interrupts have a separate Interrupt Vector in the Interrupt Vector table at the beginning of the Program memory. The interrupts have priority in accordance with their Interrupt Vector position. The lower the Interrupt Vector address, the higher the priority.

1.7.2 Flash program memory

The programmable flash memory is where the program instructions are stored. It's significantly bigger and it's non-volatile. If we need to store any big data structures such as S-box look-up tables, we must store them in Flash Program memory though storing the data in program memory will lead to the consumption of an extra clock cycle for all the read and write operations.

The ATmega163 comprises of 16K bytes On-chip In-System Self-Programmable Flash memory for program storage. Since all instructions are 16- or 32-bit words, the Flash is organized as 8K x 16. The Flash Program memory space is separated in two sections, Boot Program section and Application Program section. The Flash memory has a strength of at least 1,000 write/erase cycles. The ATmega163 Program Counter (PC) is 13 bits wide, therefore addressing the 8,192 Program Memory locations.

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

The two sections can be distinguished by their ability to execute Store Program Memory(SPM) instruction as it can rewrite any address in the Program memory but can only be accomplished 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 altered page by page (128 bytes). This leads us to the conclusion 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.

Practically, we need to precompute any converted masked S-box look-up tables before placing them into the program memory. We also need to 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 implementation of the algorithm. However, 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 created 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 surpass even the Flash memory limit of ATMega163. However, if we only use a fixed subset of all possible masks, we can fit their look-up tables in the Flash memory.

1.7.3 EEPROM

The usage of Additional Electronically Erasable Read-Only Memory is to identify device identification data that does not change frequently. It would take too much time to store or load any data from EEPROM during algorithm execution-unlike SRAM or Flash memory EEPROM can only be accessed via data bus. It can be rewritten by certain program instructions, despite being read only, though its latency too high and the data that can be stored is very limited.

1.7.4 Instruction set

A single clock cycle is enough to execute most of the instructions set described completely in which contain 130 instructions. Before executing a single instruction, the next one already 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 can only be performed on registers, on immediate value and never on register. Total Execution Time of one ALU operation consists of Register Operands Fetch, ALU Operation Execute and Result Write Back phases.

Such operation power consumption can be measured, so we can change both the consumption of the ALU Operation Execute phase and the register value. Thus, we realize using C language to write the software countermeasures in not sufficient, so we make use of AVR assembly language to implement the critical parts.

1.7.4.1 Data transfer instructions

The data space takes 2 clock cycles to load and store instructions and it takes 3 clock cycles to load program memory with LPM instructions. An extra cycle is required to load any data from the program memory and even storing is not used in practice. We made the use of program memory for the original static s-box look-up table and Rijndael field multiplication look-up tables in our implementation. The instructions to data load and store can use either direct or indirect addressing.

The Hamming weight of the value will be exposed if loading value from SRAM or flash memory and before the load operations the destination register Rd was cleared. Before we load the secret value, we can precharge the register with random value to prevent the leakage.

Many Logical operations are not used by AES encryption algorithm with look-up tables. EOR is the most frequently used instruction on ATMega163 platform. Execution of an exclusive by EOR instruction OR on registers Rd, Rr and store the result into Rd [3]:

$$Rd \leftarrow Rd \oplus Rr \tag{1.13}$$

The EOR instructions can be exploited in multiply ways. The Hamming distance of the two operands is the easiest which leaks during the instruction execution.

$$HD(Rd, Rr) = HW(Rd \oplus Rr) \tag{1.14}$$

If only one of the operands is known to the attacker, the leakage can be prevented. It is then not possible for him to use plaintext and key combination in his power consumption model. Masking the plaintext value or key value with random masks, the visibility of the hamming weight of their combination is nonexistent in the traces and the power consumption of their exclusive OR is also masked.

The EOR instruction can be exploited in another way by targeting the destination register. The number of 1s that are flipped during the write back phase determine the power consumption. Rr is the difference between old and new value of register Rd which can be used in DPA if we used the operation with the secret key loaded into Rr:

$$Rd \leftarrow 10010101 \text{ plaintext } 0x95 \tag{1.15}$$

$$Rr \leftarrow 10101010 \text{ key } 0xAA \tag{1.16}$$

$$Rd \leftarrow 10010101 \oplus 10101010 \tag{1.17}$$

$$HD(Rd, Rd \oplus Rr) = HW(Rr) \tag{1.18}$$

Chapter 2

Design and Implementation

This chapter introduces the objectives of this project, the masked algorithm that was implemented and the countermeasures used.

2.1 Objectives

Our goal is to use an existing algorithm of AES and mask it to make it more secure. Two different types of masking schemes were adopted which have been described in further detail in the upcoming sections.

The first part of this project was to use an unmasked AES implementation and try to break it using single-order DPA. Moving forward, we used a masked implementation but the mask was a single pseudo-random number which was loaded after one round of dummy encryption. We tried to use the single-order and double-order DPA on this implementation. After getting the results from this part, we moved over to the six masks masking scheme and attempted the single-order and double-order DPA on that as well. The results were recorded and are displayed in chapter 3.

2.2 Masked AES implementation

We implemented two types of masking schemes. One of them was a single pseudo-random mask and the other contained six random masks implemented after various operations during the encryption.

2.2.1 Boolean masking

The Boolean operation used in masking scheme can be implemented by exclusive OR-ing a byte of mask to byte of key or plaintext. It is very straightforward for linear operations in AES, the non-linear S-box substitution done by SubBytes on the other hand converted masked s-box look-up must be computed.

2.2.1.1 Single mask

The masking of power consumption of *AddRoundKey*(0) and *SubBytes* in the first attempt was made with one pseudo randomly generated mask. This mask was applied at the end of the first round in order to separate the inconsistent run time of the function *rand* from the principle of masking. Due to this, the first encryption round is not encrypted with the mask and it acts like a dummy encryption.

This way was successful in masking the Hamming weight of the output of *AddRoundKey*(0) → *SubBytes*() result. After the measured power consumption and correlating the traces with the consumption model based on Hamming weight, we were unable to retrieve the 16 bytes of key. The single mask scheme was successful on the Hamming weight model, but was not able to mask the Hamming distance of the value before SubBytes() and after because SubBytes() performs substitution:

$$p \oplus k \oplus m \rightarrow Sbox(p \oplus k) \oplus m \tag{2.1}$$

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

$$(p \oplus k) \oplus (SBox(p \oplus k)) \tag{2.2}$$

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

$$(p \oplus k \oplus m) \oplus (SBox(p \oplus k) \oplus m) \tag{2.3}$$

2.2.1.2 Multiple masks

This scheme makes the use of six independent masks. The masks are distributed amongst several operations of each round. The first two masks are the input and output masks for the masked SubBytes operation. The remaining masks are the inputs of the MixColumns operations.

We made the use of six independent masks in our algorithm. The masked SubBytes operation used two masks, m and $m\prime$ as its input and output. The other four masks are the input masks for the MixColumns operation.

Two precomputations take place at the beginning of the encryption. Firstly, a masked S-box table is computed. Secondly, output masks are generated for the MixColumns operation. The output masks of the MixColumns operation are denoted as $m\prime_1, m\prime_2, m\prime_3, m\prime_4$.

The plaintext is masked with $m\prime_1, m\prime_2, m\prime_3, m\prime_4$ at the beginning of each round. Thereafter, the AddRoundKey operation takes place. The masking at the round key operation is performed such that the masks $m\prime_1, m\prime_2, m\prime_3, m\prime_4$ are changed to m. Then the table look-up is performed from the S-box table. This further change the masks to $m\prime$. ShiftRows, as mentioned earlier, has no impact or influence on the masks since all of the bytes of the state are masked with $m\prime$ already. Before going into the operation for MixColumns, we change the mask from $m\prime$ to m_1 in the first row, m_2 in the second row, m_3 in the third row and m_4 in the fourth. Several rounds are masked in this way. The masks are removed by the final AddRoundKey operation at the end of the last round [2].

As mentioned above, I was unable to place these masks in the later parts of the code but were instead placed at the beginning of each round. We wanted to isolate hiding and not being able to do so caused issues later when trying to apply higher-order DPA. Due to time constraints, we decided to leave the updates needed to make this implementation better as future work.

A graphical representation of this algorithm is shown in figure 2.1.

2.2.2 Random register precharging

This countermeasure uses the worst possible value - $0x00$ for the registers that we are working with. There is a possibility that the registers from the previous routines are cleared and loading any secret value into the cleared register is equal to the leaking of the hamming distance of the value $0x00$ which is the Hamming weight of the value itself.

The hamming weight can help the attacker to narrow down the list of possible candidates but itself does not give away the contained value. If the exposition is to be prevented of the loaded value, the destination register is precharged with the random value that can be overwritten later. For each algorithm execution the leaked hamming distance is different.

We can apply a similar countermeasure to the SRAM store instruction, but the algorithm would be slowed even more, and the change of the values stored in the SRAM did not prove to be as clear as the change in the register stored values.

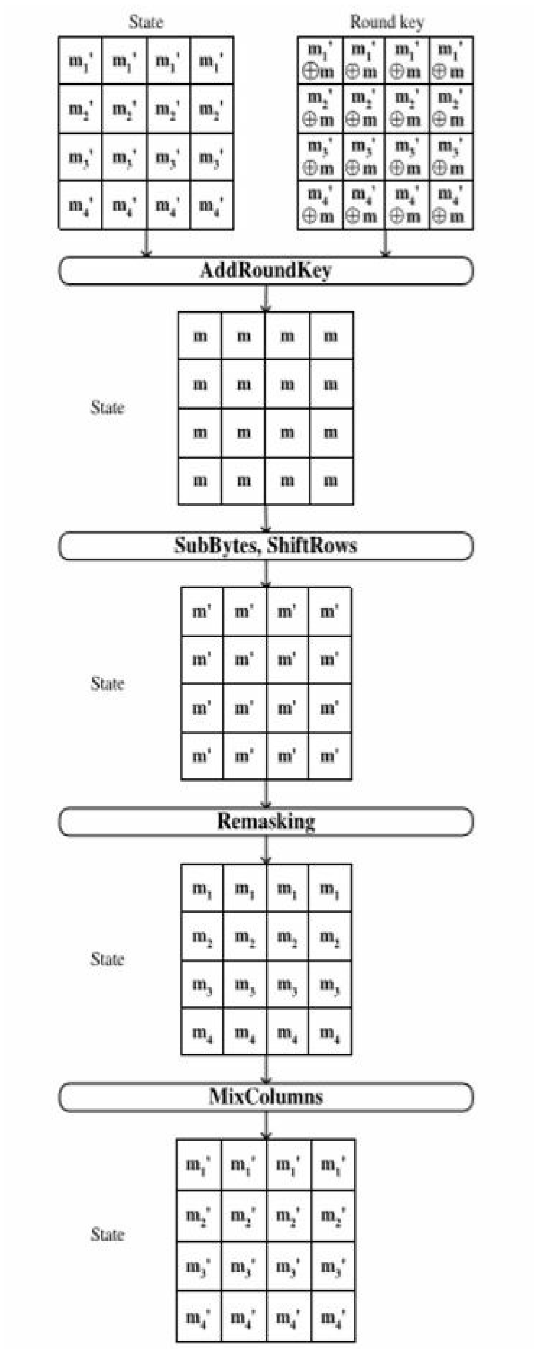


Fig. 2.1. The AES round functions change the mask of the AES state byte [2]

2.3 Unused masking countermeasures

The masking countermeasures for operations ShiftRows and MixColumns are also present. These operations are much harder to analyze because they do not work with a single byte but with combinations of 2 to 4 bytes.

The countermeasures specified in this section were not implemented due to their expected inefficiency.

2.3.1 ShiftRows

The leakage of the operation depends on the implementation of the shifts. In ShiftRows, each byte is rotated one byte towards the left. There is no shift in the first row; the values are rotated one byte left in second row as shown below.

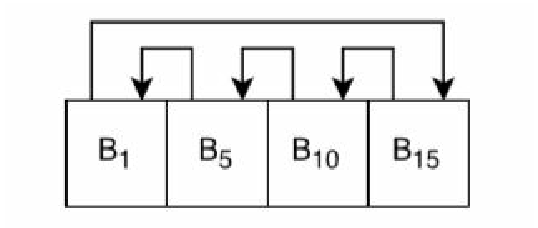


Fig. 2.2. ATMega163 RISC Architecture

The value stored originally on B1 position is defined as:

$$B_1 = Sbox(K_1 \oplus P_1) \tag{2.4}$$

With applied mask it's:

$$B_1 = Sbox(K_1 \oplus P_1) \oplus m \tag{2.5}$$

The neighbor shifted on B1 position is defined with applied mask as:

$$B_5 = Sbox(K_5 \oplus P_5) \oplus m \tag{2.6}$$

When the value on position B1 changes to B5, the leaked Hamming distance will be the Hamming weight of their difference:

$$HD(B_1, B_5) = HW(Sbox(K_1 \oplus P_1) \oplus m \oplus Sbox(K_5 \oplus P_5) \oplus m) \tag{2.7}$$

The mask on B_1 will interfere with the mask on B_5. If we wanted to model the power consumption, the hypothetical consumption for the combination of B_1 and B_5 would need to be created, that's $2^{16} = 65536$ possible combinations and doing that would mean that the two keys could be guessed at once. Compared to $2 * 28$ cost of acquiring two bytes with DPA on operations that only work with one byte, it's more difficult.

We should implement the rotation for the third and fourth row as direct displacement of two values and refrain from consecutive shifts of two and three. The number of leaking values will be lowered, and execution will be sped up. If the ShiftRows operation is masked, we would need four distinct random masks for the four values each row m_0, m_1, m_2, m_3. In this operation, no mixing of row happens. Hence, the same set of 4 masks on all three rows can be used. The first of these rows does not need to be masked, because the shift has not happened. The difference between B_1 and B_5 with new mask would be:

$$HD(B_1, B_5) = HW(Sbox(K_1 \oplus P_1) \oplus m_0 \oplus Sbox(K_5 \oplus P_5) \oplus m_1) \tag{2.8}$$

And the mask that will be intact are m0 and m1. With each execution, the hamming distance of every pair of masks changes to keep the consumption random.

The operations that our attack targets are those that only use one byte of the key. Therefore, this countermeasure was not used in practice.

2.3.2 MixColumns

MixColumns is the last operation left to analyze and protect. Values from each column are mixed in this operation. After ShiftRows, the byte C_1 is defined as:

$$C_1 = Sbox(K_5 \oplus P_5) \oplus m \tag{2.9}$$

and with MixColumns it changes to:

$$C_1 = 01 * B_0 + 02 * B_5 + 03 * B_{10} + 01 * B_{15} + m \tag{2.10}$$

To shorten the definition, we define $Sbox_i = Sbox(K_i \oplus P_i)$, where i is the position of byte in the block.

$$C_1 = 01*(Sbox_0 \oplus m) + 02*(Sbox_5 \oplus m) + 03*(Sbox_{10} \oplus m) + 01*(Sbox_{15} \oplus m) \tag{2.11}$$

First, the operation MixColumns is linear which is shown and mask from the output can be easily removed:

$$C_1 = Sbox_0 + 02*Sbox_5 + 03*Sbox_{10} + Sbox_{15} + m + (02+03)*m + m \tag{2.12}$$

$$C_1 = Sbox_0 + 02 * Sbox_5 + 03 * Sbox_{10} + Sbox_{15} + 01 * m \tag{2.13}$$

The original C_1 and the new ones difference can be defined as:

$$HD(C_1, C_1^{'}) = HW(Sbox_5 \oplus m \oplus Sbox_0 \oplus 02*Sbox_5 \oplus 03*Sbox_{10} \oplus Sbox_{15} \oplus m) \tag{2.14}$$

The power consumption model that can be created based on hamming distance of value on specific position would require 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 can be attacked separately because it is divided into multiple multiplication and addition operations to avoid the combination of multiple bytes.

The need for similar masking scheme as ShiftRows is required to mask this operation, the application will be on columns. Each column would have 4 distinct masks with random Hamming distance of all pairs for each execution.

2.3.3 Dummy cycles

The insertion of dummy cycles was considered one of a countermeasure briefly but was soon abandoned. This is a hiding countermeasure that was described before, and it has its few downsides. One of the downsides of dummy cycle is that it adds empty operation in algorithm executions that do not contribute to it, and secondly with some pre-processing it can be filtered from the traces. We could fix the misaligned traces with pre-processing, due to the reason that there will be parts that occur in all traces randomly interleaved with traces of dummy cycle.

For every execution the number of inserted dummy cycles and their total duration must be same. If this measure is not considered there will be an opportunity for timing attack because of different lengths. Should the operation in the dummy cycle stay the same, it will be easily being detected in the traces because of the repeating pattern.

Insertion of the dummy cycles as software countermeasures was not used due to the reason that it downgrades the performance significantly and by pre-processing the traces it can be invalidated.

2.4 Used programming languages

Two languages are used by the Implementation - the main language of the project is C and the AVR assembly language is used to write the few important functions. 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.

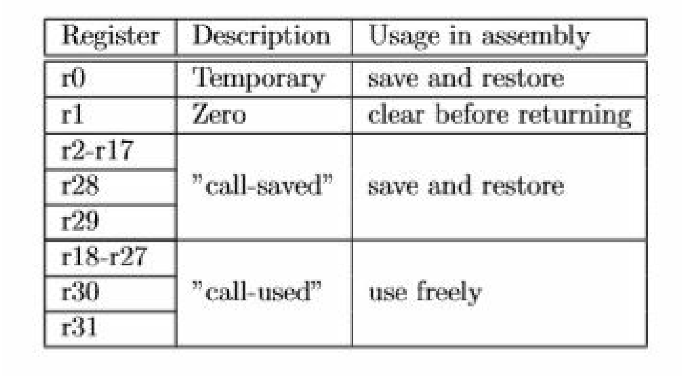
2.4.1 C and assembly mixing

With avr-gcc, we can either use the inline assembler functionality in C or combine C and assembly source codes. To write the whole functions in assembly we need to combine C and assembly source codes which helps us to avoid writing chunks of codes inside existing C functions. In AVRGCC project the C and assembly codes are combined which are described in Atmel Application Note.

The implementation makes the use of calling assembly routines, sharing global variables and passing variables to assembly. The two files with the extensions .c and .S are the C and assembly code respectively.

2.4.1.1 Using registers

The certain usage of registers dictated by the convention of avr-gcc must be complied with, to prevent the unpredictability and incorrectness of the behavior of code generated by avr-gcc from C. the table 2.3 depicts the complete summary of register usage. To make use of "call-saved" registers in our assembly routine, these registers must be pushed onto the stack and then popped back before returning form the routine. This should be accomplished before we start working with them. Avr-gcc expects that the "call-saved" registers and r0 stay intact and r1 contains the value 0x00. All other registers are available freely to assembly code, but their contents are 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		

Fig. 2.3. Register usage in assembly with avr-gcc [3]

As we can see from table 2.3, assembly code can use two of three extended registers freely: X-pointer and Z-pointer.

2.4.1.2 Sharing global variables

From C, only the global variables can be made visible to assembly, since the rest of them are defined as the 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 r_{25} accessing the address that was passed indirectly via extended Y-pointer:

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

2.4.1.3 Calling assembly routines from C

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

```
extern void subBytes ( void );  // in .c
```

```
.global subBytes
 subBytes:
;routine
 Ret    //in .S
```

Arguments are not needed by the function in this example. Thus, we are not obliged to acquire them in assembly. If it made any exceptions in terms of arguments, they would be passed in accordance to avr-gcc convention.

Arguments in fixed argument list are passed in registers $r_8 - r_{25}$. In Atmel manual and Atmel Libc reference manual, this range differs. Even numbers of registers are consumed by each argument. The unused byte in case of one-byte values is not touched. It is passed on the stack if the argument does not fit into available registers. 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 2.4.

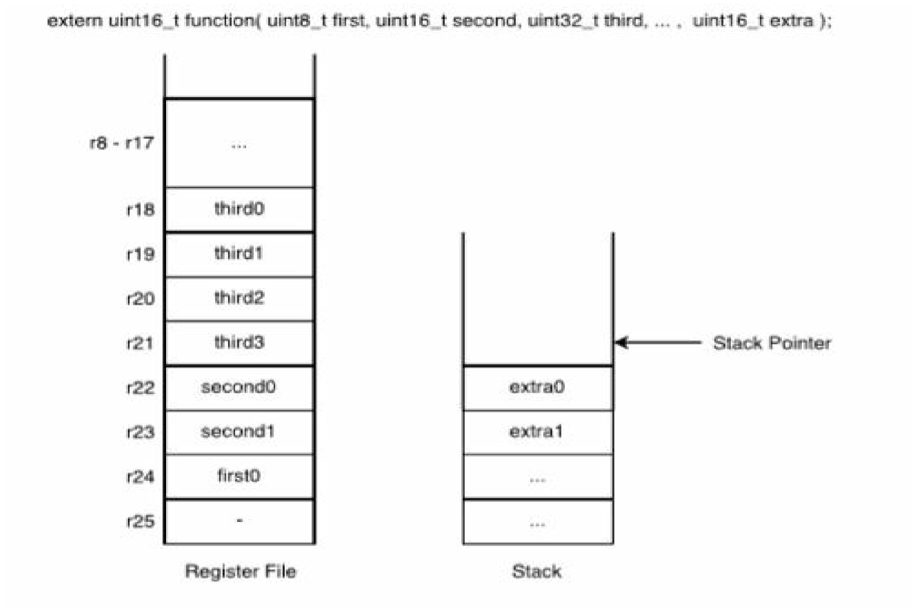


Fig. 2.4. Parameters mapping

The return value must be stored in the same manner - starting from r_{25} and maintaining the same byte order as passed arguments. The source of leakage generated by the compiler will be passing the real key value as function argument, therefore we declare the variable as global and only use associated pointer.

2.4.2 Avr-gcc compilation and optimization issues

The project in C is compiled to AVR platform executable code using avr-gcc compiler. The level of optimization is set as the smallest and fastest code: -0s. This is great for the efficiency and size of the executable but could in turn affect the designed countermeasures unintentionally.

2.4.2.1 Operation EOR

The optimization cannot negatively affect instruction EOR, but a possible leakage which is described in 1.7.4.1 is caused by its transition into assembly code.

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.

2.5 Random number generators

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(). The random number generator in the AVR version of standard library avr-libc is the standard linear congruential generator [10]. For this platform, RAND MAX is not defined as the highest $2^{31} - 1$, but only $2^{15} - 1 = 32767$.

Initial seed takes the value of 1 by default. If we start again after disconnecting the card, the state of the pseudo random generator will reset and start from the same initial state. For this level of DPA the linear congruential random number generator from the standard library is sufficient since it does not target the generator in attempts to exploit it. If we wanted more randomized mask generation, a source of entropy would be needed available to the smart card such as using a built-in real-time clock or the bits of ADC that are most noisy. 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. Each restart would guarantee distinct numbers. However, we would not be able to truly randomize the increments since it would still be predictable.

2.6 Card firmware

The modified version of Simple Operating System for Smartcard Education (SOSSE) is run by the smart card. Each time we change the executable the card needs to be re-programmed with a programmer. Writing the boot program code that would be able to update the application program code section is another possible way to update the implemented code, but the use of a 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.

2.6.1 APDU

The communication between our custom implemented functions and smart card interface is mediated by SOSSE. Application protocol data unit (APDU) is used in implementing the communication. The communication unit between the smart card and the reader defined in ISO/IEC7816 [11] is the APDU. The communication between the card and reader is divided into APDU command and APDU response.

2.6.1.1 Command

Card receives the command by the reader. Mandatory header length should be at least 4 bytes containing the instruction class, two bytes of instruction parameters and instruction code. 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.

Offset for writing might be indicated by the parameters P1 and P2. However, they are not used after they are set to 0x00.

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

Our sent data, Lc, can be defined as 0, 1 or 3 bytes. Each byte length may contain values from 1 to 255. 3 bytes indicates the maximum data length, but if we are making the use of 3 bytes length, the first byte must be set to zero. Thus, only values from 1 to 65535 can be contained in it. 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.

2.6.1.2 Response

Received by the reader from card, contains two bytes SW1-SW2 with command processing status and the response data. Command processing status gives us the detail of whether it has been successful or contains an error. If the return code is 0x9000, it shows that it has been successful. In case the return code is 0x6800, it means that it is an unsupported instruction and for 0x6a00, it tells us that an unexpected length of the command or of the expected response was encountered.

2.6.2 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.

2.7 Masked AES-128

The implemented AES-128 function encrypts one block of input using an embedded key. The masking scheme makes the use of Boolean masks. Some of the masks were exclusive-ORed with the plaintext and some others were exclusive-ORed with the first round key. Following sections describe the masking schemes that were encountered and chosen for this project.

2.7.1 Key expansion

A constant array contained the full key in our masked implementation mainly because the key was not expected to change and the expansion that was taking place before the execution of the algorithm was slowing down our measurements.

The results of the simulation before masking showed 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.

2.7.2 Key addition

Key addition is implemented by the AddRoundKey function of our algorithm. The unprotected version of our code only ex-ORed the plaintext with the key byte by byte. The function was then changed in the masked implementation where it loaded a random $mask_{in}$ and random precharge value from the global context. The destination register is precharged with the random precharge value before the loading of the round key. Each time a new byte of the key is loaded, the same value is precharged to the register.

After the random precharge, the $mask_{in}$ value needs to be added. The key value is executed with the instruction EOR in the destination register and mask in the other one. The value that is leaking the Hamming distance during the step of the register write back phase is the mask, not the key. The masked key byte is exclusive-ORed with the corresponding byte of the plaintext.

2.7.3 SubBytes

The only non-linear operation of AES is SubuBytes. Typically, this is implemented as a table look-up. Hence, we used a masked S-box table for this operation.

The S-box look-up table, which is stored in the program memory, is needed by the function that describes the $SubBytes()$. The function makes the use of one byte of block as an index into the look-up table and makes the substitution for that byte with the value that is stored in that index.

2.7.4 S-box masking

This is implemented in the function $maskSbox()$ which iterates over the S-box look-up table and fills in another look-up table with the masked S-box:

```
void maskSbox(){
  uint8_t i = 0;
  uint8_t * sbox_ptr = sbox_mem;
  register uint8_t x0, x1, x2, x3;
  do {
    x0 = pgm_read_byte(sbox_ptr++)^mask;
    x1 = pgm_read_byte(sbox_ptr++)^mask;
    x2 = pgm_read_byte(sbox_ptr++)^mask;
    x3 = pgm_read_byte(sbox_ptr++)^mask;
        sbox_masked[i++^mask] = x0;
        sbox_masked[i++^mask] = x1;
        sbox_masked[i++^mask] = x2;
        sbox_masked[i++^mask] = x3;
        } while(i);
}
```

If the input masks equal to zero, the key value would not be masked and there would be no difference. However, the S-box substitution would do mask removal and the value obtained by Sbox substitution would be the unmasked product of key addition and S-box substitution. This would be easily detected by using the Hamming weight power model and the values in the rest of the execution would stay unmasked.

If the value of the inputs and outputs of the mask were equal, the execution would be like the single mask masking scheme which is not sufficient in our case due to the drawback that the Hamming distance between the key addition and S-box substitution product leaks information.

2.7.5 ShiftRows

This operation moves the bytes of the state to different positions. At this point of our algorithm, all the states are already masked with the same mask because of which this part does not affect the masking. It rotates each row to the left by zero, one, two or three bytes.

```c
void shiftRows(void){
  register uint8_t s0,s1, *state;
  state = (uint8_t *) (*block);
  s0=state[1];
  state[1]=state[5];
  state[5]=state[9];
  state[9]=state[13];
  state[13]=s0;
  s0=state[2];
  s1=state[6];
  state[2]=state[10];
  state[6]=state[14];
  state[10]=s0;
  state[14]=s1;
  s0=state[15];
  state[15]=state[11];
  state[11]=state[7];
  state[7]=state[3];
  state[3]=s0;
}
```

Each byte is moved directly to its new position instead of byte by byte which helps us in making the code more efficient.

2.7.6 MixColumns

MixColumns were implemented by using multiplication using 02 and 03 polynomials from Rijndael finite field as a function. This is important because it mixes the bytes from different rows of a column. If two masks are used for the bytes of a column, we need to make sure that all the intermediate values stay masked which is inefficient. Therefore, it is better to make sure that each row is masked with a separate mask at this point of our algorithm. The same operations are performed on each column and a for loop was used for the algorithm.

```c
void mixColumns(void){
uint8_t *state = (uint8_t*)(*block)  ;
register unsigned char s0,s1,s2,s3;
register unsigned char c0,c1,c2,c3;
register short i;
for (i = 0; i < 4; i++) {
  s0=*(state++);
  s1=*(state++);
  s2=*(state++);
  s3=*(state++);
  c0 = field_2x(s0) ^ field_3x(s1) ^ s2 ^ s3;
  c1 = s0 ^ field_2x(s1) ^ field_3x(s2) ^ s3;
  c2 = s0 ^ s1 ^ field_2x(s2) ^ field_3x(s3);
  c3 = field_3x(s0) ^ s1 ^ s2 ^ field_2x(s3);
  state -= 4; //rewrite current column, move to the next column

  *(state++) = c0;
  *(state++) = c1;
  *(state++) = c2;
  *(state++) = c3;
}
}
```

This implementation was also used by keeping a single pseudorandom mask throughout the algorithm which was precomputed.

2.8 Differential power analysis

Differential power analysis could have been implemented using Wolfram Mathematica or even MATLAB. For this project, we chose Wolfram Mathematica due to its familiarity. Measurements that were done through the oscilloscope were loaded in Mathematica where we chose to run the DPA.

2.8.1 Range

The range of the traces was chosen such that the important operations of the round were in them. It should be wide enough for the key addition and S-box substitution at least since that is where we expected leakage to occur. In case we missed these operations, later computations for the Hamming weight model and the Hamming distance model would not leak any information. Similarly, if the range is too wide, it would mean additional time needed for the correlation process to complete.

When choosing the range for a masked implementation, we should be more careful since the repeating round patterns tend to dissolve and we cannot see a drastic difference between them. Therefore, before we chose the range for the masked implementation, we decided to choose the range for the case where the masks and precharges were equal to 0x00. Similar ranges were chosen for the masked implementation then, given that the oscilloscope configurations remained constant as shown in figure 2.5.

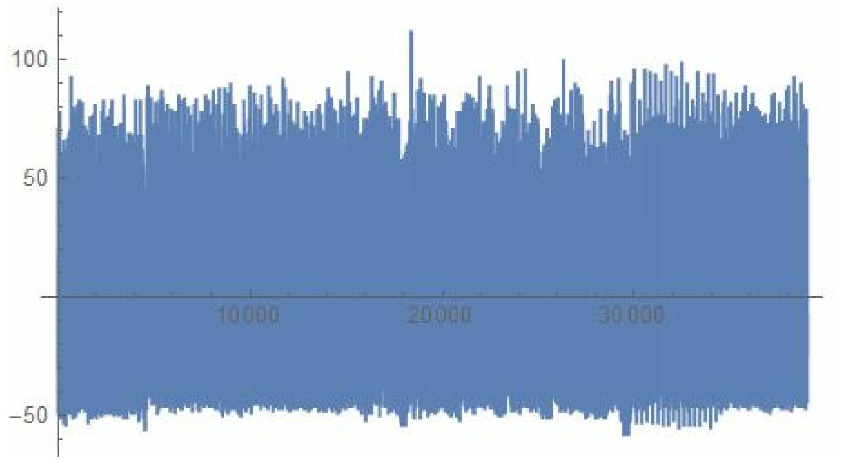


Fig. 2.5. Range of traces chosen for the masked AES-128 implementation

2.8.2 Power models used

We used the Hamming weight and Hamming distance power models. For the unmasked version of the code or in the case where the mask was set to 0x00, we used single order DPA and the Hamming weight power model. This was successful. Later, we also tested second order DPA with both the Hamming weight model and the Hamming distance model. However, the original DPA and the Hamming weight helped us to acquire the key one byte at a time. This was done by first generating all 256 possible byte values of the key. Then we applied AddRoundKey(0) on all possible pairs of keys x plaintexts. The next step was to apply SubBytes() on the combinations of keys and plaintexts. After the computation of the Hamming weight following these operations, we correlated the resulting matrix with a selected number of power traces and a selected range. The coefficient with the highest correlation in the matrix was chosen. The row which contains the highest coefficient is the correct key value and the column marks the offset in trace where the leakage occurred. This can occur in multiple offsets since the Hamming weight leaks several times after the SubBytes operation.

As mentioned earlier, this model was successful in retrieving all the correct bytes of the key using a total of 200 measured plaintext inputs and a range of 100000 samples. The correct correlation coefficient was always significantly better than the ones that followed it. This method, however, was not successful for the masked implementations.

The Hamming distance model differs from the Hamming weight model in the way that it computes the Hamming distance of the application of the AddRoundKey(0) on all pairs of keys x plaintexts with the application of SubBytes() on the combinations of keys and plaintexts. For the second order DPA, we targeted the S-box input and the S-box output since we believed the leakage to occur over there.

For this attack, we measured the power consumption of our masked AES implementation and we compressed the traces so that we could reduce the number of points. After identifying where the first round begins and correctly choosing the starting position and the length, we tried to identify the interval where the first S-box look-up took place. This was done by first finding out the number of clock cycles in our code that were needed from the beginning of the encryption to the operation of $SubBytes()$ during the execution of the first round. The instruction that was chosen was around the time where the first S-box look-up is likely to occur. The clock cycles that we found out from the code for the operation where we expected the leakage were 7695. Using this value and our chosen values for the trigger position and the start, we were able to depict the position in the traces. The following formula was used for that calculation:

$$\text{Position} = (\text{No. of Clock cycles})*(25) + (\text{Trigger postion}) - (\text{Start}) \tag{2.15}$$

The 25 in the formula was the value of a single clock cycle. We were able to attain the value of this clock cycle by plotting a single trace and zooming it to detect the value. Also, we were able to cross-check it by using the frequency from the oscilloscope and calculating the value of a single clock cycle. Our trigger position was 12,803 and our start was at 190000. We chose a total of 2000 plaintext inputs. Since the SubByte() position was found out to be at 7695, we chose a range of 7690 - 7705 so that we could use our operations on this particular part of the traces. Using these values and plugging them into our equation, we got 10053 and 10428. The Hamming distance was applied around these points through which we were able to break the first key byte for the masked implementation. Using the same equation with values that were shifted a little bit in front of the previous, we were able to attain the rest of the bytes. A loop was used inside Mathematica that contained both the Hamming weight and Hamming distance power model. This loop helped us to choose our power model with ease and we did not have to change the data manually on every evaluation of the notebook.

2.8.3 Used tools

We used AVRStudio 4.19 with a GCC compiler for the AES implementation. The hardware was done using the ATMega163 smart card. The power measurement was done on Agilent DSO-X 3012A oscilloscope. Inputs and outputs were controlled by the SC power measurement program that helped us in generating the plaintexts, ciphertexts and the file for the traces. This program makes the use of the linear congruential generator from the standard library and the generated plaintexts sequence is always the same for each respective session.

The DPA of the measured traces was then done using Mathematica 10.3.0.0. Though the software features were suitable for our work, it was slow when large amounts of data were used as input. For instance, when doing the second order DPA for our masked implementation, we had to load 2000 plaintexts and breaking each byte would take a considerable amount of time.

Chapter 3

Evaluation

Our results will be talked about in this section. We will discuss the success of our implementation, conclusions after applying single order and higher order DPA and finally, comparison with the unmasked version of the code.

3.1 Evaluation Criteria

The section will look at various aspects used for assessing the characteristics of the countermeasures and we will talk about their efficiency.

3.1.1 Quality

The quality of the differential power analysis lies mainly with finding out the correct key. It depends on the chosen number of traces, amongst other factors. It could happen that some of the key bytes may be found out correctly and some others may not be correct.

For the key bytes that were guessed correctly, we need to look at the correlation coefficients. The value of the correlation coefficients lies between -1 and $+1$. In our implementation, the highest correlation value is chosen and looking at the difference between the the second best value of the correlation coefficient, we can guess whether the found key byte is correct or not. If the distance is large, the found byte is usually usually correct. In the case where the distance is small, the key byte may be incorrect.

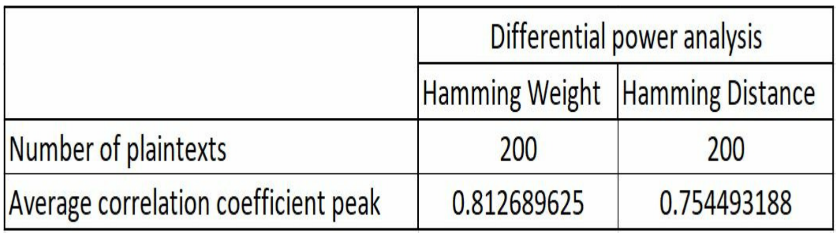
3.1.2 Efficiency

In order for our implementation to be efficient, we need to have a balance between security and time. We can not let the encryption take up too many clock cycles for operations that could be managed in less. Similarly, we want our algorithm to be as secure as possible and we would need operations that might take additional time and so, finding a good balance between the two is important.

3.2 Results

3.2.1 Unmasked AES-128

The unmasked version is the one where the value of the mask is $0x00$. The program was loaded onto the smart card and the traces were measured. We chose a total of 200 plaintexts for this version. In the analysis using single order DPA, we tried to analyze the output of the SubBytes operation in the first round. Both the power models were used for this version: Hamming weight and Hamming distance. As expected, they were both successful with high correlation coefficient peaks. The following table shows the observed results:



	Differential power analysis	
	Hamming Weight	Hamming Distance
Number of plaintexts	200	200
Average correlation coefficient peak	0.812689625	0.754493188

Fig. 3.1. Average correlation peaks for unmasked implementation

The above given data in table 3.1 shows us that the correlation peaks were quite high for the key bytes and we did not need many traces to successfully attack the implementation. All 16 key bytes were retrieved.

For the hamming distance implementation, we used the second-order DPA before trying it for the masked implementation. For the first key byte, we were able to see two high peaks. The first peak was for the SubBytes operation and the second was for the MixColumns. As talked about before, this is where we expected the leakage. The peak for SubBytes occurred at 10293, which helped us to get the clock cycle at which the leakage occurred. This was done by using equation 2.15. The clock cycles found were 7699. For the second peak, which occurred at 17830, the clock cycles found were 8000. Using the values of these clock cycles, we are able to depict the exact instructions that leaked information.

The number of plaintexts that were encrypted during the measurement stage were directly proportional to the correlation coefficient peak. Using a large number of plaintexts would give us better coefficients, ensuring that our key bytes were correct. Since an average correlation coefficient peak of 0.81269 is already quite high, we did not need to use a larger number of plaintexts.

The correlation coefficient peaks for the first byte are shown below in figure 3.2. From the plot, we can observe that the peaks have good correlation

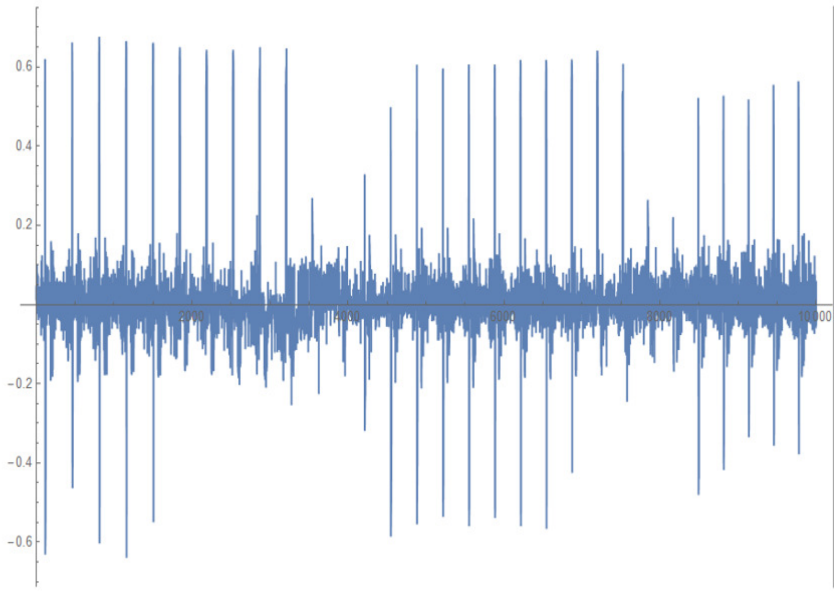


Fig. 3.2. Correlation peaks for the first byte of the unmasked implementation

coefficients for the unmasked version. The peak with the highest correlation coefficient is then chosen to get the correct key byte.

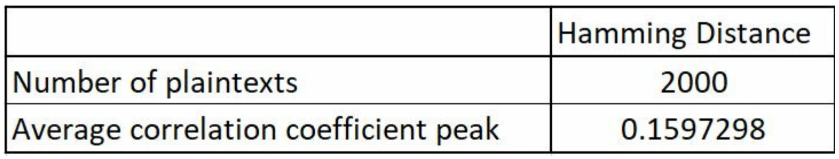
3.2.2 Masked AES

For the single and random mask implementation, we were able to find the correct key using second order DPA and a total of 2000 plaintexts. Using the information about where the leakage occurs from the unmasked masking scheme, we were able to use that over here to find the clock cycles and then the position in the traces. The clock cycles from the code for the leakage during the operation $SubBytes()$ were 7695. Using equation 2.15 and choosing the range of clock cycles from 7690 to 7705, we were able to get the points in the traces and we applied our second order DPA on those points. We used the absolute difference preprocessing function which is shown below:

$$\text{Table}[\text{Abs}[t[[All, P1]] - t[[All, P2]]], \{P1, 10053, 10427\}, \{P2, P1+1, 10428\}]; \tag{3.1}$$

The equation 3.1 is a representation of the preprocessing function that was used to find the first key byte for the single mask masking scheme. t is the matrix of traces and $P1$ and $P2$ are the two points that we discovered from the equation 2.15 using the clock cycles found around the leakage. As we can see above, the equation computes the absolute difference of all the values and $P1$ is incremented from 10053 to 10427 while $P2$ is incremented from $P1+1$ to 10428. The matrix that we get from this result is transposed and then the Hamming distance model is applied to get the correlation.

The table 3.3 below shows the average correlation coefficient that was obtained for all the correct key bytes.



	Hamming Distance
Number of plaintexts	2000
Average correlation coefficient peak	0.1597298

Fig. 3.3. Average correlation peaks for masked implementation

However, our correlation peaks were quite low. A larger number of traces would give us better peaks but would also increase the time needed to process the data. If an attacker would be trying to attain the correct key, he would most definitely opt for larger number of inputs so that his correlation coefficients would be more convincing since he would not be aware of the correct key. Figure 3.4 shows the values of the correlation coefficient peaks for the first byte. We can clearly see the difference from the unmasked implementation by having a look at the values of the peaks on the y-axis. The same range of the traces and the same key bytes were chosen for the comparison of these two. We can see that the peaks for the masked implementation do not go beyond 0.11 in this plot.

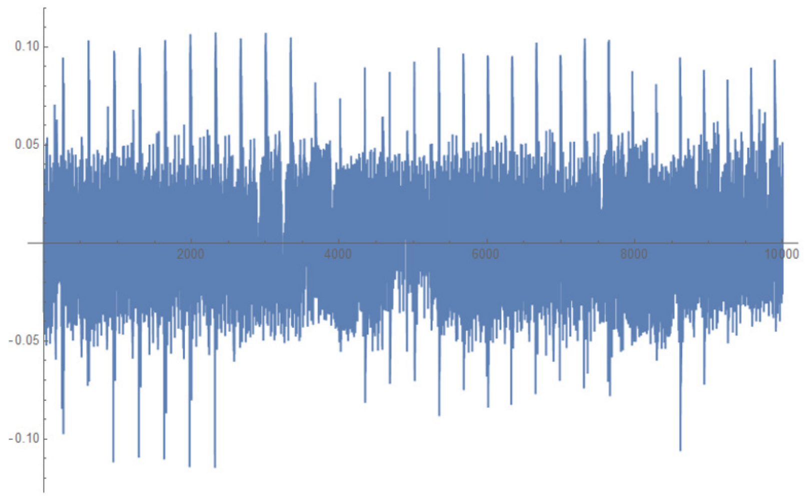


Fig. 3.4. Correlation peaks for the first byte of the single mask implementation

Finally, we had to apply second-order DPA on our multi-masked masking scheme. Unfortunately, our implementation contained the rand function before the processing of the rounds and we were not able to guess the correct clock cycles for the operations during SubBytes that were leaking information. Our times were shifted on each iteration of the code due to hiding which was not the case for our single mask masking scheme. When trying to move the rand function to the end of the round, other issues started to present themselves which were not faced due to time constraints and will be touched in the future. I believe that if we are able to successfully move the rand instructions to the end of the round, we would be able to exploit this using second-order DPA, just as we were able to exploit the single mask masking scheme.

3.3 Efficiency

The single mask masking scheme was much faster from our multi-mask masking scheme since our functions were less time consuming. Instead of using nested for loops, we opted for directly assigning values. The highest number of clock cycles were needed by the *maskSbox* function which iterated over 256 values and used XOR for each value twice, consuming more memory as well.

Improvements could be made by implementing the functions in assembly. However, that may make them incompatible with some other platforms while C code is easily usable and can be compiled on machine code without any changes.

Conclusion

Summary

This thesis started with the description of AES and possible countermeasures to protect it from side channel attacks. Our used countermeasures involved two masking schemes - single mask masking scheme and multi-mask masking scheme.

Differential power analysis was initially used on an unprotected version of AES in which the mask was set to zero. Single-order DPA was more than enough to break this implementation. Thereafter, single-order DPA was attempted on the single and multi-mask masking scheme. This, however, was not successful. Following this conclusion, second-order DPA was applied to break the masked implementation. We were able to break the single mask masking scheme using second-order DPA after which we tried it on the multi-mask masking scheme. This was not successful because we were unable to isolate hiding from masking in our code. Our rand instruction that computed the pseudo-random number that was used for the mask, was at the beginning of the round which was not the case in the single mask masking scheme. Due to this, we were unable to depict the clock cycle around which the leakage occurred and because of that, we were not able to successfully use the second-order DPA on it.

Mainly, the implementation lacks a true random number generator which would surely be needed to generate such a mask that can not be predicted easily. However, ATMega163 has very limited options for the implementation of this task so this was out of scope for this thesis.

Future work

The implementation of the multi-mask masking scheme can surely be improved. First, the rand function should be used at the end of the round without disrupting other parts of the code. Secondly, it can be made more efficient by avoiding the usage of nested for loops and using better algorithms to complete the operations in the rounds.

While processing large inputs, like when we measured the encryption of 2000 plaintexts, Wolfram Mathematica was quite slow and we were not able to compute the second-order DPA for all bytes along with the computation of the correlation coefficients in a single loop. This had to be done separately for each byte. One probable solution would be to automatize large data inputs to enable larger than 1 GB to be correlated without running out of memory.

Bibliography

[1] Zhang, L.; Vega, L.; Taylor, M. Power Side Channels in Security ICs: Hardware Countermeasures. *arXiv preprint arXiv:1605.00681*, 2016.

[2] Mangard, S.; Oswald, E.; Popp, T. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[3] Sochůrková, A. Programové prostředky obrany proti diferenciální odběrové analŷze. 2016.

[4] Kocher, P.; Jaffe, J.; Jun, B. Differential power analysis. In *Annual International Cryptology Conference*, Springer, 1999, pp. 388–397.

[5] FIPS, P. 197, Advanced Encryption Standard (AES), National Institute of Standards and Technology, US Department of Commerce, November 2001. *Link in: http://csrc. nist. gov/publications/fips/fips197/fips-197. pdf*, 2009.

[6] Daemen, J.; Rijmen, V. *The design of Rijndael: AES-the advanced encryption standard.* Springer Science & Business Media, 2013.

[7] Wiener, M. *Advances in Cryptology-CRYPTO'99: 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999 Proceedings.* 1666, Springer Science & Business Media, 1999.

[8] Chakraborty, R. S.; Matyas, V.; Schaumont, P. *Security, Privacy, and Applied Cryptography Engineering: 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings*, volume 8804. Springer, 2014.

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

[10] Park, S. K.; Miller, K. W. Random number generators: good ones are hard to find. *Communications of the ACM*, volume 31, no. 10, 1988: pp. 1192–1201.

[11] *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange [online]*. Available from: `https://www.iso.org/standard/54550.html`

Acronyms

AES	Advanced Encryption Standard
CMOS	Complementary Metal-Oxide-Semiconductor
DPA	Differential Power Analysis
EEPROM	Electrically Erasable Programmable Read-Only Memory
GCC	GNU Compiler Collection
RISC	Reduced Instruction Set Computing
SCA	Side-Channel Attack
SPA	Simple Power Analysis
SRAM	Static Random Access Memory

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ..................................... the directory of source codes
  aes single mask faster . the source code of the single mask masking
  scheme
  Mathematica experiments..Results obtained during the experimental
  part
  Multi mask masking code the source code of the multi-mask masking
  scheme
  thesis .............. the directory of LaTeX source codes of the thesis
    *.text ...................... the LaTeX source codes of the thesis
thesis.pdf ............................ the thesis text in PDF format
```