Comparative Study of AES-Based Authenticated Encryption Algorithms in Matters of Security and Performance on a Microcontroller Platform

Daniel Brolli
daniel.lobenwein@student.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Stefan Mangard
Assessor: Thomas Korak

May, 2015

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

Ich bestätige hiermit, diese Arbeit selbständig verfasst zu haben. Teile der Diplomarbeit, die auf Arbeiten anderer Autoren beruhen, sind durch Angabe der entsprechenden Referenz gekennzeichnet.

Daniel Brolli

Acknowledgements

The last months have been a thoughtful and rewarding journey. I have received support and encouragement from a great number of persons. I would like to express my gratitude to my supervisor Thomas Korak for his helpful comments, sharing his expertise, and his exceptional support in all stages of my thesis. Furthermore, I would like to thank the people at the *Institute for Applied Information Processing and Communications* providing me with helpful input, information, and assistance. Additionally, I would like to extend my sincerest thanks to *evolaris next level GmbH* allowing me to flexibly adjust my working time in order to conclude this thesis in the best way possible.

Finally, I would like to express my greatest appreciation to my wife, Anna Brolli, for her extraordinary patience during my work and all my friends, who supported me in any way possible. Special thanks to Andreas Plaschka for proofreading my thesis.

Abstract

Securing sensitive data has become more important these days, as it was decades ago. Due to the fact, that devices, such as sensors, smartphones, tablets, or computers are now more interconnected, the amount of transmitted data increases. In case of sensitive data, unauthorized access should be avoided. The sensitive data should be secured in such a way, that only persons, allowed to read the data, are able to do so. In order to accomplish confidentiality of data, the given data has to be securely stored or transmitted in an encrypted way. Hence, the need for confidentiality, integrity, and authenticity is an important issue nowadays. A lot of algorithms have been established during the last years in order to fulfill these requirements. Authenticated Encryption with Associated Data (AE) algorithms provide these requirements. In order to extend the portfolio of already existing AE algorithms, like GCM, CCM, or OCB, the CAESAR competition has been established in 2014. In this thesis we evaluate the algorithms, submitted to the CAESAR competition. Four algorithms are selected based on previously defined criteria: AES-based algorithms, optimized for embedded systems, and similar size of input data. The selected algorithms are discussed and evaluated in detail according to their security and performance, with a special focus on embedded systems. Therefore they are implemented on a microcontroller of the MSP430 family in order to validate their suitability for embedded devices. Additionally, these algorithms implemented on a microcontroller are expected to be vulnerable against physical attacks. Hence, a differential power analysis (DPA) attack is performed on the algorithms. The results confirm the expected vulnerability of the algorithms against DPA attacks. In order to secure the implementations a recently proposed masking countermeasure was added. This countermeasure results in a large overhead in terms of execution time, but the previously successful DPA attacks on the algorithms do not succeed anymore.

Keywords: microcontroller, MSP430, DPA, AES

Kurzfassung

Das Schützen von sensiblen Daten ist heutzutage wichtiger als es vor Jahrzehnten nötig war. Der Grund dafür liegt vor allem darin, dass die Geräte, wie Sensoren, Smartphones, Tablets, oder Computer, heutzutage viel vernetzter miteinander sind und deswegen die Menge der versendeten Daten signifikant ansteigt. In den meisten Fällen ist es aber nicht erwünscht, dass diese Daten von jedem gelesen werden können. Es sollte hingehend so gesichert sein, dass nur Personen, die dafür bestimmt sind die Daten zu lesen, dies auch tun können. Um diese Vertraulichkeit zu erreichen, müssen die Daten verschlüsselt gespeichert bzw. verschlüsselt übertragen werden. Viele Algorithmen wurden in den letzten Jahren entwickelt, welche Vertrauen, Integrität, und Echtheit gewährleisten. Authentifizierende Verschlüsselungsalgorithmen mit zugehörigen Daten (AE) erfüllen genau diese Vorraussetzungen. Um das Portfolio von existierenden AE Algorithmen, wie GCM, CCM, oder OCB zu erweitern, wurde 2014 der CAESAR Wettbewerb ausgerufen. In dieser Arbeit werden wir Algorithmen evaluieren, die bei dem CAESAR Wettbewerb eingereicht wurden. Vier Algorithmen wurden basierend auf zuvor festgelegten Kriterien ausgewählt: Algorithmen müssen AES basierend sein, sie sollten für Mikrocontroller optimiert sein, und sie sollten ähnliche Größen bei den Inputdaten haben. Die ausgewählten Algorithmen werden bezüglich Sicherheit, Performance und Anwendbarkeit für eingebettete Systeme genauer diskutiert und evaluiert. Weiters werden sie auf einem Mikrocontroller der Familie MSP430 implementiert, um festzustellen, ob die selektierten AE Algorithmen geeignet für Mikrocontroller sind. Zusätzlich wird angenommen, dass diese Algorithmen durch physikalische Attacken angreifbar sind. Um diese Verwundbarkeit zu verifizieren, werden die Algorithmen mit Hilfe einer sogenannten 'Differential Power Analysis (DPA)' attackiert. Die Ergebnisse bestätigen die Annahme, dass alle ausgewählten Algorithmen angreifbar sind. Um die Angriffe zu erschweren, wurden die Implementierungen mit einer kürzlich vorgestellten Gegenmaßnahme gesichert. Diese Gegenmaßnahme führt dazu, dass die Laufzeit signifikant erhöht wird, jedoch schlagen die selben DPA Angriffe jetzt fehl.

Stichwörter: Mikrocontroller, MSP430, DPA, AES

Contents

Chapter 1

Introduction

Lots of devices nowadays need to be capable of processing, transmitting, and storing sensitive data e.g user data on smartphones, communication data in a network or measurement data recorded by various connected sensors, just to name a few. This data is confidential and should not be accessible in an unauthorized way. In order to achieve protection against such an unauthorized access, the information needs to be stored and transmitted in a secure way. Various cryptographic algorithms and primitives can be applied to ensure confidentiality, integrity, and authenticity. Authenticated encryption with Associated Data (AEAD) algorithms, like CCM (Counter with CBC-MAC) or GCM (Galois/Counter Mode), fulfill those generic requirements. In 2014, the CAESAR competition [3] has been started with the goal to further extend the portfolio of authenticated encryption (AE) algorithms. More than 50 algorithms have been submitted and are evaluated by independent researchers all over the world. The algorithms differ in the underlying mode of operation, as well as their performance in hardware and software implementations. Also the theoretical security of the algorithms is examined in detail in the course of the challenge. However the possibilities an attacker gains if he has physical access to the device, executing such an algorithm, are often disregarded. Those so-called physical attacks can be performed in a way, that the attacker either alters the way the device operates (active attacks) or by observing leaking side-channel information (passive attacks). Such side-channel information can be gained by observing e.g. the power consumption [26], the electromagnetic field [13], or specific timing behaviors [27]. In case of active attacks, the device's execution path is actively manipulated by inducing faults. Those faults can be induced by altering e.g. the power supply, modifying the clock signal [16], exposing the chip with a focused laser beam [39], or running the device out of its specified operating condition(e.g. temperature). The way such an attack is performed depends on the device, but can be subgrouped into three different attacks, depending on the damage done to the device. If the chip is not modified at all, it is called non-invasive. The device is not altered in any way and no evidence of an attack is left behind. This changes if the applied attack is either semi-invasive or even invasive. The device package has to be removed in order to gain access to the inner structure of the device, allowing more precise attacks. Depackaging, analyzing, and inducing precise faults requires highly specialized equipment and a lot more effort compared to passive attacks. Therefore passive attacks have gained popularity over the years.

In this work we implement selected AE algorithms of the CAESAR competition [3] on a microcontroller unit (MCU). The selection is based on the suitability of the algorithms for embedded devices. We will evaluate all AES-based submissions of the CAESAR challenge

and narrow them down on four candidates, which are then being implemented on an MCU and attacked. Furthermore, the practical security of the implemented algorithms is evaluated by performing differential power analysis (DPA) attacks. The selected AE algorithms are then compared to each other in order to provide a comparison in matters of performance and security. In detail, this work consists of four main goals:

- Evaluation of selected first round candidates of the CAESAR challenge by applying the following set of selection criteria. The candidates should all be AES-based with a key length of 128 bit. The used AES version should not be altered in any way and the AE algorithm should be optimized for embedded systems.
- Implementation of the selected algorithms on the MCU with focus on power management, executing the algorithm with a small amount of power to be suitable for e.g. the RFID domain [30].
- We will analyze the resistance of the selected algorithms against physical attacks by applying a differential power analysis (DPA) attack and revealing the master key of the device. Furthermore, we will try to minimize the leakage with elected countermeasures.
- The selected AE algorithms will be benchmarked with main focus on code execution time and code size. The outcome will be compared to each other and to a reference benchmark of AES-GCM. Moreover, the same benchmark will be run, after the selected countermeasures securing the algorithm have been applied. This way a direct comparison of the countermeasure's impact in matters of performance is possible.

The outline of this work is as follows: First, we discuss the evaluation and selection of the chosen microcontroller unit in Chapter 2. Next, we give a general description about Symmetric Encryption and its principles, focusing on Advanced Encryption Standard (AES) as the encryption scheme. In Chapter 4, we discuss the general primitives of authenticated ciphers and introduce available ciphers like GCM or CCM. Moreover, the evaluated algorithms of the CAESAR challenge are presented. A comparison between the different AES implementations in addition to the various implementations of the AE algorithms are discussed in Chapter 5, followed by a description of the attack scenario for each of the algorithms in Chapter 6. A full coverage of the implemented countermeasures for each attack is given in Chapter 7. Chapter 8 will give a description of the overall benchmark settings and its individual results. Conclusions are drawn in Chapter 9.

Chapter 2

Hardware

The goal of this thesis is to implement and evaluate certain authenticated encryption algorithms on a MCU (microcontroller unit). The target hardware platform, where the algorithms are implemented and evaluated, should be well suited for a wide range of applications, also including embedded systems. Those embedded systems often are battery powered or are built into RFID (radio-frequency identification) systems, therefore the used microcontroller should be capable of operating with a small amount of power. Because of that fact, the focus is on power efficient hardware, which is able to endure a long time irrespective of the available amount of energy. Several MCUs from different vendors fulfill the mentioned requirements. In the following sections we want to discuss some of them. We also want to introduce the selected MCU family, the MSP430 built by TI (Texas Instruments).

2.1 Microcontroller Portfolio for Embedded Systems

For this work three different MCUs have been considered. That would be the NXP LPC1114FN28, the Atmel ATxmega256, and the TI MSP430. The first chip uses a 32 bit ARM Cortex processor, designed for a wide range of embedded systems. It represents the smallest ARM processor based on the ARMv6-M architecture. It implements a RISC architecture with a reduced instruction set of 56 instructions, also called the ARM Thumb instruction set. The MCU itself can store up to 32 KB program code in the internal flash memory and has additionally 4 KB of SRAM. The maximum clock frequency is given with 50 MHz.
The second microcontroller, the Atmel ATxmega256 is a 8 bit microcontroller, which is based on a Harvard architecture, meaning that its program memory and data memory is separated and their corresponding buses are distinct. The instruction set, also known as the Atmel AVR instruction set, is based on a RISC architecture and supports 142 different instructions. The Atmel ATxmega256 comes with 256 KB internal flash memory, 16 KB SRAM, and 4 KB EEPROM. Its maximum clock frequency is 32 MHz. In contrast to the NXP LPC1114FN28 this microcontroller is equipped with a security module, capable of performing 128 bit AES encryption and decryption, respectively, as well as performing DES (Data Encryption Standard). As stated in the official datasheet [10] performing 128 bit AES requires 375 clock cycles.
The last microcontroller in this discussion is the TI MSP430. It is, like the others, optimized for embedded systems and therefore optimized for low power consumption. The

processor is based on a 16 bit RISC architecture, with a reduced instruction set of 51 instructions. The maximum clock frequency of the MSP430 is given with 16 MHz. Several models of the family also feature a built-in crypto engine, that is capable of performing 128 bit, 192 bit, and 256 bit AES encryption and decryption. However, its greatest advantage over the Atmel ATxmega256 and the NXP LPC1114FN28 is its memory. In contrast to the others it uses 64 KB of ultra-low-power FRAM (Ferroelectric RAM) as unified memory, meaning program code and program data is in one single space, allowing flexible code and data allocation. According to the official datasheet [21] its write and reading speed is at 64 KB in 4 ms. Due to fact that some MSP430 types are equipped with a cryptographic module, capable of performing e.g. AES, and FRAM as its memory type, the MSP430 was chosen as the used MCU and will be discussed in more detail in the next section.

2.2 Microcontroller Family MSP430

The MSP430 Microcontroller is a complete system on-a-chip. It provides a variety of functions like LCD control, ADC (Analog to Digital Converter), I/O ports, ROM, RAM, basic timer, watchdog timer or UART (Universal Asynchronous Receiver/Transmitter). The MSP430 family is optimized for low power consumption and therefore is well suited for battery-powered embedded devices, as well as for passively powered applications such as the WISP5.0 (Wireless Identification and Sensing Platform), which is a battery-free RFID platform for experimentation with low power sensing, computation, and communication [30].
The MSP430's current draw in idle mode can be less than 1 µA, as stated in the official low power guide of TI [19], depending on the controller and operating mode. The MSP430 is based on a 16 bit RISC architecture, with a reduced instruction set.

To speed up development, most of the MSP430 microcontrollers are available on a logic board, often referred as launchpad. This board simplifies working with the controller. Next to the controller unit itself, it comes with a built-in USB interface for connecting the launchpad to the PC, which is helpful for deploying new software. The board is usually equipped with general-purpose input/output (GPIO) pins, depending on the chosen MSP430 series. Those pins can be used for general functionality like triggering devices, lighting up LEDs or measuring input signal from various connected sensors. Beside the GPIO pins and the USB interface, the launchpad is equipped with a debugging interface, used for debugging the microcontroller during runtime on the computer. This feature allows to detect and remove bugs efficiently. Furthermore, the UART module allows to communicate with the device (e.g send commands or receive measurement data).

For the scope of this work we choose the MSP430 FR5969 mounted on a launchpad. This unit can be clocked up to 16 MHz and its power consumption is approximately 0.4 µA during standby mode [21]. Next to speed and low power consumption, it has two more features concerning performance and security: First, the flash memory on the MSP430 FR5969 is replaced by Ferroelectric RAM (FRAM) and its memory parts are being unified, meaning program code and program data is in one single space, allowing flexible code and data allocation. Although the memory is unified, it is still partitioned into several partitions to maintain certain access privileges (e.g. code sections should not be writeable). These memory sections are generated while linking the program and are monitored by the MPU (Memory Protection Unit) to prevent unauthorized access [20]. Second, the MSP430 has an extra crypto module, which is capable of performing AES en-/decryption. The latter is of great advantage for the evaluations performed in this

work, because the algorithms evaluated are all AES-based. In the following two sections, the FRAM and the AES module will be discussed in more detail.

2.3 Ferroelectric Random Access Memory (FRAM)

Every microcontroller needs some kind of memory to store information. This data sometimes must be stored in a non-volatile way, meaning that it should not be lost if the microcontroller is powered off.

Before FRAM was actively developed, nearly all MCUs have been equipped with EEPROM (Electronically Erasable Programmable Read-Only Memory). This kind of memory is able to keep data in its storage, even in case of a power-down. However, writing and reading is very slow on these memory types, therefore newer microcontroller are using its successor, the so-called flash memory. Technically it is still a type of EEPROM, but its main advantage is, that it is capable of erasing large blocks instead of small ones, mainly bytes. Unfortunately flash memories are still rather slow. This is primarily because once one state is set to "0", the full block containing that state must be erased as a whole to reset the state to "1". Unfortunately, this operations takes its time. These kind of limitations are not given on FRAM memory types, which brings us to the next big problem, endurance. As already mentioned there are many erasures going on during the lifetime of a controller. However, the write endurance of a flash memory is about 10^4. Having a program, which reads sensor data once every second and writes it to the internal memory, this limit would be reached in about 2.7 hours, which is quite fast.

The disadvantages of EEPROM/flash memory stated above will not occur on Ferroelectric memory. FRAM memory uses a ferroelectric crystal, whose polarization corresponds to the information stored on the memory. The energy required to polarize the crystal is relatively low compared to the flash memory, which uses a charge pump to set the information. To load so-called charge pumps, the amount of needed energy is very high. Applications nowadays are focusing on energy derived from natural sources like sunlight or mechanical change. Those applications rely on small bursts of energy, which basically means the higher the energy, the more instructions can be executed before the microcontroller needs to shut down again. Scenarios, where the microcontroller needs a few micro amperes (µA) just for a single write on the non-volatile memory, should be avoided.

These are the reasons why microcontroller manufacturers are changing to FRAM. Like stated above, it needs a fraction of power while reading or writing compared to flash memory. The write endurance of FRAM is around 10^{15} [21], which is, compared to general flash memory, whose write cycles are around 10^4, nearly unlimited. Compared to the given program example above, having FRAM as memory, the MCU would need to write approximately $31 \cdot 10^6$ years to reach this limit. However, it is not just the endurance of those new memory types, it is also the speed that matters. FRAM is capable of reading/writing at 125 ns per word. This is about 64 KB in 4 ms as stated in the official datasheet [19]. Those numbers taken into account, results in up to 500x-1000x faster read/write access compared to the traditional memory type.

2.4 AES Acceleration

As mentioned above one of the key features of the used MCU is its capability of hardware-supported AES (Advanced Encryption Standard) encryption and decryption, respectively.

The MSP430 FR5969 has a built-in hardware module, which allows to perform AES encryptions or decryptions in hardware without the need of any software implementation. The module is capable of handling 128 bit data with keys, whose length is either 128 bit, 192 bit or 256 bit. Using the AES hardware acceleration yields in a much better performance as it is with a software implementation. According to the official Users's Guide [23] a simple AES encryption with a keylength of 128 bit, 192 bit or 256 bit needs 168 cycles, 204 cycles, or 234 cycles, respectively. Focusing on hardware decryption, we can see that those given results are pretty much the same, except the case with a keylength of 192 bit. In that case the hardware module needs another two cycles to decrypt the data as seen in Table 2.1.

Keylength	Encryption	Decryption
bit	cycles	cycles
128	168	168
192	204	206
256	234	234

Table 2.1: Runtime of AES acceleration

Unfortunately relying on hardware is not always as good as it sounds. In case that the used AES acceleration module is not secure against SCA (side-channel analysis) attacks, it is not possible to implement any countermeasures. The only way to counteract is to use a software variant of AES with additional countermeasures. Having a software version in place would help designing new countermeasures and counteract attacks. In this work, critical AES encryptions will be implemented in software. This will cover all encryptions where the used key is the master key, and the plain-/ciphertext is known and not constant. Securing those vulnerable parts assures that the security is not dependent on the underlying hardware.

2.5 Low-Power Modes

The MSP430 offers different low-power modes whose primarily job it is to shut down certain hardware modules to decrease the power drain. The MSP430 family and its microcontroller can have up to five different low power modes (LPM0, LPM1, LPM2, LPM3, LPM4) excluding the active mode (AM). It should be mentioned that the low-power modes LPM3.5 and LPM4.5, respectively, give the lowest power consumption, but their usage should be limited to applications, where the controller will stay in one of those modes a long time, because it takes a long time to wake up again and all the contents of the SRAM (static RAM) will be lost.

To achieve such a power reduction throughout the operating modes, the microcontroller disables certain clocks in each of the low power states. The clock for the MSP430 family can be either provided by an internal clock signal or external sources. Considering just the internal clock generators, there are three main clock signals available from the clock module. The MCLK (master clock), the ACKL (auxiliary clock), and the SMCLK (subsystem master clock). Additionally to those main clocks, the MSP430 FR5969 also has a module clock (MODCLK) and the VLOCLK, known as the very low power oscillator, built into its hardware.

Each of the clock signals can be activated individually, if requested by a module, or

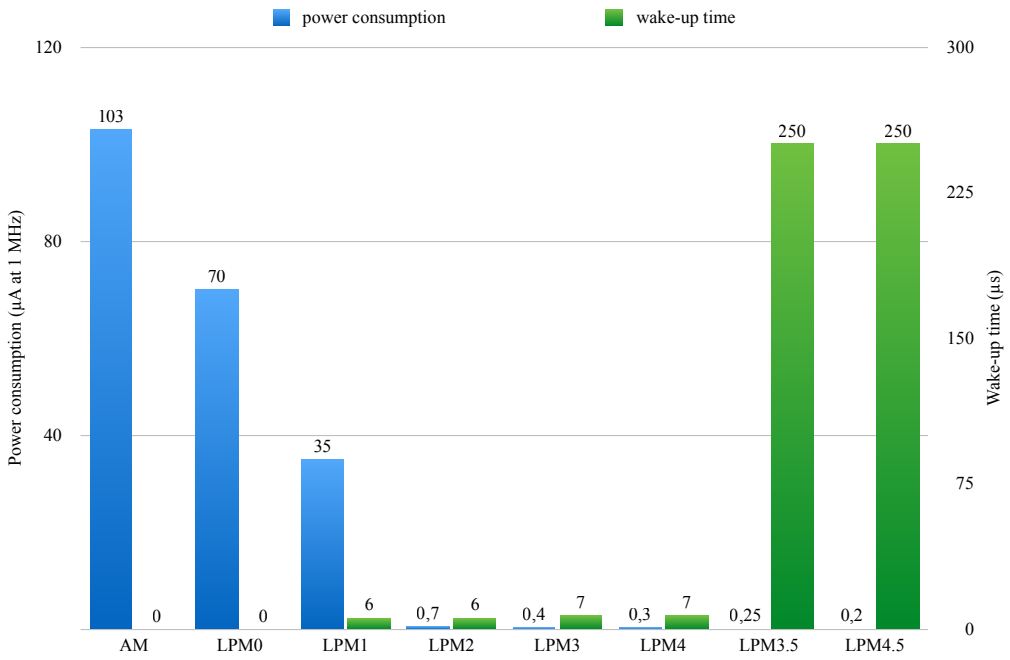

Figure 2.1: Energy consumption vs. wake up times

deactivated, to save some energy. In LPM0, all clocks, except MCLK, are up an running. Therefore the CPU will be disabled in this power mode. Additionally, the sub-system main clock can be enabled in AM up to LPM1. Lower operating modes will turn off SMCLK automatically. The clock which will be kept enabled the longest, is the auxiliary clock. Its signal is running until LPM3, LPM3 included (Table 2.2).

Clock	AM	LPM0	LPM1	LPM2	LPM3	LPM4
MCLK	**on**	off	off	off	off	off
SMCLK	optional	optional	optional	off	off	off
ACLK	**on**	**on**	**on**	**on**	**on**	off

Table 2.2: Low power modes and its related clocks

Chapter 3

Symmetric Encryption

Data encryption and decryption is an important part in all kinds of applications, where confidential data is processed. In order to secure data, which should not be accessed by unauthorized persons, the information has to be encrypted. Depending on the data to be processed and required security level, the used encryption scheme can differ. Basically encryption can be categorized into two subgroups. Symmetric and asymmetric encryption. For symmetric encryption the same cryptographic key is used for both encrypting plaintexts and decrypting ciphertexts. Asymmetric encryption is a primitive using a public and a private key in order to perform encryption and decryption. Due to the fact, that all selected AE algorithms in this work are based on AES, which is a symmetric encryption scheme, this chapter will cover all the basic definition and principles about symmetric encryption.

3.1 Definition and Principles

Symmetric Encryption is using the same cryptographic key for encrypting and decrypting data. Basically symmetric encryption can be either used as a stream cipher or block cipher. While stream ciphers are encrypting the data byte-wise, block ciphers will take a predefined number of bytes and encrypt this data block as a whole. If the input data is not a multiple of the defined block size, the data will be padded with a pre-defined padding scheme. A block cipher by itself will only allow encryption of one single data block, because the given input data will vary in length, the input data has to be partitioned and the block cipher used in a so-called mode of operation. The simplest mode of operation, also known as electronic codebook (ECB), will partition the data into separate blocks according to the cipher's block size and, if needed, pad the last block to the correct size. All this blocks are now encrypted or decrypted separately. However, such a trivial method is rather insecure, due to the fact, that encrypting the same plaintext will lead to the same ciphertext, as long as the encryption key is the same. This observation leads to the fact, that data patterns are not hidden very well. Hence, further modes of operations have been introduced. The goal is to prevent such a naive method as ECB. The basic idea is to mask the plaintext with an initial value (IV) in order to randomize the plaintext. One of the most common modes of operation is the cipher block chaining (CBC).

CBC is using a random generated IV, which is first XOR'ed to the first plaintext block P_1 and then encrypted with the block cipher E_k. The output is the first cipher block C_1. C_1 is additionally XOR'ed to the second plaintext block P_2, resulting in the next

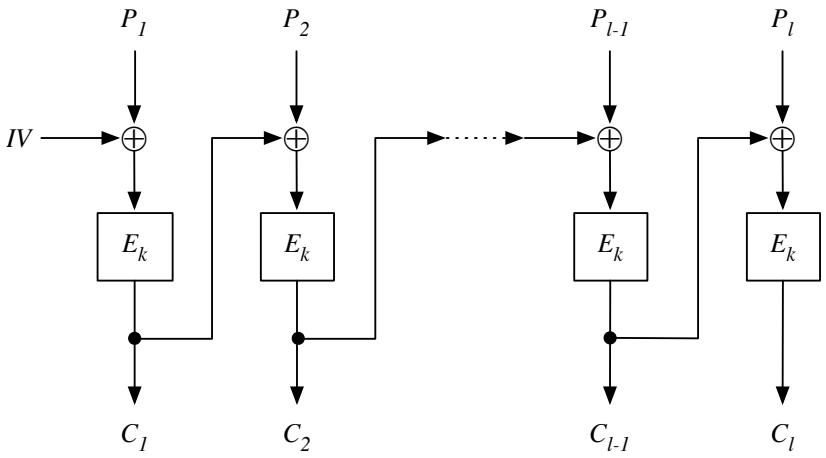

Figure 3.1: Encryption scheme of cipher block chaining (CBC)

cipher block C_2. This encryption scheme is being illustrated in Figure 3.1. Further modes of operation would be Output Feedback Mode (OFB), Cipher Feedback Mode (CFB), or Counter Mode (CTR) just to name a few. Interested readers are welcome to read the official paper published by the National Institute of Standard and Technology (NIST) [12], discussing the recommendation for block cipher modes. A common underlying block cipher for these modes of operation is the Advanced Encryption Standard (AES), which will be discussed in the next section.

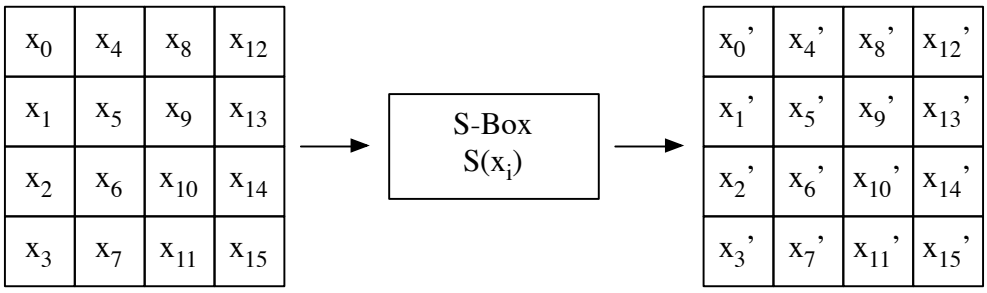

Figure 3.2: SubBytes operation of AES

3.2 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [1], sometimes referenced as Rijndael [6], is the successor of the Data Encryption Scheme (DES) and was established by the U.S. National Institute of Standard and Technology (NIST) in 2001. Developed by two Belgian cryptographers Joan Daemen and Vincent Rijmen it is able to encrypt data blocks of size 128 bit, but with three different key sizes (128 bit, 196 bit, 256 bit). Depending on the used key size, the AES algorithm uses 10, 12 or 14 rounds in order to encrypt the plaintext. Each round, except for the last one, executes the same four operations: SubBytes, ShiftRows, AddRoundKey, and MixColumns. Solely the last rounds omits the MixColumns operation.

The internal state of the AES is represented as a 4×4 matrix of bytes and all operations are performed on this AES state.
The SubBytes operation is basically a S-Box Lookup $S(x)$. The S-Box used in AES is an 8 bit substitution box, referenced as the Rijndael S-Box, which serves as a constant lookup table. This S-Box is derived from the multiplicative inverse over GF(2^8), known to have good non-linearity properties ($S(x \oplus y) \neq S(x) \oplus S(y)$). The SubBytes operation is the only non-linear function in the AES. The input of the SubBytes operation is one byte of the state x_i, which gets substituted by $S(x_i) = x_i'$ as depicted in Figure 3.2.
The ShiftRows operation is immediately followed by the SubBytes operation. The basic idea is to shift each row by a fixed value. The first row stays the same, the second row is shifted to the left by one, the third is shifted by two and the last row is shifted by three bytes to the left. Resulting in a 4×4 matrix depicted in Figure 3.3.

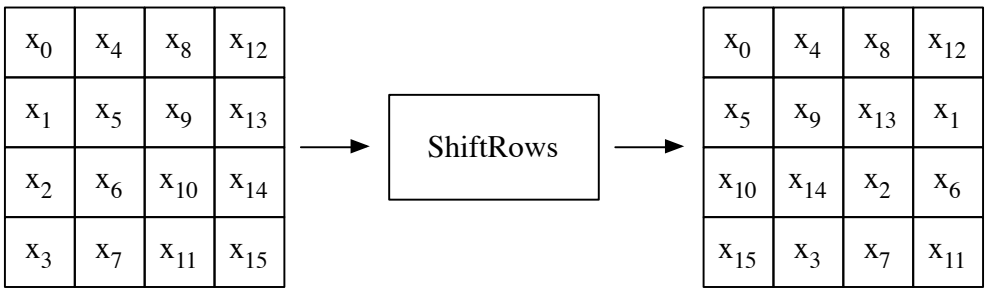

Figure 3.3: ShiftRows operation of AES

The next step is the MixColumns operation, which operates on the columns of the 4×4 state. It takes four bytes as input, multiplies the values with a fixed and pre-defined 4×4 matrix C and outputs a four byte column vector again. In combination with ShiftRows, MixColumns provides diffusion in the cipher. The basic idea of diffusion is, that changing one byte in the plaintext, causes several bytes in the ciphertext to change and vice versa. Due to this property, it is hard to find a relationship between the plaintext data and the ciphertext. The MixColumns operation is illustrated in Figure 3.4.

The last step in the AES round is the AddRoundKey operation, which will XOR the 16 byte round key to the 16 byte state. For each round a new round key is derived from the master key. This is done using the Rijndael key schedule. This key schedule takes the initial key as its input and, depending on the key size, generates 10, 12, or 14 different round keys plus the initial master key. Before the encryption of the AES starts AddRoundKey is called once to add the initial master key to the internal state.
This whole procedure applies for encrypting plaintexts. In order to perform decryption, all the operations are performed in reverse order. This is visualized in Figure 3.5. The AddRoundKey operations stays the same, while ShiftRows, SubBytes, and MixColumns have to be inverted. The inverse operations are referenced as InvSubBytes, InvShiftRows, and InvMixColumns. The InvShiftRows operation is now shifting the rows to the right instead to the left, the InvSubBytes operation has its own calculated inverse S-Box, derived from the original S-Box, and the InvMixColumns operation changes the multiplication matrix C. A more detailed discussion of AES can be found in the official paper of AES [1].

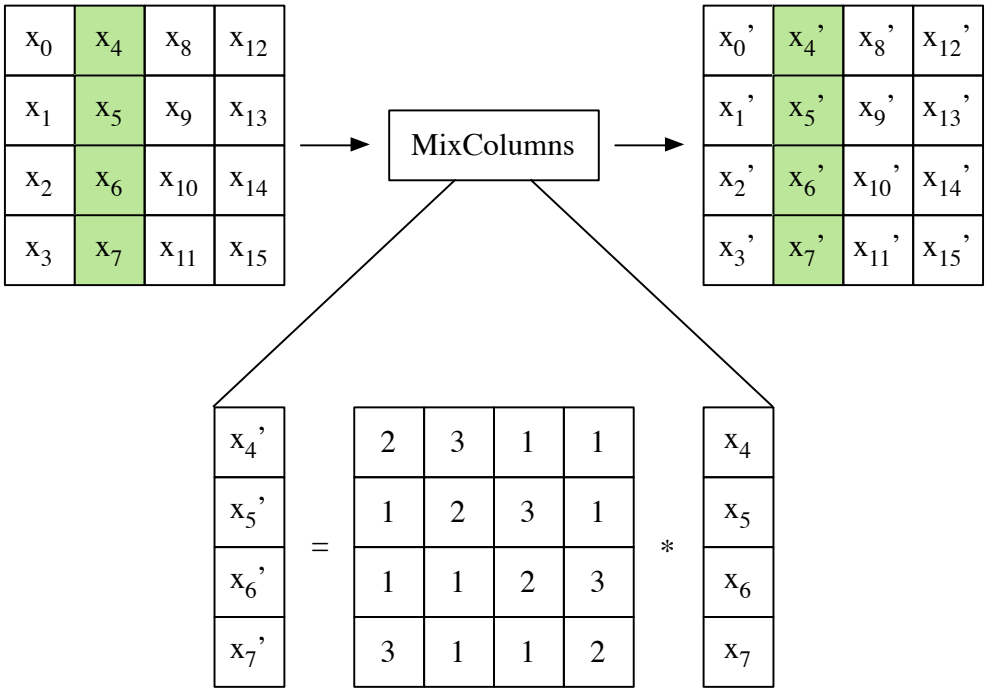

Figure 3.4: MixColumns operation of AES

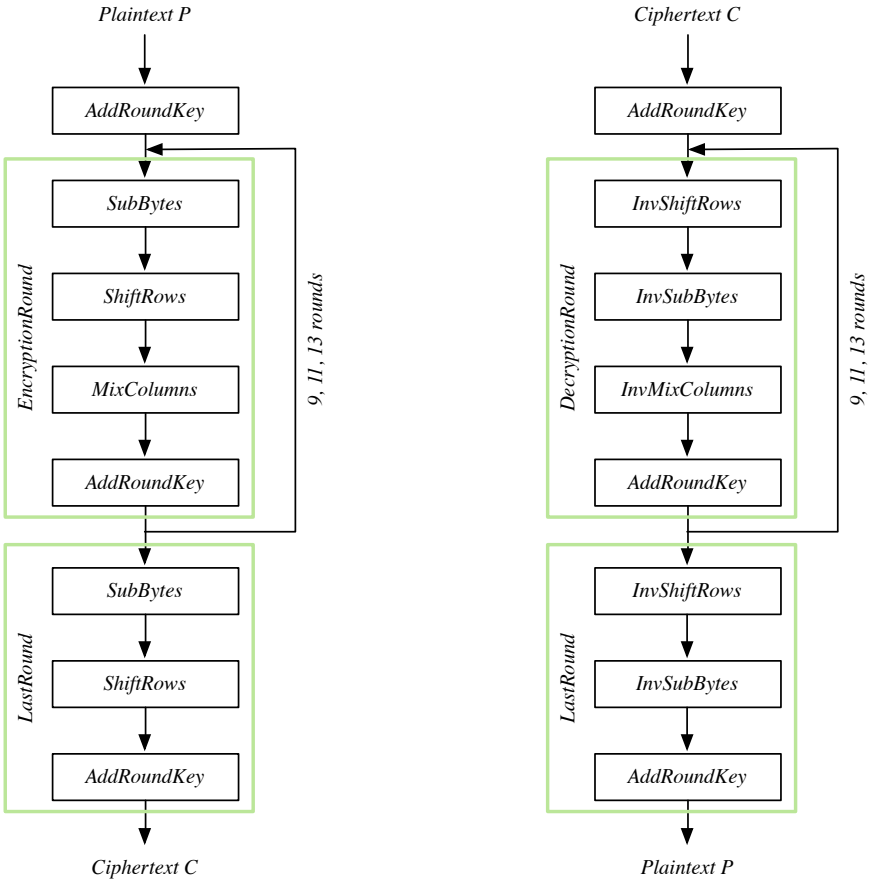

Figure 3.5: AES and its encryption (left) and decryption (right) process

Chapter 4

Authenticated Encryption

Authenticated Encryption or Authenticated Encryption with Associated Data (AE) is a mode of operation comparable to the ones discussed in Section 3.1. However, this mode of operation provides confidentiality, integrity, and authenticity of the data unlike other modes of operation like CBC, providing only confidentiality. The basic idea is to calculate a ciphertext based on a given input, and additionally create a message authentication code (MAC) based on the given plaintext and the optional header data. It should be noted that the associated data, also referred as header data, will not be encrypted. The National Institute for Software and Technology (NIST) standardized various AE algorithms, like Offset Code-Book (OCB), Counter-Mode with CBC-MAC (CCM), or Gaulois/Counter Mode (GCM), just to name a few. In 2014, the CAESAR competition [3] has been started with the goal to further extend this portfolio of authenticated encryption (AE) algorithms. More than 50 algorithms have been submitted and are evaluated by independent researchers all over the world. The algorithms differ in the underlying mode of operation, as well as their performance in hardware and software implementations. The focus of this work is evaluating those algorithms, which use AES as its encryption function. This chapter gives a general description of those aforementioned algorithms, with special focus on GCM, hence it is being used as a reference for the benchmark results. Furthermore, we introduce the evaluation criteria used to select the algorithms to be evaluated in this thesis.

4.1 Authenticated Encryption with Associated Data based on AES

Authenticated Encryption with Associated Data (AEAD or AE) is a mode of operation. As stated in the previous chapter these modes of operation are using an encryption function in order to guarantee confidentiality of the given data. However, it is sometimes the case, that confidentiality on its own is not enough. Depending on the usage, the data should be confidential and it should be possible to check its integrity, in order to ensure, the data was not modified. AE algorithms are those modes of operation combining all those three properties. The most common AE algorithms are Galois/Counter Mode (GCM), Offset Codebook Mode (OCB), and Counter Mode with CBC-MAC (CCM). All of them have been standardized by NIST and are used in everyday applications. If the underlying block cipher is the Advanced Encryption Scheme (AES), we speak of Authenticated Encryption based on AES. This thesis will only focus on AE algorithms based on AES.

4.1.1 Galois/Counter Mode (GCM)

The Gaulois/Counter Mode, in short GCM, is one of the AE algorithms, which are used in a lot of applications. This is due the fact that GCM can be implemented very efficiently and performs very well. GCM is able to process data in a parallel way, due to this fact, hardware pipelining can be used to speedup the algorithm. GCM provides integrity and confidentiality of the encrypted data and performs encryption on a block with the size of 16 bytes. GCM combines the counter mode (CTR) and the Gaulois mode of operation as depicted in Figure 4.1. In order to produce a valid ciphertext C, which can be validated with the tag T, the GCM will first encrypt the plaintext data. This is accomplished by using the CTR mode. An initial counter, containing an initialization vector IV, is set. This value is now being encrypted with the underlying block cipher E_k and is used at the end of the algorithm to generate the tag T. The initial counter is now incremented and used for the second iteration of GCM. This round is the part, where the actual encryption of the plaintext starts. The new counter is encrypted with the encryption function E_k and the outcome is XOR'ed with the first plaintext block P_1, resulting in the first ciphertext block C_1. This procedure is repeated until all plaintext blocks have been processed. In case the last plaintext block P_l is not aligned to 16 bytes, it will be padded accordingly. The last part of the algorithm is the generation of the authentication tag T in order to ensure integrity of the previously generated ciphertext C.

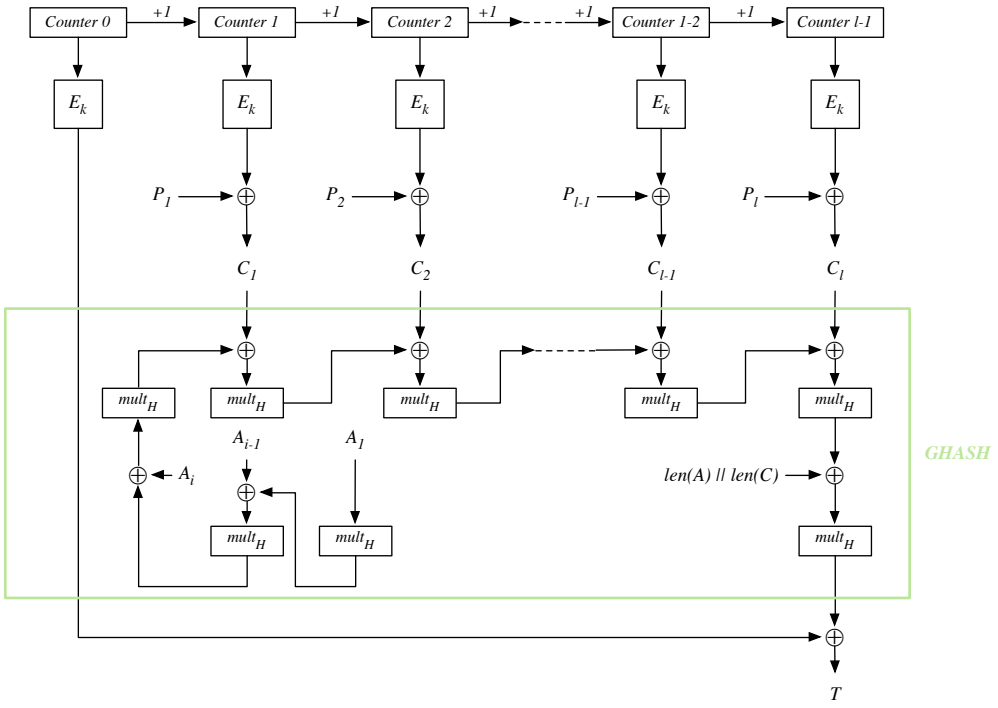

Figure 4.1: Block Diagram of the GCM Encryption

In order to authenticate the associated data in combination with the plaintext, a hash function, often referred as GHASH function, is applied. Its input is the value $H = E_k(0^{128})$, the optional associated data (AD) and the previously generated cipher C. The GHASH function as illustrated in Figure 4.2 and in the lower half of Figure 4.1

$$X_i = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus A_i) \cdot H & \text{for } i = 1, \ldots, m-1 \\ (X_{m-1} \oplus (A_m^* \| 0^{128-v})) \cdot H & \text{for } i = m \\ (X_{i-1} \oplus C_{i-m}) \cdot H & \text{for } i = m+1, \ldots, m+n-1 \\ (X_{m+n-1} \oplus (C_n^* \| 0^{128-u})) \cdot H & \text{for } i = m+n \\ (X_{m+n} \oplus (\text{len}(A) \| \text{len}(C))) \cdot H & \text{for } i = m+n+1 \end{cases}$$

Figure 4.2: GHASH function of AES-GCM

is an iterative function and the value X_i is dependent of the previous value X_{i-1}. The output of the GHASH function is XOR'ed with the encrypted IV in order to produce the authentication tag T. The basic idea of the GHASH function is, that the associated data A is first processed, followed by the ciphertext C. If the last block of either A or C is not aligned to 16 bytes, it will be padded accordingly. Each of the processed blocks will be multiplied, denoted by $X \cdot Y$, or $mult_H$, with H in a specific gaulois field GF(2^{128}). The irreducible polynomial used to reduce the result of the multiplication is $x^{128}+x^7+x^2+x+1$. Each block processed in the algorithm invokes one encryption call and one gaulois field multiplication. It can be seen, that the computation of the authentication tag, to be more precise the gaulois field multiplication of each block, can be performed in parallel allowing a much higher throughput.

4.1.2 Counter Mode with CBC-MAC (CCM)

This mode of operation is defined for block sizes of 128 bit, as it would be the case for AES. CCM operates in an authenticate-then-encrypt way. This basically means, in contrast to GCM, that first the authentication tag is generated, which is then encrypted with the plaintext. In order to generate the MAC, CBC-MAC is used as mode of operation. CBC-MAC is working similar to CBC, seen in Figure 3.1 with the only difference, that calculating the message authentication code requires no output of the intermediate cipher values C_i, except for the last block C_l. C_l is considered to be the authentication tag. The input for CBC-MAC is the plaintext and the associated data (AD). The previously calculated authentication tag and the plaintext are now encrypted using the CTR mode. The basic idea of CTR mode is, that the initial value (IV) used for the first block is being incremented for each new block. The basic encryption scheme of the CTR mode is illustrated in Figure 4.3. Due to the fact that the CTR mode is used, which encrypts each block separately, bit errors arising during transmission, will only affect the according plaintext part and will not render the next blocks unusable, as it would happen with e.g. CBC. A more detailed description of CCM can be found online at [2].

4.1.3 Offset Codebook Mode (OCB)

Offset Codebook Mode (OCB) is like GCM and CCM an AEAD algorithm providing confidentiality and integrity. In contrast to the previously introduced algorithms it does generate the ciphertext and message authentication code (MAC) in one call. This fact will lead to a faster computation, which is one of the biggest benefits of OCB. In order to calculate the ciphertext and the authentication tag, the plaintext is split up into 16 byte

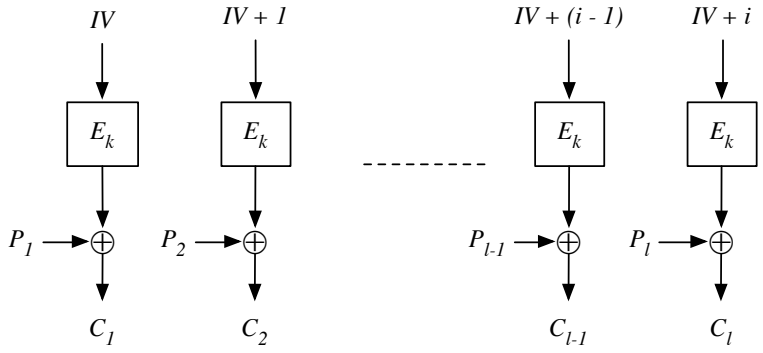

Figure 4.3: Block Diagram of the Counter (CTR) mode

blocks, XOR'ed with a value d derived from an IV, encrypted, and XOR'ed once more with the value d. The last step of OCB is the calculation of the tag. The checksum of the previously processed plaintext blocks P_i is calculated in such a way that $checksum = P_1 \oplus P_2 \oplus \cdots \oplus P_l$. The checksum is now being XOR'ed with d and encrypted. The result of the encryption is now XOR'ed with T_{AD}, denoted as the result of the AD processing ($T_{AD} = AES_k(A_1 \oplus d_1) \oplus ... \oplus AES_k(A_l \oplus d_l)$), resulting in the authentication tag. The full description of OCB can be found online at [4].

4.2 CAESAR Competition

In 2014, the CAESAR competition [3] has been started with the goal to extend the already existing pool of AE algorithms with new ones. Cryptographers all over the world submitted their proposal for a new authenticated encryption algorithm. More than 50 algorithms have been submitted. These algorithms will be evaluated by independent researchers all over the world. The competition will have four rounds each with the goal, to cancel out candidates. This competition will last until the end of 2016, as stated on the official website. The goal of this thesis is, to select all candidates using AES as its underlying mode of encryption. This set of algorithms are evaluated and thinned out based on the evaluation criteria given in the following section. The best algorithms are then picked and discussed in more detail.

4.2.1 Evaluation Criteria and Results

Due to the fact, that over 50 different AE algorithms have been submitted and the goal of this thesis is to compare different algorithms, a manageable amount of algorithms has to be selected. This was done by choosing various evaluation criteria:

- *Algorithms with default AES as its basic encryption primitive*
  The first criterion, in order to divide the amount of algorithms, was to select algorithms using AES as its mode of encryption. Furthermore, we isolated those algorithms, where the encryption function (AES) was not modified at all and supports encryption with 128 bit. The idea behind this criterion is, that the evaluated algorithms are implemented on a microcontroller unit capable of performing hardware accelerated AES execution, where no modification of the AES execution is allowed.

– *Algorithms Optimized for Embedded Systems*
Various AE algorithms submitted to the CAESAR competition are optimized for embedded systems by e.g skipping highly intense calculations in their design, like gaulois field multiplication. The aspect of being optimized for embedded systems is quite interesting, due to the fact that the evaluated algorithms are all implemented on an MCU.

– *Algorithms with Similar Input Data Size*
All the evaluated algorithms are being tested in matters of performance to each other. The idea of this evaluation criterion is, that the preferred input data sizes for the public message number and key size should be as similar as possible. This way we can assure, that all benchmarked algorithms have to process the same amount of data.

Applying the aforementioned evaluation criteria, we are able to cancel out a high number of the submitted algorithms. Starting with nearly 50 algorithms to evaluate, we choose all non-withdrawn algorithms with AES as its underlying encryption scheme. 17 algorithms remain. We now remove all algorithms, which are not using the default AES encryption scheme e.g. altered key schedule, or less rounds, resulting in 4 less algorithms. Next, we select all those algorithms with a similar input size, removing five more algorithms from the list. The last evaluation step is to find those algorithms being optimized for embedded systems. As a result, three algorithms (OTR, CLOC, SILC) are remaining. However, we slightly change the result. Due to the fact, that SILC is based on CLOC, but with less intense calculations, we substitute CLOC, with a different algorithm not optimized for embedded systems (iFeed). Furthermore, we think it is a good idea, to add one more non-optimized algorithm. This way we can directly compare two non-optimized algorithms, against two algorithms, that are optimized for embedded systems. Due to this assumption, we add COPA to our list of evaluated algorithms. A general description of the evaluated algorithms iFeed, COPA, OTR, and SILC will be given in the following sections. A visualization of the evaluation process can be found in Figure 4.4.

4.2.2 iFeed

iFeed is one of the selected algorithms to be implemented and attacked on a microcontroller platform in this thesis. It is a mode of operation for authenticated encryption. iFeed can operate with block ciphers or compression functions as its underlying primitive. However, for this thesis and in the submitted paper [40], the algorithm is based on AES-128. Its inputs are, next to the message and associated data (AD), a random key and a random public message number (nonce), which should never be repeated. If the nonce is never repeated and the key is random, iFeed can provide integrity and confidentiality for the given data. The security claims for iFeed are the same as for the underlying encryption algorithm (AES). In case of repetition of the public message number, iFeed cannot guarantee any integrity or confidentiality. While the security bounds for confidentiality is still 128 bit, plaintexts can now be distinguished from the ciphertexts, resulting in a complete loss of the pseudo-randomness.

4.2.3 COPA

COPA is the second candidate chosen to be evaluated in this thesis. In contrast to iFeed it is designed to work with block sizes with a key length of 128 bit, 192 bit, or 256 bit

Figure 4.4: Evaluation process of the AE algorithms. Algorithms using a modified version of AES-128 are marked brown. Grey algorithms are not optimized for embedded systems. The bold algorithms are the ones remaining after applying the evaluation criteria. The underlined algorithms, are the selected algorithms for this work.

only. The input data of COPA is a random key and a public message number, sometimes referred to nonce. The key can have a length of of 128 bit, 192 bit, or 256 bit and the given nonce must have a size of 16 bytes. The associated data and plaintext can be of a variable length, but the combined length of those two must not exceed the length of $2^{64} \cdot 16$ bytes, in order to comply with the security goals of the algorithm. The security claims for COPA, with the preferred settings (16 byte key, 16 byte nonce, and 16 byte tag), are all based on the birthday bound security on the block size of AES. The interesting fact is, these security claims even hold if the nonce is re-used. At least the paper claims, it provides full security, if the nonce is not repeated and full security, up to a common prefix, if the nonce is used again. The detailed description of the security goals can be found online at [9].

4.2.4 OTR

Offset Two-Round (OTR) is, like the others, a mode of operation. It is designed to work with AES as its encryption function, with all possible key sizes (128 bit, 192 bit, or 256 bit). Beside that, the nonce length for OTR is variable. It must have one byte minimum and maximum a length up to 15 bytes. Roughly the same as for the nonce length applies for the output tag length with the only difference, that the minimum length has to be four bytes, while the maximum size is limited to 16 bytes. OTR is the only algorithm capable of processing the associated data either parallel or serial, gaining a performance increase.

The security goals of this algorithm are equal to iFeed and COPA. Those security claims do not depend whether the AD was processed parallel or serial, they are the same. It should be noted, that in contrast to COPA, no nonce reuse for encryption is allowed. The full discussion about the security analysis and claims, can be found at [32].

4.2.5 SILC

The last algorithm, next to iFeed, COPA, and OTR, is SILC, Simple Lightweight CFB. As already mentioned in Section 4.2.1, SILC is based on CLOC. SILC does, in contrast to CLOC, carefully avoid hardware-unfriendly operations e.g. conditional operations, or branching, requiring multiplexers in hardware, increasing the suitability for small hardware. Furthermore, SILC, like the other algorithms too, maintains the provable security of the underlying block cipher. The underlying block cipher can be chosen, to an algorithm, that suits best. In the version implemented in this thesis, this algorithm will be AES, with a key length of 128 bit. The public message number in case of AES-128 can have a size between one and 15 bytes, while all the other input parameters are fixed. The security claims for this algorithm are discussed in more detail in the submitted paper found online at [24]. As mentioned before, like all the other selected AE algorithms, SILC maintains the provable security of AES-128. Meaning that SILC has provable security bounds up to the standard birthday bound of the AES-128.

Chapter 5

Implementation of Selected Algorithms

All of the selected algorithms are based on the AES-128 as underlying block cipher. Hence, this chapter will introduce the different implementations of AES being used throughout this work. The different AES implementations are a non-optimized version provided by Texas Instruments (TI) written in C [17], an assembler version implemented by IAIK and furthermore the hardware AES acceleration of the MSP430 FR5969 [23], allowing to perform AES en-/decryptions with very little software implementation and improved performance. Additionally, this chapter will give a general description of the implementation of every individual AE algorithm. All of the needed software was being developed and deployed on the microcontroller with the help of the Code Composer Studio (CCS) by TI [18]. Each of the implementations is power optimized by following the power advices provided by CCS.

5.1 Comparison of AES Implementations

Each of the evaluated AE algorithms is using the AES block cipher as fundamental encryption or decryption scheme respectively. That means, the performance of the AES implementation highly influences the overall performance of the AE algorithm, due to the fact, that typically several AES calls per encryption are performed. This section will introduce three different AES implementations and its benefits. A description of AES and its structure can be found in Section 3.2.

5.1.1 Software AES Version Implemented by Texas Instruments

The software version of AES provided by Texas Instruments (TI) is written in C and can be found online at [17]. This implementation only supports AES-128 encryption and decryption. In order to decrease needed code size, both operations have been merged into one function *aes_enc_dec(unsigned char *state, unsigned char *key, unsigned char dir)*. The last argument *dir* is needed in order to differentiate between encryption and decryption mode.
Benchmarking the given AES version, revealed that this version is in fact very slow compared to the versions introduced in the following two sections, illustrated in Figure 5.1. A single encryption on a MSP430 FR5969 with a clock frequency of 16 MHz requires 1.422 ms (22 752 clock cycles) and the decryption requires 2.165 ms (34 640 clock cycles). This high

difference of the execution time between encryption and decryption is mainly caused by the pre-computation of the round keys during decryption and the MixColumns operation, which is performing additional calculations during decryption.

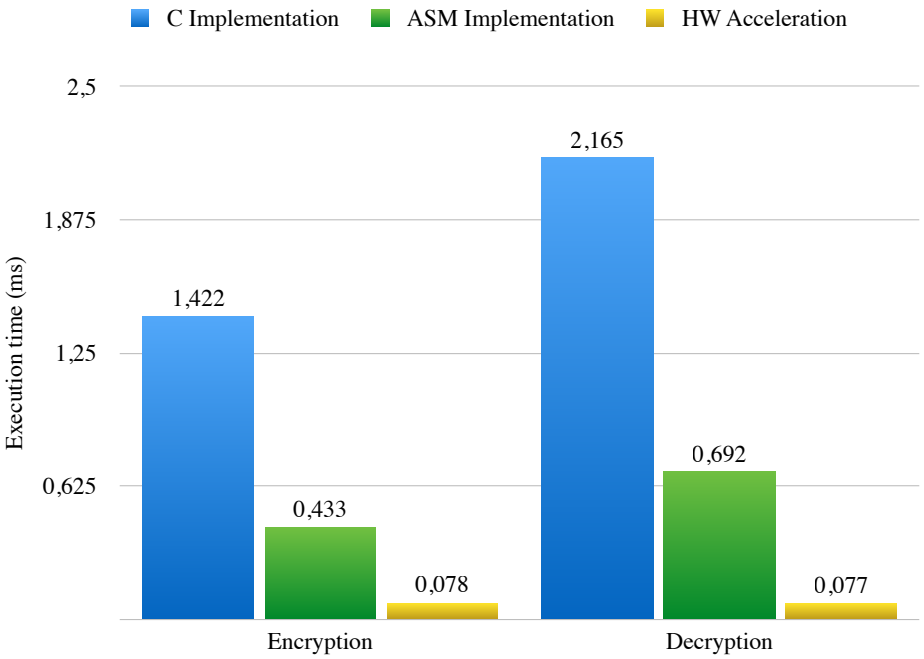

Figure 5.1: Comparison between the different AES implementations

5.1.2 Assembler AES version implemented by IAIK

The AES version provided by the Institute of Applied Information Processing and Communications (IAIK) is written in assembler code and is highly optimized in matters of speed. Moreover, the implementations aims to be executed on an MSP430. A single AES encryption requires 0.433 ms (6 928 clock cycles), while a single decryption requires 0.692 ms (11 072 clock cycles) illustrated in Figure 5.1. This results in a 3.28 times faster encryption, and a 3.13 times faster decryption compared to the pure C implementation. Due to the fact that this version aims to be implemented on a MSP430 it can take full advantage of its hardware, hence it utilizes all General Purpose Registers (R4 - R15) available on the MCU as discussed in [21]. Having a more detailed look at the implementation, it is noticeable that the SubBytes operation is executing the ShiftRows operation implicitly decreasing the overall execution time. The upper two rows of the internal 4×4 state are processed first as illustrated in Figure 5.2. The value stored in a_1 is first substituted using the S-Box $a'_1 = S(a_1)$ and stored in one of the General Purpose Registers (This process is marked in Figure 5.2 with a grey box). Furthermore the values a_0 and a_5 are substituted using the S-Box operation and shifted accordingly. This process is repeated until all values are processed. In the last step the value a'_1 from the register is loaded back into the internal state.

This procedure is repeated for the lower two rows as illustrated in Figure 5.3. The last rows in the ShiftRows operation are shifted left by 2, or 3 columns respectively. This fact

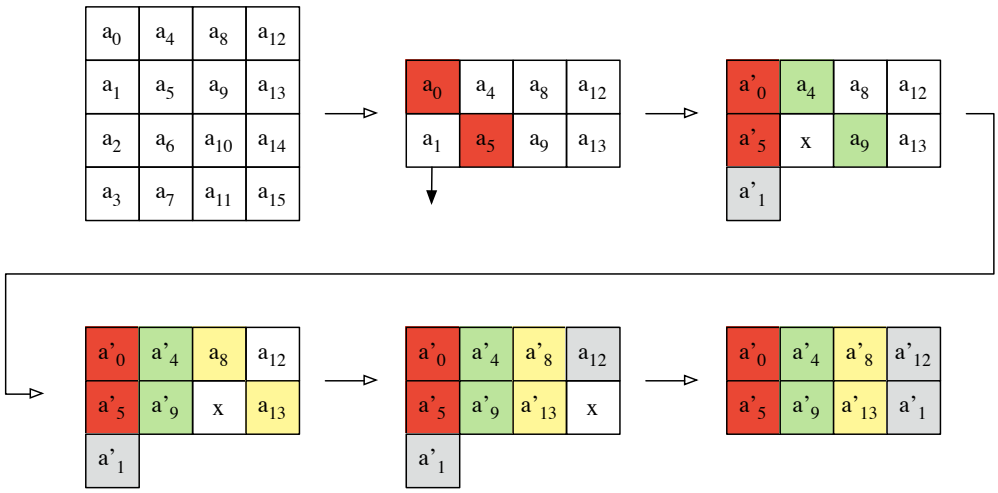

Figure 5.2: Execution of SubBytes & ShiftRows of the first two rows

forces the algorithm to store more intermediate values $a'_{10}, a'_{15}, a'_2, a'_7$ into the available registers first. Those intermediate values are now loaded back into their appropriate positions in the internal state (red and green boxes), right after a_3 and a_{11} have been substituted and loaded into one of the usable registers. The same operation is done for the values a_6 and a_{14}. In the last step those four values are then loaded back in the internal state at the correct position. This does in fact increase the overall performance of this AES implementation, due to the fact, that two out of four operations are executed at once.

5.1.3 Hardware AES Version Provided by the Microcontroller

In the this section we will introduce the hardware acceleration of the MSP430 FR5969, how it is being used and how it is performing in comparison to the two software versions introduced in the previous sections. The basics of the AES hardware acceleration have been already covered in Section 2.4, this section will give a general description of its usage.

As in the User's Guide of the MSP430 [23] stated, we need to setup the AES module first. The accelerator is configured with user software, hence the module can be completely setup with code. First of all, the operation mode of the module and the key length has to be set. This is done by setting the AESKLx (key length) and AESOPx (operation mode) bit accordingly. The AESOPx flag will determine if the operation will be an encryption (00) or a decryption (01). AESKLx will determine if AES-128, AES-192, or AES-256 is going to be performed. For the scope of this work, this will always be set to 00, telling the accelerator we want to use a 128 bit key. With those two flags set, the accelerator is setup and can now be loaded with the according key and input. The key can be loaded by writing to the AESAKEY register. Right after the key has been loaded, the input data can be loaded into the register AESADIN. As soon as the last byte has been written to AESADIN, the AESDINWR flag is set and the encryption or decryption process will start automatically, setting the AESBUSY flag to 1. It is important to mention that, while the AES accelerator is busy (AESBUSY = 1), the access to the module is generally restricted as follows:

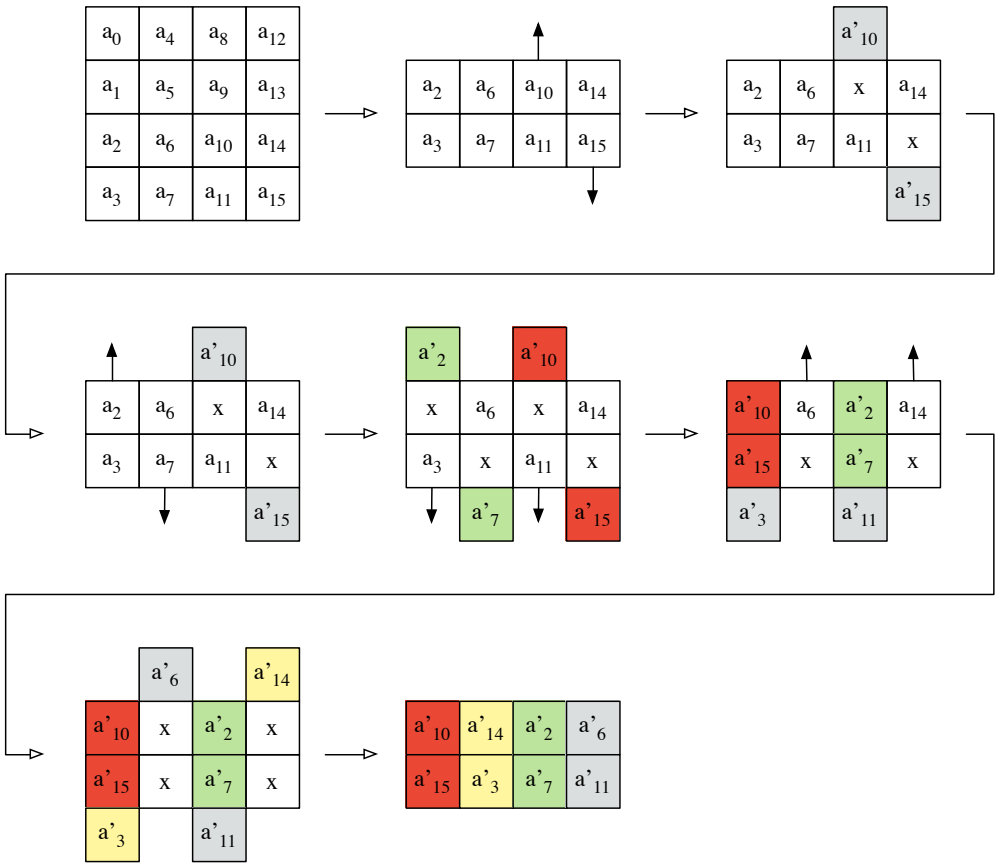

Figure 5.3: Execution of SubBytes & ShiftRows of the last two rows

– AESADOUT will always read as zero

– AESDOUTCNTx, AESDOUTRD and AESDINWR are reset

– Changing the values of AESOPx, AESKLx, AESDINWR, or AESKEYWR has no effect

– Changing the values of AESAKEY, AESADIN aborts the current operation, the error flag AESERRFG is set, and the module is reset

As soon as the encryption/decryption process has finished, the AESBUSY flag is reset and the data can be read from AESADOUT. This whole procedure is illustrated in Figure 5.4. Additionally to the AESBUSY flag, interrupts can be enabled for the AES module, which will trigger the AESRDYIFG interrupt, as soon as the current operation has finished.

The same routine applies if the mode of operation of the module is set to decryption, with the slight difference, that in order to decrypt the ciphertext, the last round key has to be generated before the decryption can start. This process can be seen in Figure 5.4.

Texas Instruments has built a library *MSP Driver Library* providing easy-to-use function calls to most of the available modules. In the implementation of the hardware acceleration, these functions have been used:

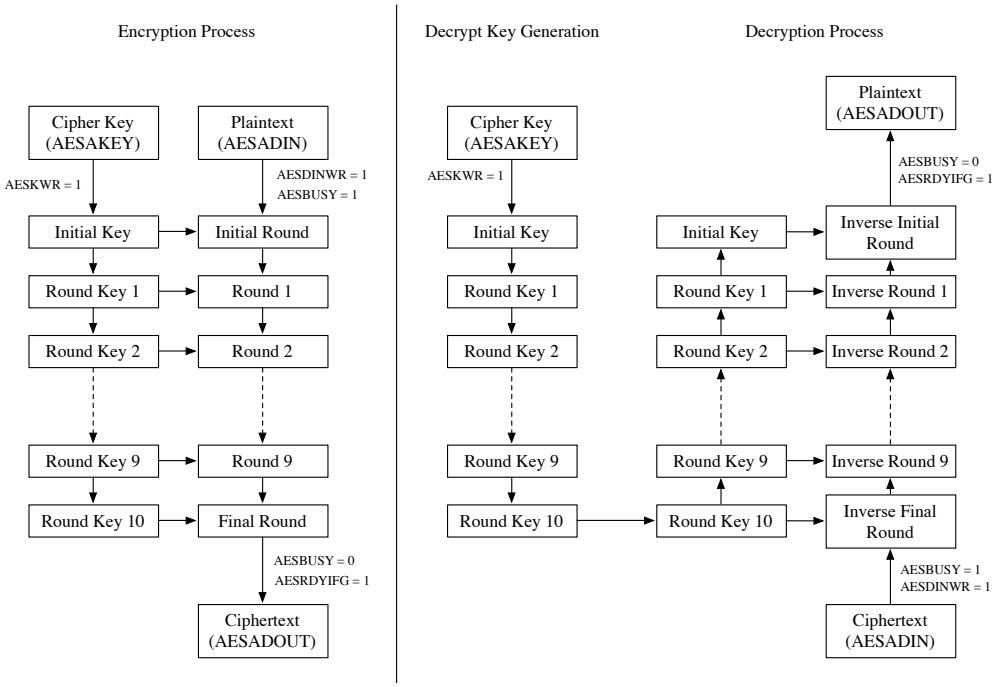

Figure 5.4: Process of a Single En-/Decryption using the Hardware Acceleration

– *AES256_enableInterrupt(uint16_t baseAddress)*
  In order to receive the AESRDYIFG interrupt, the interrupts for this module must be enabled. This is accomplished by calling this method. The only argument *baseAddress* is used to locate the AES module.

– *AES256_set(De-)CipherKey(uint16_t baseAddress, const uint8_t *cipherKey, uint16_t keyLength)*
  This function is called in order to load the according key, additionally it sets the accelerator in the right operation mode.

– *AES256_start(En-)DecryptData(uint16_t baseAddress, const uint8_t *data)*
  This will load *data* in AESADIN and start the encryption, or decryption process respectively. This function is called asynchronously, meaning that the function will return before the AES data processing has finished. In order to get the processed data, either interrupts have to be enabled and the according interrupt routine has to be implemented or polling for the AESBUSY flag is required.

– *AES256_getDataOut(uint16_t baseAddress, uint8_t *outputData)*
  Calling this function will load the data from AESADOUT into the passed parameter *outputData*.

The benchmark results can be seen in Figure 5.1. A single encryption on the hardware requires 0.078 ms (1 261 clock cycles), while decryption requires 0.077 ms (1 238 clock cycles). An attentive reader will notice, that this is a deviation to previous stated clock cycles of 168 for encryption and 204 clock cycles for decryption in Section 2.4. This is due the fact, that these numbers are actually for the pure encryption process. However, the hardware module has to set up, the cipher key has to be set, the data has to be copied to

the right register and the encrypted data has to be read from the register after a successful encryption, all needing its time to finish. In order to get a comparable benchmark result, we are taking these operations into account, resulting in a higher clock cycle count, than initially expected. It is noticeable that the speed gain compared to the software versions is substantial. Compared to the ASM implementation, the encryption is 5.5 times faster and the decryption process is approximately 9.0 times faster. The speed gain compared to the plain C implementation is even higher. One single encryption round with the hardware acceleration is 18.2 times faster. This result is being topped by comparing the decryption process. Decrypting data is nearly 28.1 times faster on the hardware than with the pure C implementation.

5.2 Implementation of iFeed

This section aims to give a general description of the functionality of iFeed and how the algorithm works, which is illustrated in Figure 5.5. This version of iFeed has been implemented with the first recommended parameter settings (Table 5.1) as stated in the official paper [40].

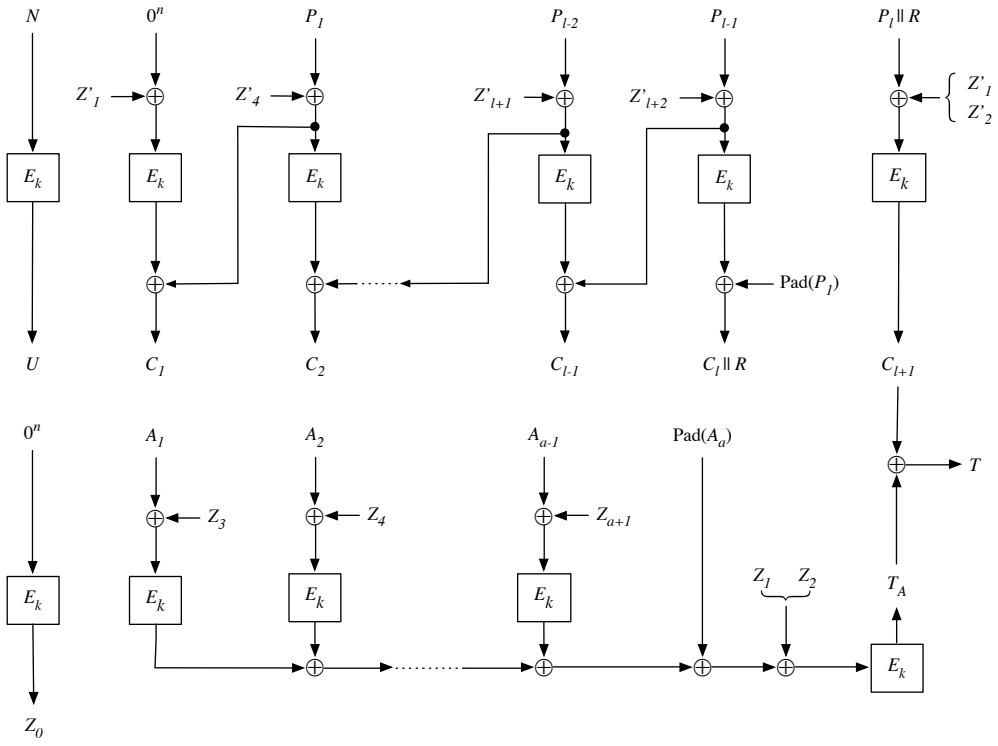

Figure 5.5: Structure of the iFeed encryption

The input of the iFeed algorithm is a 128 bit secret key K, a public message number PMN with a size of 12 bytes, a plaintext PT, and an optional associated data AD. The output of the algorithm includes a ciphertext CT with a length equal to the length of PT and a tag T. CT is a result of encrypting PT and can be used to recover the plaintext without integrity checking. In order to check the integrity, the tag T has to be verified.

Parameter Size (bytes)	1^{st} recommended	2^{nd} recommended
Key	16	16
Public Nonce	12	13
Tag	16	16

Table 5.1: Recommended parameter settings

PMN is often referred to public nonce and should be used that way, meaning that it should not be reused. If this happens in any case, the security of iFeed will be reduced.

In order to begin with the processing, two masks $U = PMN||10^{127-|PMN|}$ and $Z_0 = AES_k(0^{128}) \triangleq E_k(0^{128})$ have to be generated. Moreover, the unknown masks for the remaining parts can be pre-calculated. To be more precise, this would be $Z_i = Z_{i-1} \cdot 2$ for $i = 1 \dots max\{|AD|, |PT|\} + 2$ and $Z'_i = Z_i \oplus U$. Z_i and Z'_i are needed to mask the input data as illustrated in Figure 5.5. The output of the associated data processing T_A and the output of the last part of the plaintext processing C_{l+1} are XOR'ed in order to calculate the output tag $T = C_{l+1} \oplus T_A$.

5.3 Implementation of COPA

This section will describe the mode of operation for COPA and its input/output values. In contrast to iFeed, the key size can be set. Depending on the given key length (128 bit, 192 bit, or 256 bit), the according AES version is used. Other than that, the same parameters as for iFeed apply. The public message number, also referred to public nonce, has to be a 16-byte value, while the tag length can be between 8 bytes and 16 bytes. The version of COPA, implemented for this work, uses the recommended parameter settings as discussed in [9] with a tag and key length of 16 bytes.

In order to start the processing of the plaintext, or associated data respectively, a mask L has to be generated. This is accomplished by encrypting the value 0 with the master key ($L = E_k(0^{128}) \triangleq AES_k(0^{128})$). With L as its starting mask all derived masks $(3L, 2 \cdot 3L, \dots)$ can be calculated by performing a multiplication over a gaulois field GF(2) with a fixed polynomial $f(x) = x^{128} + x^7 + x^2 + x + 1$. For $a(x), b(x) \in GF(2^n)$, their product is defined as $a(x)b(x)\ mod\ f(x)$. The elements of GF(2^n) are denoted as integers based on the GF(2^n) conversion. As an example the value 7 can be converted into a polynomial representation of $x^2 + x + 1$.

In the first step, the intermediate value V is calculated by processing the associated data blocks A_i. The last block of the associated data is processed different, depending on the length of A. In case that the length is not equal to the block size $n = 128$ bit of the AES, the block is padded with a constant value $pad(A_i) = A_1||10*$. In order to retrieve the ciphertext C_i, the masks L and V are XOR'ed with the encryption output of the plaintext blocks P_i. The last intermediate value, right before the last ciphertext part is generated, is used to calculate the tag T. The full procedure of the algorithm is illustrated in Figure 5.6.

5.4 Implementation of OTR

Offset Two-Round (OTR) can be performed either in a parallel way as depicted in Figure 5.7 or a serial way illustrated in Figure 5.8. Both algorithms, irrespective of the used

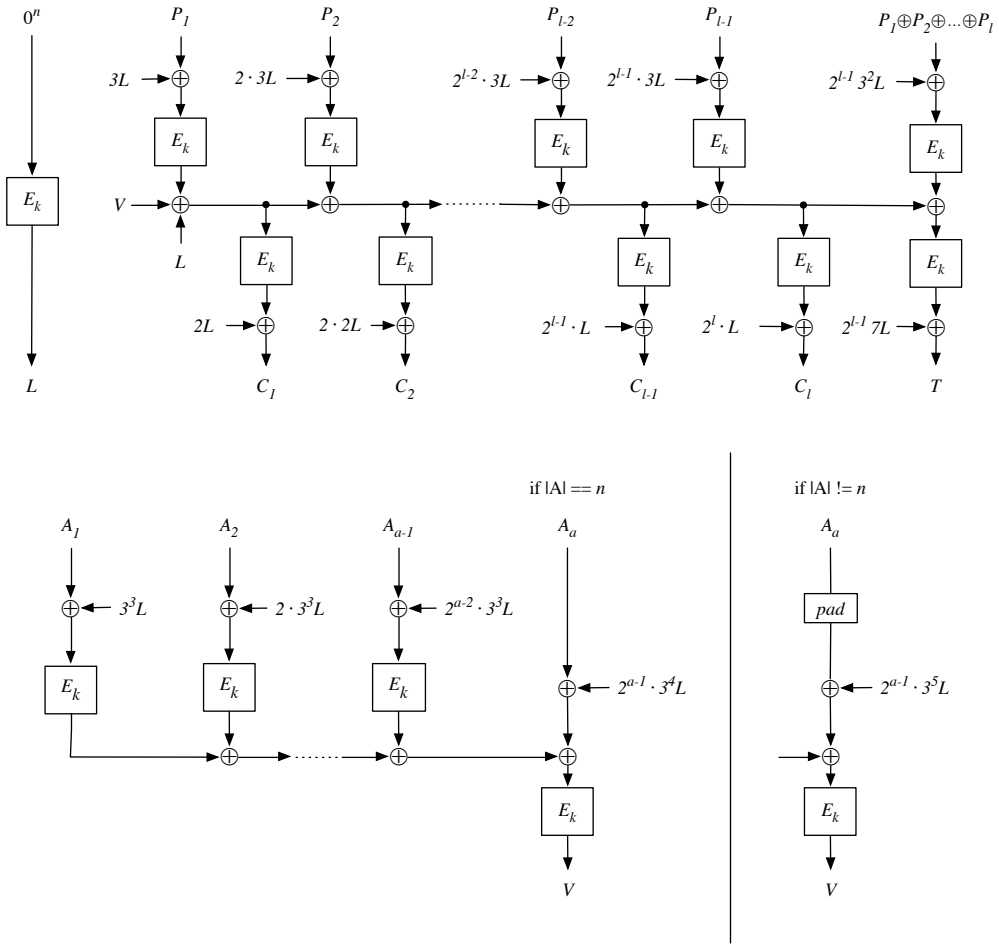

Figure 5.6: Structure of the COPA encryption

Parameter Size (bytes)	1^{st} recommended	2^{nd} recommended
Key	16	32
Public Nonce	12	12
Tag	16	16

Table 5.2: Recommended parameter settings for OTR Parallel/Serial

version, have the same input parameter set. The different parameter sets can be found in Table 5.2. Both versions are implemented using the recommended parameter set as proposed in the official paper [32]. The key and tag length are 16 bytes and the public message number has a size of 12 bytes. The difference between the parallel and the serial version is the processing of the associated data (AD). While the serial version is adding ($\oplus$) the previous processed associated data block to the current data block right before the encryption, the parallel version adds the previous block after the current AD block has been fully processed. This way the AD blocks can be computed separately and XOR'ed at the end. The order of the processing differs from one version to the other too.

The parallel version of OTR is processing the plaintext part first, followed by the associated data. This order is reversed for the serial version of OTR due to the fact, that the

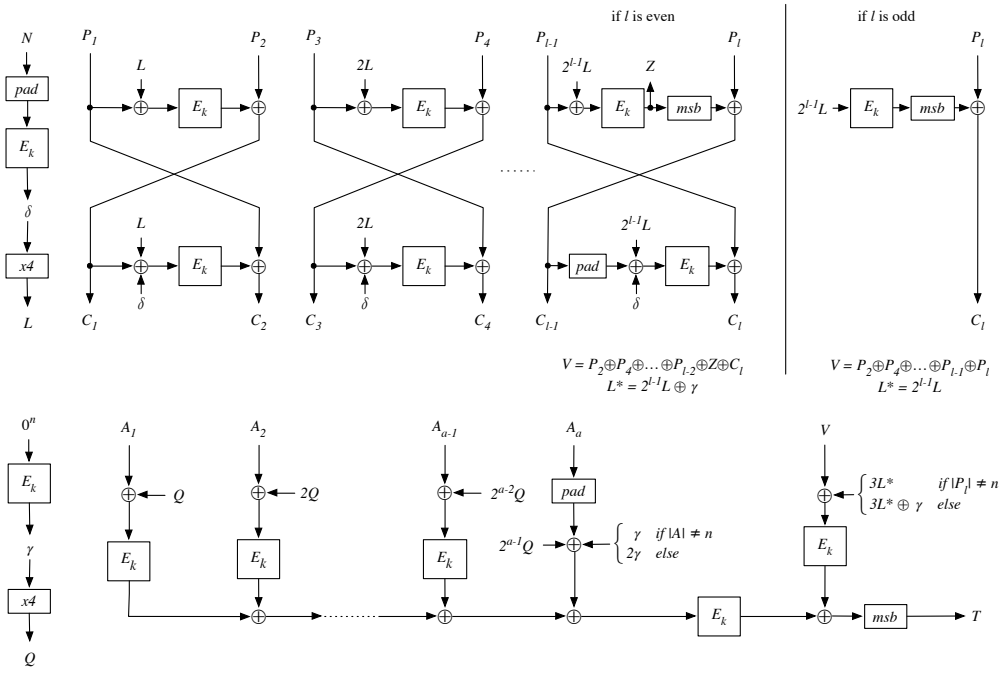

Figure 5.7: Structure of the parallel OTR version

calculation of mask L is dependent on the intermediate output of the associated data processing T_A as depicted in Figure 5.8. The mask L and its multiplications $2L, 2^2L, \ldots, 2^{l-1}L$ are used to perform the encryption of the plaintext, which is identical in both versions of the OTR. Right after handling the plaintext, the intermediate values V and L^* can be generated. V and L^* are calculated differently if the number of plaintext blocks l is even or odd, respectively. In the latter case V is an addition of all the even plaintext blocks $V = P_2 \oplus P_4 \oplus \cdots \oplus P_{l-1} \oplus P_l$ and $L^* = 2^{l-1}L$. If the count of the plaintext blocks are even, calculation of V and L^* slightly changes to $V = P_2 \oplus P_4 \oplus \cdots \oplus P_{l-2} \oplus Z \oplus C_l$ and $L^* = 2^{l-1}L \oplus \gamma$. The value Z is the result of the encryption of P_{l-1} XOR'ed with $2^{l-1}L$, while γ is denoted by $\gamma = E_k(0^{128}) \triangleq AES_k(0^{128})$.

The intermediate values V and L^* are needed to calculate the output tag T. In case of the parallel version, calculating T is also dependent on the output of the AD processing, which is not the case for the serial version.

5.5 Implementation of SILC

Simple Lightweight CFB (SILC) is the last algorithms implemented in this work. It has, like the previous discussed algorithms, a configurable parameter set. The recommended parameter sets are illustrated in Table 5.3 and are also discussed in the official paper [24]. The 1^{st} recommended parameter set (16-byte key length, 12-byte nonce length, and 8-byte tag length) has been used for the implementation on the microcontroller. SILC is first processing the associated data to calculate the intermediate value V. As it can be seen in Figure 5.9 the public nonce (N) is processed in a zero prepending function (*zpp*) to pad the nonce to 16 bytes. The last block of the associated data A_a is also padded by calling *zap*. In contrast to *zpp*, *zap* appends zeros to the input block. The last step to calculate V

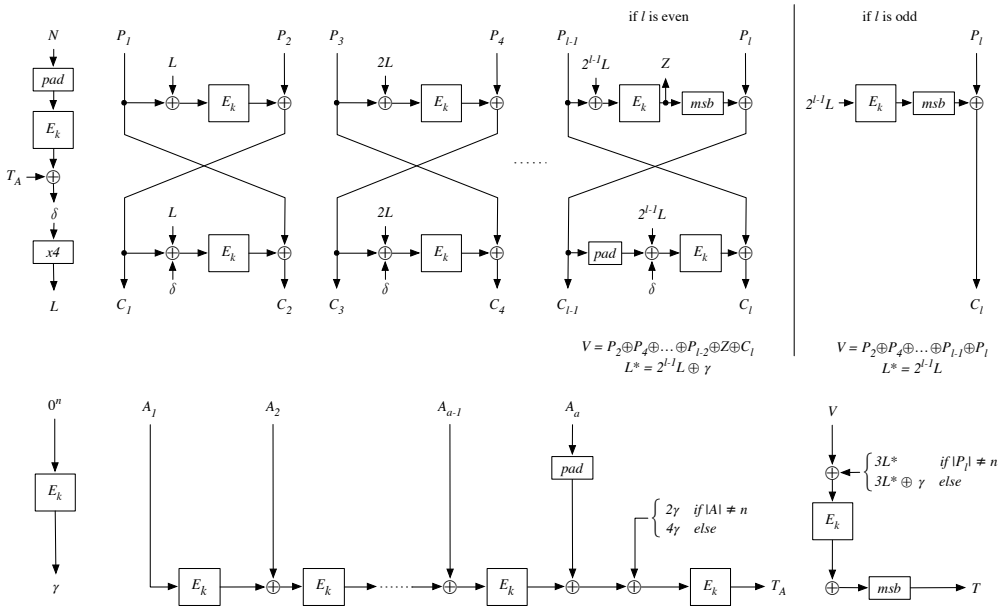

Figure 5.8: Structure of the serial OTR version

is calling the function g, that is defined as $g(X) = [X_2, X_3, \ldots, x_{16}, (X_1 \oplus X_2)]$ for X being 128 bit long and split into single byte blocks $X_1, X_2, \ldots, X_{15}, X_{16}$. This same procedure, but with eight blocks, would apply, if X is 64 bit long. However, X corresponds to the used block size of the underlying encryption function E_k. Due to the fact, that we are using AES-128 in our work, X will always have a length of 128 bit. In other words, the function $g(X)$ can be interpreted as an 1-byte shift to the left and adding $X_1 \oplus X_2$ to the rightmost output byte. This intermediate value V is now used for plaintext processing and tag generation. Furthermore it can be seen in Figure 5.9, that computing the ciphertext C_i, requires the function fix_1 to be executed, right before the encryption E_k. The function fix_1 is denoted as $fix_1(X) = X \vee 10^{|X|-1}$. That means, the value of the most-significant bit of X is fixed to '1'.

Parameter Size (bytes)	1^{st} recommended	2^{nd} recommended
Key	16	16
Public Nonce	12	8
Tag	8	8

Table 5.3: Recommended parameter settings for SILC

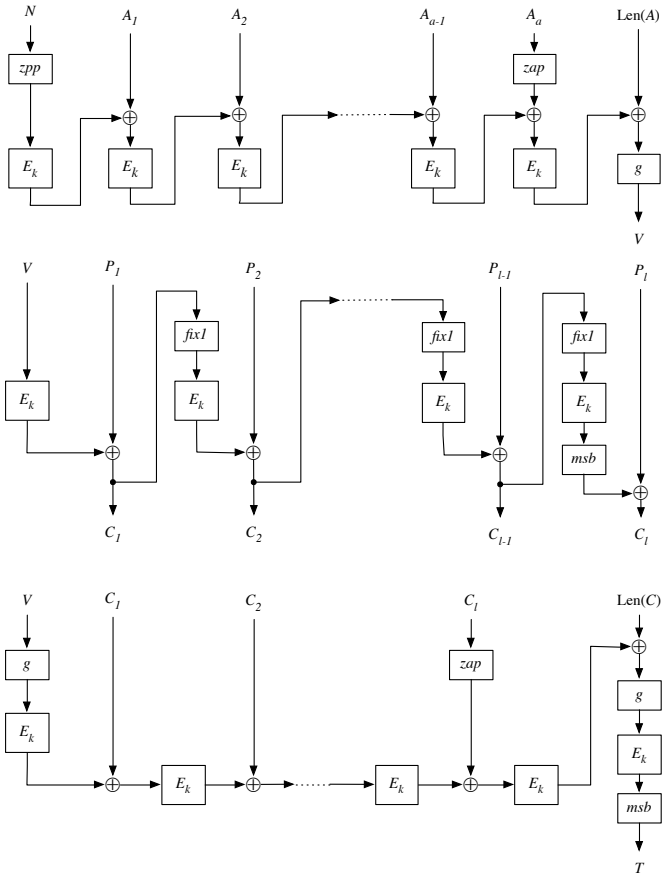

Figure 5.9: Scheme of a SILC encryption

Chapter 6

Implementation Attack

One of the main goals of this work, beside implementing and evaluating the algorithms, is analyzing their security with regard to implementation attacks, which can be grouped into two main categories, the active and passive attacks. Furthermore those can be subgrouped into invasive, semi-invasive, and non-invasive attacks, which will be discussed in the next two sections. Furthermore, the selected AE algorithms will be analyzed to find parts in the given algorithms, that are vulnerable for certain attacks. The attack, that suits best for the algorithms will be discussed in more detail.

6.1 Active Attacks

Microcontrollers require certain operating conditions to work properly. This could be for instance the temperature range, the maximum operating frequency or the supplied voltage. If those conditions are not kept, faults can occur. This observation can be used by attackers to manually induce hardware faults and therefore gain knowledge of some secret information. This is the goal of an active attack, which is also commonly known as fault attack. Using such a fault attack to break AES is discussed in e.g. [11, 36].
Those fault attacks can be categorized into three categories: invasive, semi-invasive, and non-invasive. The latter keeps the chip as is and it is not being altered in any way, there will be no evidence of an attack left behind (e.g. exploiting just the available interfaces on the board) [16, 29]. In semi-invasive attacks, the device itself is being optical inspected and, if needed, opened up, but the chip or circuits are not being altered. This attack is often used to induce faults with the help of light [39] or electromagnetic fields [34]. In contrast to those two methods is the invasive attack. It is the strongest type of attack, but also requires the most effort. There is literally no limit to what is done to the chip [28]. The difference between those methods can be found in Table 6.1.
The most popular fault attacks on cryptographic devices are spike/glitch attacks and optical attacks. The first one is a non-invasive attack and it induces faults by either altering the power supply or by modifying the clock signal for a short moment as being discussed in [16]. Optical attacks on the other hand are categorized as semi-invasive attacks and they induce faults by switching transistors with the help of light, for instance focused laser beams. Therefore the chip has to be depackaged to gain access to the silicone.

	Active	**Passive**
Invasive	Forcing Attacks (e.g. alter circuits) [28]	Probing [38]
Semi-Invasive	Optical Attacks (e.g. bit flips induced by laser) [39]	EM Analysis [13]
Non-Invasive	Spike Attacks (e.g voltage drops) [16, 29]	Timing [27], Power Analysis [26], or EM Analysis [13]

Table 6.1: Different types of attacks

6.2 Passive Attacks

While a cryptographic device is performing encryptions or decryptions, side-channel information can be observed and collected. This could be for instance timing observations, the used power consumption, optical emissions or measuring the electromagnetic field. All this information is called side-channel information and the goal of a passive attack is to process those side-channel information to obtain the desired information.
As it can be seen in Table 6.1, passive attacks, can be categorized into invasive, semi-invasive, and non-invasive as well. The modifications done to the crypto device are similar to the active attacks, the only thing that differs, is that the device is just being used to gather information. In passive invasive attacks, the targeted chip is being opened up and different components (e.g. buses) of the device are directly probed and the data signals are being observed [37, 38]. For semi-invasive attacks a typical approach is to read out the content of the memory, without the use of probing stations. To observe information in a non-invasive and passive manner, one can for instance try to measure the power consumption, the execution time or the electromagnetic field. The latter is also known as side-channel attacks (SCA). The three main types are timing attacks [27], power analysis attacks [26] and electromagnetic attacks [13]. In the next section the differential power analysis (DPA) attack will be discussed in detail. This type of attack will be used to analyze the security of the implemented AE algorithms.

6.3 Differential Power Analysis on AES Encryption

The goal of a DPA attack is to reveal the secret key by analyzing given power traces. Those traces are recorded, while the cryptographic device is encrypting or decrypting various messages. In order to reveal the secret key of a cryptographic device based on a DPA attack, five steps are required.

First, an intermediate result of the underlying algorithm has to be chosen. This result needs to be a function $f(d, k)$, where d is a known non-constant value and k is a part of the secret key. Only functions fulfilling those prerequisites can be used as an intermediate result.

Second, the power consumption of the given device needs to be measured. In order to perform this step, a vector $\mathbf{d} = (d_1, \ldots, d_D)$ has to be constructed, where each of the values in d corresponds to one known input value of the function f. Those individual values are now being used for an encryption or decryption run. While the execution is in progress, the power consumption of the device is being measured. Having a trace for each of the known inputs, a matrix $\mathbf{T}$ of size $D \times T$, where D denotes the number of plaintexts

and T denotes the length of a single trace, can be constructed. It is important, that the given power traces, which reflect the power consumption over a specific time interval, are well aligned to each other. In other words, the traces should reflect the same operation for the exact same moment, in order for the differential power analysis to work properly.

Third, hypothetical intermediate values for every possible choice of the unknown key part k, also often referred to key hypotheses, have to be calculated and are then being written into a vector $\mathbf{k} = (k_1, \ldots, k_K)$. With the given plaintexts $\mathbf{d}$ and the previously calculated key hypotheses $\mathbf{k}$, it is a simple task to compute the hypothetical intermediate values $f(d, k)$, resulting in a $D \times K$ matrix $\mathbf{V}$, where D denotes the number of plaintexts and K is the number of key hypotheses.

Fourth, the hypothetical intermediate values of $\mathbf{V}$ is mapped to the hypothetical power consumption values. This is the crucial part of the DPA, the better the chosen power model matches the real power consumption of the device, the better are the results. The most commonly used power models are the hamming-weight (HW) and the hamming-distance (HD) model [31]. The HD measures the minimum number of substitutions required to change one value x_0 into another value x_1. In other words it reflects the number of bits of x_0 that need to be changed in order to retrieve x_1. Registers of an MCU are triggered by a clock signal and therefore will change their value at a maximum of once per cycle. This observation can be used to simulate the power consumption by calculating the hamming distance of the values of two consecutive clock cycles, this would be the values x_0, x_1. Due to this fact the HD power model is well suited for attacking the device's register, if at least the consecutive values are known by the attacker. However, sometimes it is not possible for the attacker to know these values needed to apply the HD model. If this is the case, it is possible to apply the hamming-weight power model. The HW model is simpler compared to the HD model and does not rely on any knowledge of consecutive data values. This model utilizes the assumption that the number of bits set in a processed data value are proportional to the power consumption of the device. To calculate the hamming-weight of a binary value, all bits, that are set, are summed up. By applying any of those power models, it is now possible to simulate every hypothetical value v and obtain a hypothetical power consumption h, which will result in a matrix $\mathbf{H}$.

The final step of the attack is to compare every hypothetical value of $\mathbf{H}$ with the values of $\mathbf{T}$. This way, every possible key hypotheses and its related hypothetical power consumption is compared with the real power traces of the device. In order to determine the relationship between the columns h_i for $i = 1, \ldots, K$ and t_j for $j = 1, \ldots, T$ the correlation coefficient is applied (6.1). This will result in a $T \times K$ matrix $\mathbf{R}$, where every $r_{i,j}$ will contain values from up to 1 down to -1, where higher values are an indicator for a stronger relationship.

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i) \cdot (t_{d,j} - \bar{t}_j)}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i)^2 \cdot \sum_{d=1}^{D}(t_{d,j} - \bar{t}_j)^2}} \tag{6.1}$$

Figure 6.1 illustrates the rows (key hypotheses) of this matrix $\mathbf{R}$, with the black curve being the one for the correct key hypothesis.
In this section the basics of Differential Power Analysis has been explained. For a more sophisticated clarification on this topic, we suggest to read [31].

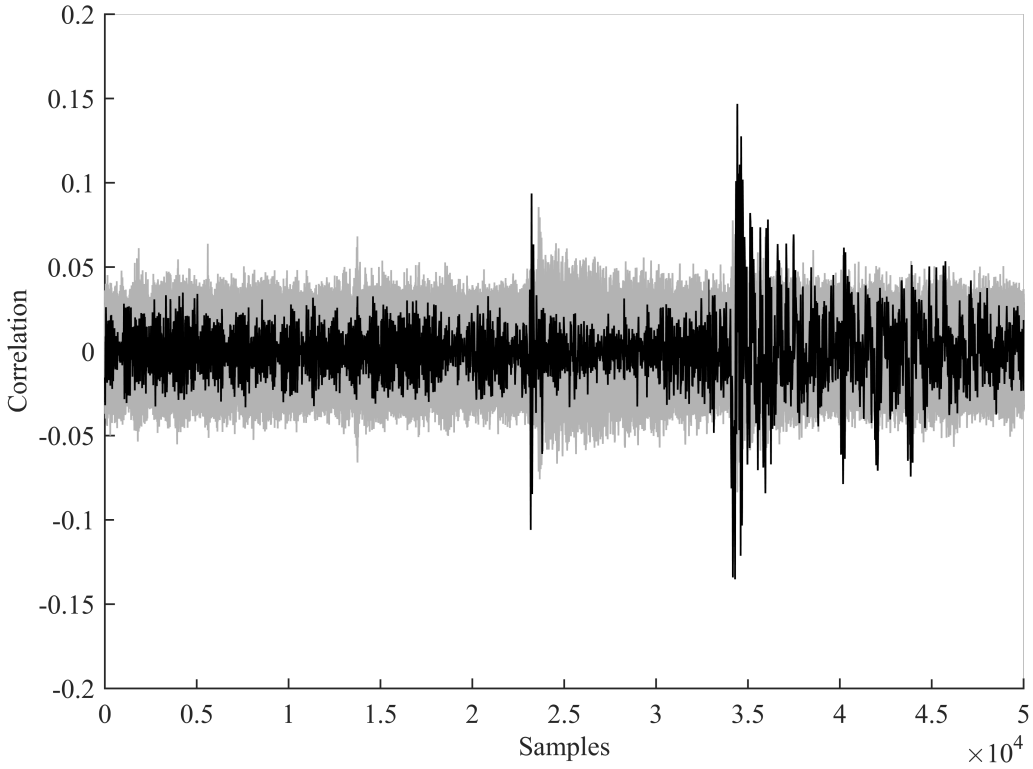

Figure 6.1: Correlation of all key hypotheses. The correct key value is marked black.

6.4 Possible Attacks Based on Selected Algorithms

The given AEAD algorithms are based on AES with a constant master key. Due to this fact we will try to reveal the secret key by mounting a DPA attack on the standalone AES. This is a common way to retrieve the secret information. In order to perform this attack, someone either needs to know the plaintext value, in order to attack the first round or the generated ciphertext is known, so the last round of the AES can be attacked. In the next few sections we will have a detailed look on each of the implemented algorithms and evaluate the possibilities on how to reveal their secret key by applying a DPA attack. The used MCU, namely the MSP430 FR5969, has a built-in crypto module, which is able to perform AES encryption or decryption. Anyway, performing AES on a hardware module has its drawbacks. If the module is not secure enough and is leaking too much information to the attacker, there is no possibility, to counteract such a leakage. It was shown in [33], that the AES module of the MSP430 FR5969 is indeed not well-secured. 100 000 traces are enough to reveal the secret key. Hence, we will focus on the software implementation of the AES, which will give us the opportunity to implement countermeasures for any leakages and swap the hardware AES to the software version in critical locations in the AEAD algorithms.

6.4.1 Attack on Software AES Encryption

The procedure of mounting a DPA attack is the same for software as for hardware implementation. First of all, it is crucial to decide the right time interval in the algorithm for the attack to work. If the input values are known, it is quite common to attack the

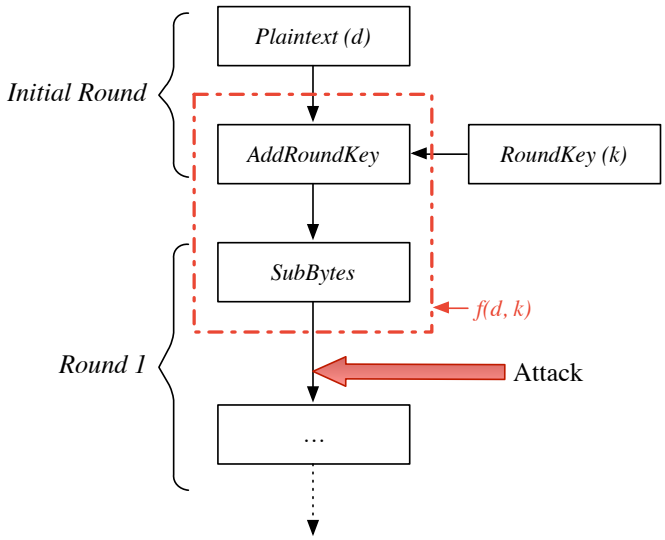

Figure 6.2: AES Attack on first round

first round of the AES. If the ciphertexts are known, but not the input values, it is typical to attack the last round of the AES. All selected algorithms have at least one call to the AES, where the plaintext is known, hence it is sufficient to attack the first round of AES.

Recalling the previous chapter about DPA, we know that we need five steps to perform a differential power analysis. The first step is to find an intermediate result of the algorithm, being a function similar to $f(d, k)$, where d is a known, non-constant input value and k is a small part of the secret key. As in the introduction of the AES already defined, the rounds are defined as follows. It starts with an initial round, where the cipher key is added to the plaintext, followed by the first round. The first operation in this round is the SubBytes operation, which is a S-Box lookup for every of the 16 bytes of the input. Splitting up the input into its 16 separate bytes, will result in a function $f(d, k)$, that is needed to perform the first step of the DPA. k being one byte of the full key and d being a single byte of the plaintext as it can be seen in Figure 6.2. Nevertheless, we could as well attack the AddRoundKey operation, instead of the SubBytes, but the latter has the benefit of being a non-linear function. Therefore changing one bit in the input value, has a bigger impact on the output, as it would be for a simple XOR operation. Additionally, a DPA attack on a simple XOR operation, will yield more than one key candidate for the given byte in contrast to the SubBytes operation. Both facts will eventually lead to a better result, if the attack is mounted on the SubBytes operation. The right time interval for the attack is now known, so step 2 can be performed, whose purpose it is to measure the power consumption during the encryption or decryption process. It is important that all of those traces are aligned to each other, which can be achieved by setting up a trigger signal, which for instance is being activated right before the AES encryption process. The measurement device (e.g oscilloscope) will now listen to this trigger and hence all recorded power traces, will be properly aligned. However, it is often the case, that it is not possible to align the traces with the help of a trigger (e.g. no software alternation possible on the given device). In this case, the traces have to be aligned manually. One way to achieve this would be to misuse the communication port of the device. If the given device starts its encryption right after the input data has been transmitted, the attacker could use this

observation to record the traces in an aligned matter. The attacker sends the input data, waits for a specified time and records the power traces. These recorded traces are now used to construct a $D \times T$ matrix $\mathbf{T}$, where D denotes the number of plaintexts and T the length of one single trace. Once the power traces have been acquired, the needed hypothetical values can be calculated. In order to accomplish this, it is important to first determine all possible key hypotheses. In the case of an first-round AES attack, this would be all permutations of one single key byte, resulting in 2^8 different values $k_i = 0 \dots 255$ for $i = 1 \dots 256$. The next step is to XOR all the key hypotheses with the corresponding plaintext byte p_j for every given trace in $\mathbf{T}$, j being the corresponding key byte of the function f. Furthermore the SubBytes operation is now applied onto the XOR'ed result, producing the needed intermediate result $f(d, k) = S(p_j \oplus k_i) = ir_{i,j}$. Finally, the hamming-weight (HW) power model is applied to retrieve the needed hypothetical power consumption $m = HW(ir_{i,j})$. All those hypothetical power consumptions m are now being saved in a matrix $\mathbf{M}$ with a size $D \times K$, D denoting the number of plaintexts and K being the number of key hypotheses.
In order to complete the DPA attack, the relationship between the hypothetical power consumption and the real power consumption has to be evaluated. One common way to achieve this is by applying the correlation coefficient (6.1). All it needs now is to find the correlation with the maximum value, the higher this value is compared to the others, the higher the chance will be that the given key hypothesis it the correct one. The last step is illustrated in Figure 6.3. It displays the various correlation values for the key hypotheses 10, 54, 90, and 249. The correlation graph for the key byte with the value of 90 (bottom left), is the highest of all four. Its value is nearly 0.15 while the maximum correlation of the other plots (key byte 10, 54, or 249) is between 0.06 and -0.06. This will lead to the assumption, that the key byte with value 90 is the correct key guess. It should be noted, that this procedure is just for one key byte. In order to reveal the full key, the same aforementioned procedure has to be repeated for all missing key bytes. Every key byte requires the hypothetical values to be recalculated with the according plaintext byte p_j to retrieve a new matrix $\mathbf{M}$. As soon as the new matrix $\mathbf{M}$ is calculated, we can now evaluate the relationship between the hypothetical power consumption and the real power consumption by applying the correlation coefficient.

6.4.2 Attack on Hardware AES Encryption

DPA attacks targeting the hardware module are similar to attacks targeting the software implementation introduced in the previous section. However, it is often the case, that those cryptographic modules are already protected against differential power analysis attacks. Depending on the applied hardware countermeasures it is often sufficient to increase the number of recorded traces in order to reveal the secret key. Occasionally, the AES acceleration module on the chip is rather well secured and recording more traces will not leak the needed information to reveal the key. However in the worst case scenario the crypto module of the given chip is not secured at all. In this case the hardware acceleration should not be used where the algorithm is depending on the master key, because it is not possible to secure the crypto module anymore. The used countermeasures are often not well-documented for the given device, hence it is hard to know if the underlying AES module is well-secured or not. In order to achieve a high level of compatibility between the available microcontrollers and still benefit from the speed increase, achieved by using the hardware acceleration, a compromise has to be found. Every call to the AES encryp-

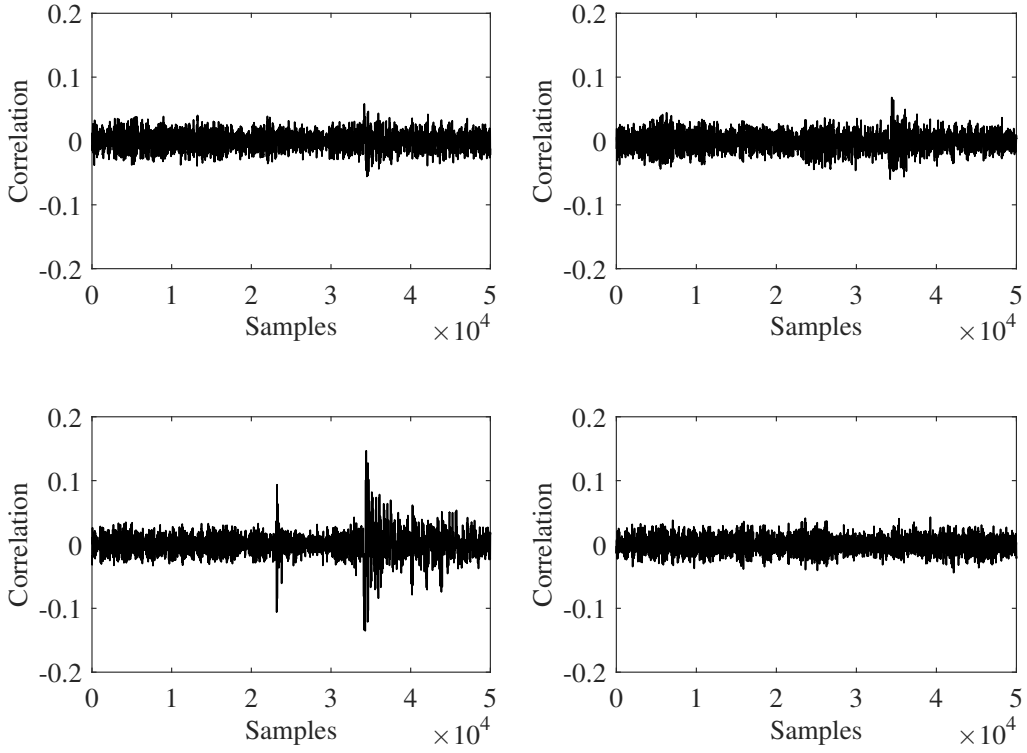

Figure 6.3: Correlation for the key hypotheses 10, 54, 90, 249

tion, where the master key is used and the premise for a DPA attack is given, should be executed by a software implementation, that can be secured by implementing additional countermeasures. All the other AES calls can be handled by the hardware acceleration of the device. This tradeoff will leak less information of the key while maintaining a well balanced compatibility throughout all the available MCUs.

6.4.3 Attack on iFeed

Recalling the Section 6.4.1 about DPA attacks on a software implementation, we know, that performing such an attack, requires finding a proper time interval, which happens to be the first round of AES at least if we know the input values. In this section we are going to analyze the iFeed algorithm and mark any possible location, which is vulnerable to a DPA attack targeting the first round of AES. First of all, we are going to analyze the processing of the associated data (AD). The procedure can be seen in Figure 6.4. It consists of two main parts, a mask calculation Z_0 and the actual AD processing, which results in the calculated tag T_A. Both parts are using the standalone AES encryption function with the master key of the device. Unfortunately, even though we know the input values of those two parts, both of them can't be used for an attack. The first part, calculating the mask Z_0, is done by encrypting the value 0. Hence, the requirement that the input value should be non-constant is not given.
The real AD processing on the other hand would be a good location to perform the attack, because we know the input data. The AD value is non-constant and can be of arbitrary length. Unfortunately, right before the encryption process of the AD starts, the value is being XOR'ed with a calculated mask Z_i for $i = 3 \ldots len(AD) + 1$, which we do not know,

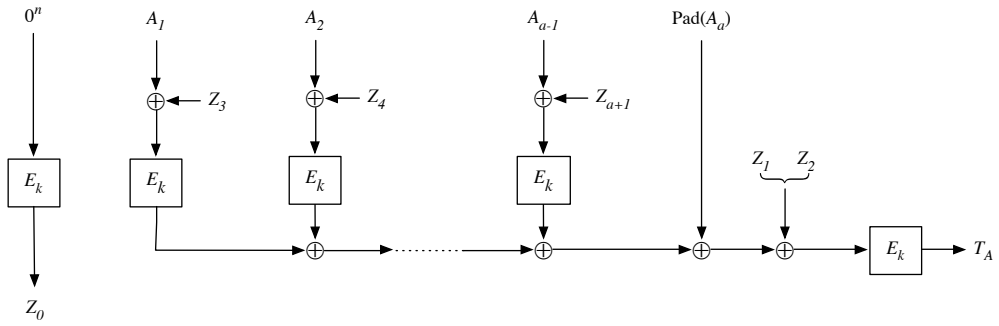

Figure 6.4: iFeed AD processing

because it is being derived from Z_0. Due to this fact, the AD processing part of the iFeed algorithm is eliminated as a candidate for the DPA attack as well.
These observations reduce the possibilities for an attack to the plaintext processing part of the algorithm, which can be seen in Figure 6.5. In contrast to the AD processing it consists of three different parts. The first part is needed to calculate a secret value U, which is furthermore needed to calculate the mask Z'. The last two parts are responsible for calculating the ciphertext C. Processing the plaintext yields the same problem as we had with the AD processing. The masking value Z'_i, derived from U and added to the input right before the encryption, is not known.

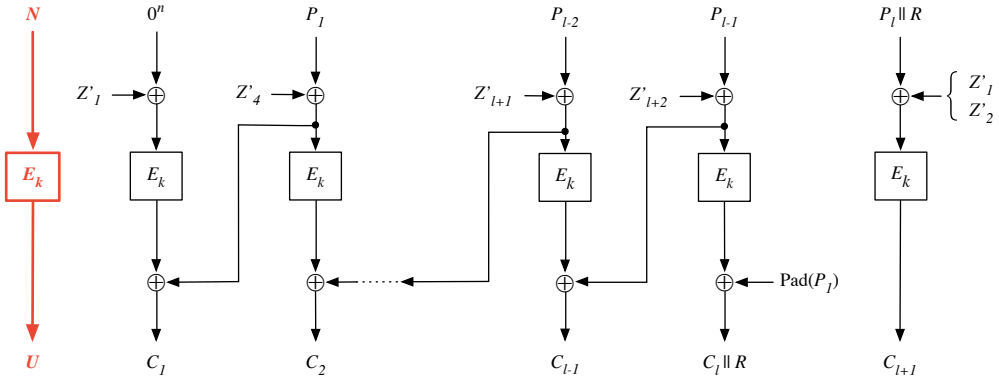

Figure 6.5: iFeed plaintext processing with the vulnerable location marked in red.

However, having a closer look on the calculation of U, it can be seen, that this is actually a very well-suited place to mount a DPA attack. The input for the encryption is N, which is by definition a public known nonce. As we already know from chapter 5.2, the nonce N is a public chosen, non-constant value with a size of 1 to 15 bytes. This value is now being padded with a constant value, in order to get the needed size of 16 bytes. This fact will lead to a partial leakage of the non-padded bytes of the secret key, depending on the length of N. The padded bytes are constant values and are leaking nothing at all.

In our case we chose to use the preferred settings for the nonce length (12 bytes), as discussed in [40]. Due to this fact, we will be able to retrieve 12 bytes of the secret key and the remaining four bytes have to be brute-forced.

6.4.4 Attack on SILC

The SILC algorithm is working different in comparison to iFeed, which can be seen in Figure 6.6 and in Figure 6.7. The AD is being processed first, because the output V is later needed in order to generate the ciphertext C. However, we still need to find a location in the structure of the algorithm, where a DPA attack is possible. By analyzing Figure 6.6, we can see, that each block is dependent on the previous one, whose output is being XOR'ed to the new block right before the encryption E_k.

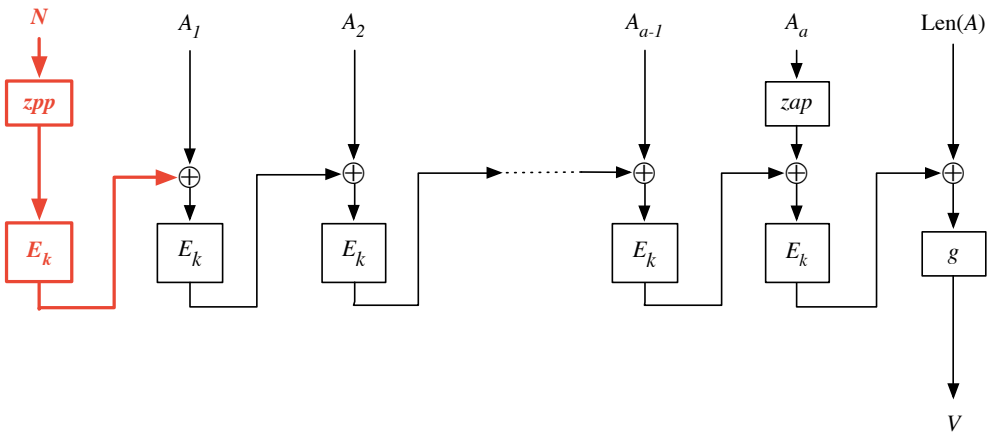

Figure 6.6: SILC AD processing with the vulnerable location marked in red.

However, right before the associated data is even processed, an initial vector (IV) is being calculated by first applying a zero-prepending function *zpp* on the public nonce N and second performing an AES encryption with the master key. The *zpp* function will take an input of arbitrary length and will yield a 16-byte value padded with a given constant. This IV calculation is actually pretty similar to the calculation of U in the iFeed algorithm, except that the constant value is now prepended to the input data. Due to this fact, it actually looks like a promising and well-suited place to mount a DPA attack. The nonce N is a 12-byte, public known, non-constant value. Due to the similarity to the iFeed algorithm, it has the same drawback. Revealing the full secret key will not be possible, because the first bytes are padded with a constant value. We have no alternative but to brute-force the last remaining bytes.
Additionally to the processing of AD, the analysis of the plaintext processing yields no proper location for a DPA attack. This is due the fact, that in order to know the input values, we need to know the output value of the AD processing V, which is unacquainted (Figure 6.7). Those facts and observations reduce the possibilities for an attack to exactly one place and that is when the IV for the AD processing part is being calculated.

6.4.5 Attack on OTR

OTR, as already described in Section 5.4, can be performed serial or parallel as being illustrated in Figure 6.8 and 6.9. The algorithm is, like iFeed and SILC, split up into two main parts, the plaintext processing and the associated data processing.

Starting with analysis of the parallel version of OTR (Figure 6.8) and considering the prerequisites needed for a DPA attack of the first AES round, it can be seen, that there

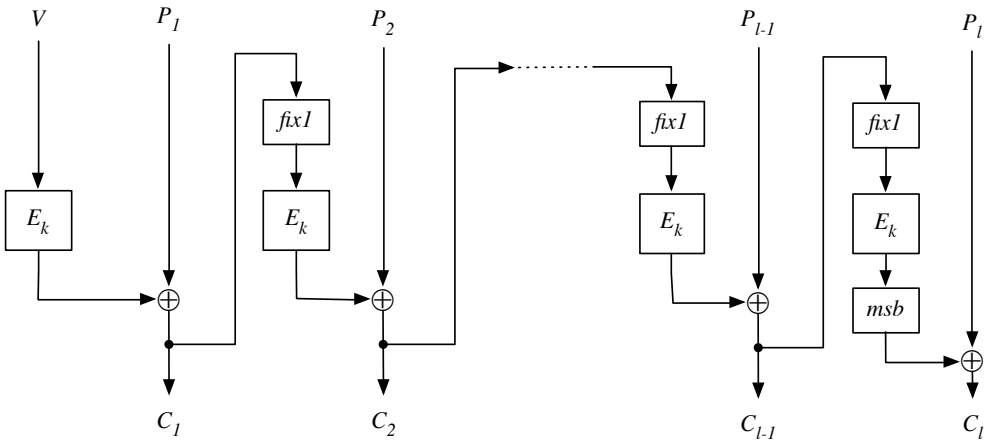

Figure 6.7: SILC Plaintext processing

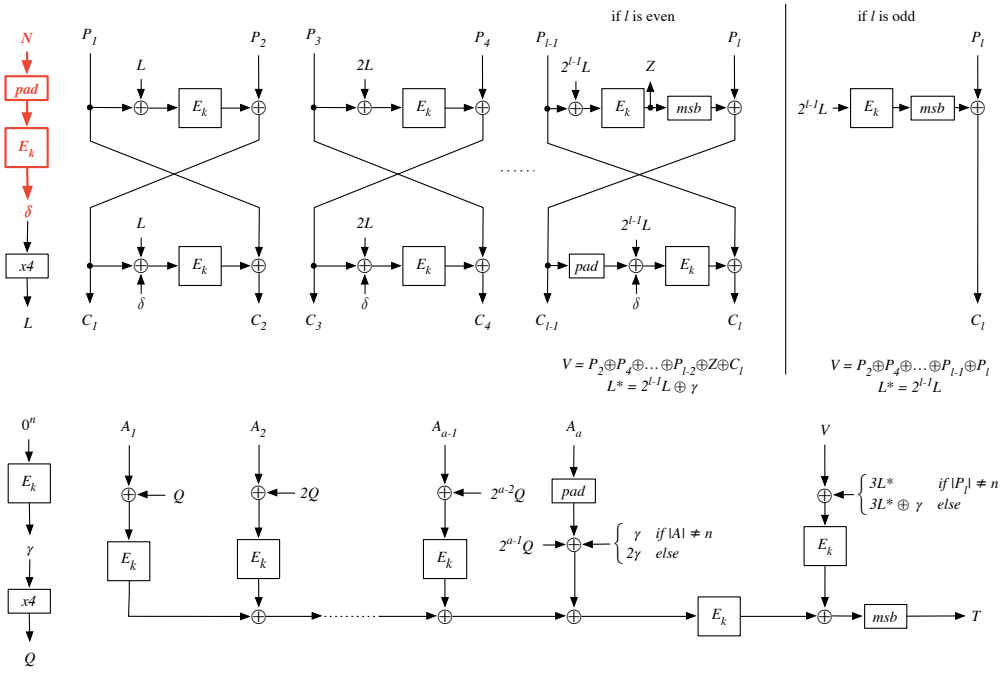

Figure 6.8: OTR with parallel plaintext and AD processing. The vulnerable location is marked in red.

is only one location in the algorithm where the attack can be mounted, the calculation of L. This computation is similar to the mask computation in iFeed and SILC. To get the value L, the public, 12-byte nonce is first padded to retrieve a 16-byte value and is then encrypted with the master key. The value L and its derivations are now being used in all plaintext processing parts and therefore kicks out the plaintext processing as a potential location for an attack. This is analog to the AD processing part, with the difference that the mask generation Q is performed with a constant value, meaning that there will be no leakage at all.

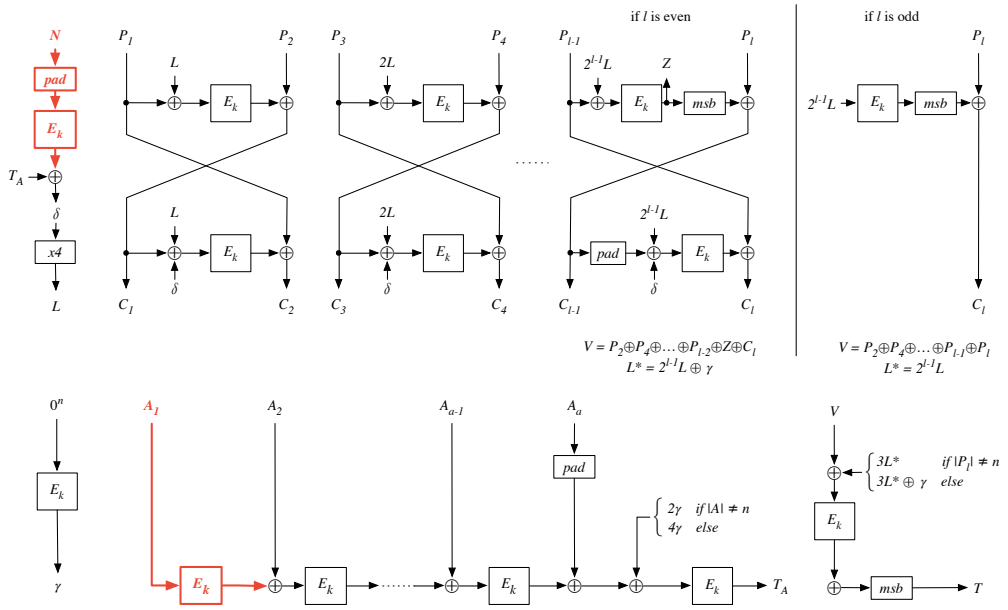

Figure 6.9: OTR with serial plaintext and AD processing with the vulnerable locations marked in red.

The same attack can be applied on the serial version of OTR, as it can be seen in Figure 6.9, with the difference that now the computed tag T_A is now being added right after the encryption process. Beside that, comparing the AD data processing part of the two OTR versions, it is noticeable, that the process is slightly different. While the parallel part is using a value Q in order to mask the associated data, this step is removed on the serial execution of the algorithm. The results of each encrypted AD part are simply being XOR'ed (Figure 6.9). This will eventually lead to an attackable location, due to the fact that the associated data can be freely chosen by the attacker. In fact, this location is even better than the attack on the nonce N. Setting the right length of the associated data, it will eventually lead to a full key recovery, in contrast to the previous attack, where a maximum of 12 bytes can be revealed. The length of the AD should be at least 16 bytes, this will ensure that the algorithm will execute the first iteration of the AD processing $E_K(A_1)$ without any additional padding. This observation can now be misused as an intermediate result for the DPA attack and therefore used to reveal the secret key.

6.4.6 Attack on COPA

The last algorithm which is being analyzed is COPA. The location for the attack is not that obvious as it was with the previous algorithms. First, let's have a closer look on the plaintext processing. It is noticeable that in order to process the plaintext a mask $L = E_k(0^n)$ has to be generated, which is then being XOR'ed in various modifications to the input data right before the encryption. For the first byte of the plaintext block P_1, L is being multiplied with $x+1$ (3 in integer representation) over a GF(2) with an irreducible polynomial $f(x) = x^{128} + x^7 + x^2 + x + 1$. This fact that the mask L and its modifications are being used right before any encryption, regardless of AD or plaintext processing, it is not possible to find out the master key with just one simple DPA attack. However, let's have a closer look on the first iteration of the plaintext processing and assume we

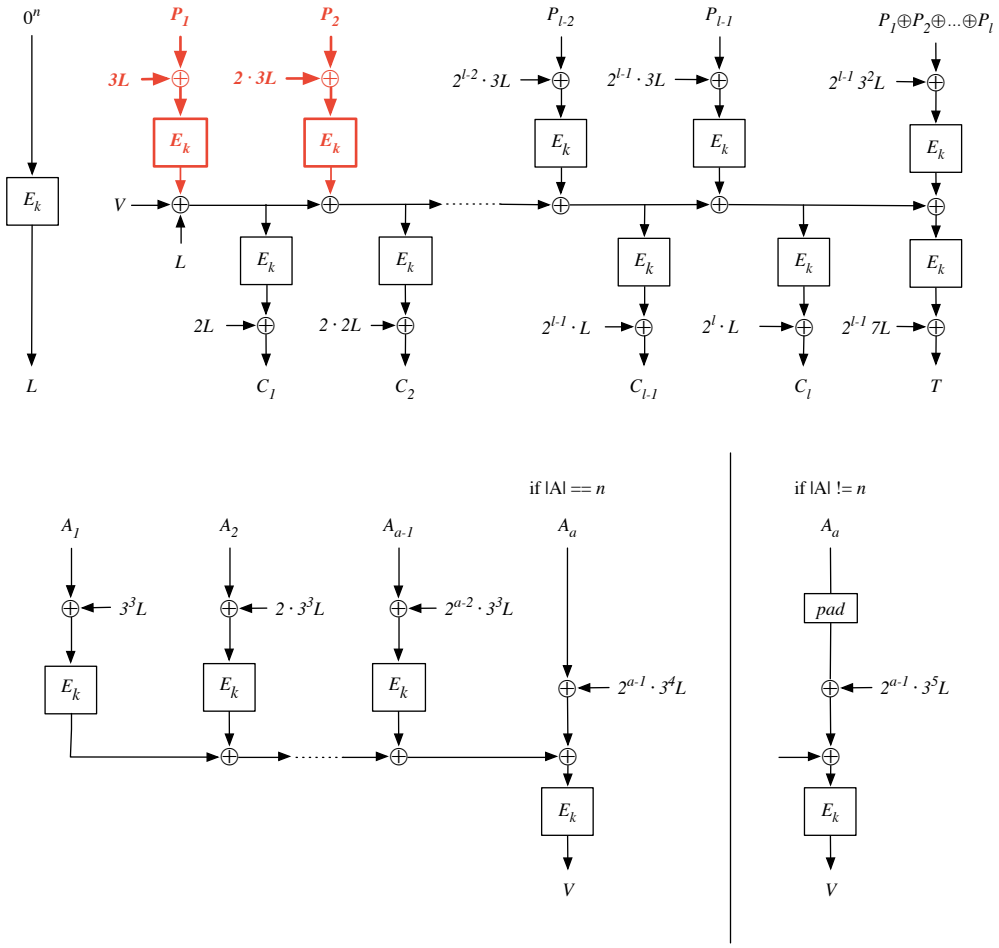

Figure 6.10: COPA plaintext and AD processing. The attacked location is marked in red.

cannot reveal the mask L. We know the intermediate value $x_1 = P_1 \oplus 3L$ is the input for the AES encryption. The first operation, which is done right before the first round of AES will start, is adding the cipher key k to the input data $x_1 k = x_1 \oplus k$. Furthermore $x_1 k$ is now being processed in the SubBytes operations $sx_1 = S(x_1 k)$. This means that $sx_1 = S(x_1 k) = S(x_1 \oplus k) = S(P_1 \oplus 3L \oplus k)$. If we assume there is no masking right before the encryption, the formula to calculate sx_1 would slightly change to $sx_1 = S(P_1 \oplus k)$. Attacking the latter one will eventually reveal the secret key k. This observation can be applied on the first formula containing the mask. A successful DPA attack on the first round of the plaintext processing of the algorithm will not reveal the secret key k, but a modification $k_1^* = 3L \oplus k$. Unfortunately this value k_1^* on its own will not be sufficient to reveal the secret, because it is masked. Therefore we mount the same attack on the second plaintext block P_2, which will result in $sx_2 = S(x_2 k) = S(x_2 \oplus k) = S(P_2 \oplus (2 \cdot 3L \oplus k))$. We can see that for the second iteration, we get another modified key $k_2^* = 2 \cdot 3L \oplus k$.

$$
\begin{aligned}
k_1^* = 3L \oplus k &\implies k = 3L \oplus k_1^* \\
k_2^* = 2 \cdot 3L \oplus k &\implies k = 6L \oplus k_2^* \\
3L \oplus k_1^* &= 6L \oplus k_2^* \\
k_1^* \oplus k_2^* &= 3L \oplus 6L \\
k_1^* \oplus k_2^* &= L(3 \oplus 6) \\
k_1^* \oplus k_2^* &= 5L \\
L &= (k_1^* \oplus k_2^*)^{-5}
\end{aligned}
\tag{6.2}
$$

The values k_1^* and k_2^* and the fact that L and its modifications are multiplications in a finite field we can use simple algebra to solve an equation with one unknown variable, to reveal the original value $L = (k_1^* \oplus k_2^*)^{-5}$ as being illustrated in Equation 6.2. Acquiring L is the important step to retrieve the secret key. The knowledge of L leads to $3L$ by simply multiplying L with $x+1$. Considering the first iteration of the plaintext, the attack yielded a modified key $k_1^* = 3L \oplus k$. In order to unmask the key k, it is sufficient to add ($\oplus$) the previously calculated mask $3L$ to k_1^*. This will remove the mask and reveal the master key k.

6.4.7 Complexity of each Attack

The complexity of iFeed and SILC are actually the same in the used setup, both performing its encryption with a 12-byte nonce. As in Section 6.4.3 and Section 6.4.4 described, it is not possible to reveal the full key due to the constant padding. Still, retrieving 12 bytes of the secret key, reduces the key space of those two algorithms from 128 bit to 32 bit. It can be decreased even more, if the chosen nonce length is bigger than 12 bytes. The remaining bits of the key can be brute-forced in a conceivable amount of time, depending on the used machine.

The complexity of OTR depends on the used version. In case the parallel processing version of the algorithm is deployed on the MCU, the resulting complexity is the same as for iFeed or SILC. This, however, changes with the serial version of the algorithm. There are two locations to mount a DPA attack, both yielding different complexity. Attacking the nonce will eventually lead to the same result as for the parallel version. More interesting is the part where the attack is being performed on the associated data. The AD data can be fully controlled in matters of size and value, hence the full 128 bit key can be recovered.

Considering the fact that revealing the secret key of COPA, needs two independent DPA attacks, it is obvious that the complexity of this algorithm is doubled. However, as it was introduced in the previous section, we are using the plaintext input as data for the attack, hence we have full control over the input data. This will sooner or later lead to a full key recovery, with the drawback, that we need to perform two different DPA attacks.

Having analyzed all four algorithms and their weaknesses for DPA, it is noticeable, that all four AE algorithms, if not well-secured, will partially or even fully leak the secret key.

Chapter 7

Countermeasures

The previous chapter showed that it is possible to reveal the secret key of all the selected AE algorithms by applying DPA attacks. This chapter will discuss the possibilities to secure those algorithms by applying countermeasures. Since first DPA attacks were published by Kocher, Jaffe, Jun [26], various ways to harden an implementation against DPA attacks have been introduced. Inserting dummy instructions [8], randomization of operations [35], or masking of the data [7, 14] are common ways to secure implementations. This chapter will give a general description on securing an AES implementation by applying a masking scheme. To be more precise, we will apply the more sophisticated version of masking as introduced in [25].

7.1 General Countermeasures Counteracting DPA Attacks

DPA attacks work, because the power consumption depends on the sensitive intermediate values used during encryption. The goal of a countermeasure is to reduce this dependency to a minimum. In order to accomplish this, the sensitive intermediate values are being hidden. Hiding causes the dependency between the sensitive intermediate value and the power consumption to be dissolved. The most common ways are inserting dummy instructions or randomizing operations. In Section 6.4 it is stated that the measured power traces should be aligned to produce good results. If this premise is not given the attacker needs significantly more power traces. Due to this fact, some countermeasures try to create random power traces. The more random, the harder it is for an attacker to retrieve the secret information. One way to randomize the power traces would be by inserting dummy instructions. The basic idea is inserting dummy instructions before, during, and after the execution of the algorithm. The amount of dummy instructions used throughout the execution is decided by a random value, calculated right before the encryption starts. However, it is important that the overall count of inserted dummy instruction is equal for each execution, otherwise the attacker would be able to gain information about the number of inserted instructions by measuring the execution time. Although, it is easy to implement, the increased count of instructions, due to the insertion of dummy instruction, has an impact on the throughput. The more dummy instructions are inserted, the less bytes per cycle can be processed. The second approach to randomize the power trace would be shuffling the instructions. In contrast to inserting dummy instructions, no operations are inserted, hence the throughput is not that much affected. The idea is to shuffle certain operations in the algorithm, that can be performed in arbitrary order, hence not

being dependent on each other. In the case of AES, this would be the S-Box lookup. Each round 16 bytes have to be substituted by an S-Box lookup. However, each byte is independent of the other 15 bytes. Therefore the sequence of lookups can be randomized. The downside of shuffling is, that it is limited to a certain operation, where the consecutive operations are not dependent of each other. Both schemes, shuffling and insertion of dummy instructions, are often combined in practice. However, next to inserting dummy instructions or randomizing operations, there is a third option. Breaking the linkage between the measured power consumption and its intermediate values can be achieved by blinding the intermediate value. The basic idea of blinding is to add a random value to the intermediate value, hence randomizing the intermediate value. This way, even if the device has a a data-dependent power consumption, it will leak the masked intermediate value, which is independent of the original value. This approach will be discussed in more detail in the following section.

7.2 Securing AES with Boolean Masking

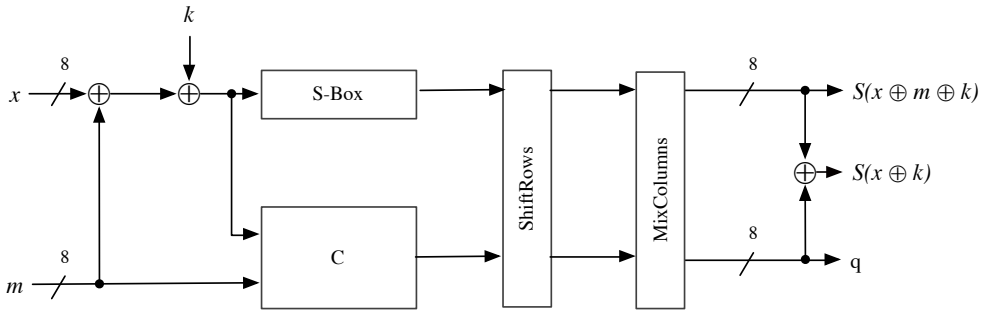

Figure 7.1: Boolean Masking of an AES Round for one of the 16 Key Bytes

One common way to counteract DPA attacks is by masking the sensitive intermediate values. A random and independent variable m is calculated and added to the plaintext (x), which is denoted by $m \oplus x$. Figure 7.1 illustrates this method for one AES round. A random value m, being independent from input x, is chosen and added ($\oplus$) to x, which is now used for the AES round. The according round key k is added and substituted by the corresponding S-Box value. Due to the fact, that the S-Box operation is a non-linear function ($S(x \oplus k) \neq S(x) \oplus S(k)$) inverting the masking process is more complex. A correction function C has to be implemented, taking both m and $x \oplus m \oplus k$ as input in order to produce an output mask q such that $S(x \oplus m \oplus k) = S(x \oplus k) \oplus q$. The values q and $S(x \oplus m \oplus k)$ are now both shifted in the ShiftRows operation and the MixColumns operation is performed on both values either. ShiftRows and MixColumns are both linear operation, denoted as $f(x \oplus k) = f(x) \oplus f(k)$, therefore no special handling of the mask q has to performed. In order to calculate the unmasked value, it is enough to XOR q and the masked value. However, targeting more sensitive variables in the algorithm by applying a high-order attack an attacker is still capable of retrieving the secret information. In [31] it was shown that a second-order attack combining $S(x \oplus k \oplus m)$ and the mask m is sufficient to reveal the key. In 2013 a new countermeasure being theoretically secure against DPA attacks of any order was presented [25].

7.3 Masking AES with Randomized Lookup Tables

In [25] a countermeasure has been introduced using randomized lookup tables (RLUT). This countermeasure is theoretically secure against DPA attacks of all orders. The idea is to add more random data during computation. Adding more randomization is done in such a way that the lookup tables G_1, G_2, R, RC are pre-computed based on a master key k and random data a_1, a_2, a_3 in a secure environment. As each of the pre-computations involves a random variable a_i, independent of the others, combining the leakage of the look-up tables should not reveal any secret information to the attacker.

Algorithm 1 Table Pre-Computation

– **input**: P_k
1: Generate random variables a_1, a_2, a_3
2: $\forall i, j : G_1(i, j) = i \oplus j \oplus a_1$
3: $\forall i : R(i) = P_k(i) \oplus a_2$
4: $\forall i, j : RC(i, j) = R(i) \oplus P_k(i \oplus j \oplus a_1) \oplus a_3$
5: $\forall i, j : G_2(i, j) = R(i) \oplus j \oplus a_3$
– **output**: G_1, G_2, R, RC

The initial process of the pre-computation needs the generation of three random variables a_1, a_2, a_3 and a table P_k, which is supposed to be performed in a secure environment. The table P_k is a lookup table replacing the KeyAddition and S-Box lookup. The idea is to iterate through all possible input bytes, add the constant master key byte k, perform a S-Box substitution and save the results in the table P_k. In other words $P_k(x) = S(x \oplus k)$, x denoting the current plaintext byte. The masks a_1, a_2, a_3 are then used to pre-compute the lookup tables as illustrated in Algorithm 1. In order to perform the encryption, the values x, denoted as the plaintext, and the initial mask m are now being used to lookup the required values in the according tables as being illustrated in Algorithm 2.

Algorithm 2 operation

– **input**: G_1, R, RC, x
1: Generate random mask m
2: Compute $G_1(x, m) = x \oplus m \oplus a_1$
3: Compute $R(G_1(x, m)) = P_k(G_1(x, m)) \oplus a_2$
4: Compute $RC(G_1(x, m), m) = R(G_1(x, m)) \oplus P_k(G_1(x, m) \oplus m \oplus a_1) \oplus a_3$
– **output**: $R(G_1(x, m)), RC(G_1(x, m))$

The tables G_1 and G_2 are never explicitly used during encryption. G_1 is used to mask the plaintext x and G_2 is used at the end of the encryption round to unmask the intermediate value as illustrated in Figure 7.2.

7.4 Implementation of the Randomized Lookup Tables

A simplified and reduced version of LED cipher [15] has been implemented as an example to show the above discussed scheme. The reduced version of LED consists only of four rounds and a 16 bit data width in contrast to the full version of LED, which has a 64 bit data width and has 32 rounds in order to process the full 128 bit key. However, the algorithms in this thesis are all AES-based, therefore we reimplement the aforementioned

scheme to work for the AES-128 encryption on the MSP430 FR5969. Recalling the last section, we know, that in order to pre-compute the tables we need three random numbers. The MCU does not provide any pseudo random number generator (PRNG), so we had to implement a PRNG first. In the paper [25] the PRNG was implemented by using a linear feedback shift register (LFSR) in combination with a CRC-32 polynomial in order to illustrate the countermeasure.

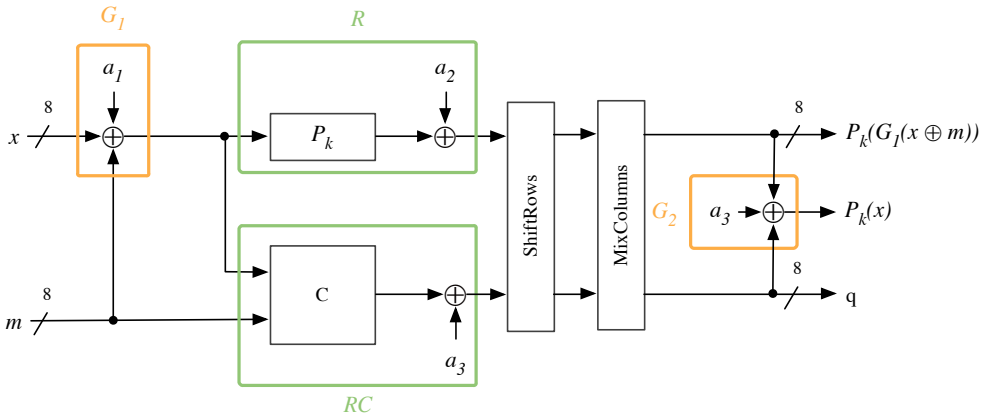

Figure 7.2: Masking with Randomized Lookup Table. G_1 and G_2 are used to mask and unmask the value x, while R and RC represent the randomized lookup tables

In our case we slightly modified this version and swapped the CRC-32 algorithm with the AES-128 as pseudo random function (PRF). A 16 bit Fibonacci LFSR is serving the input to the AES encryption. Alternate ways of generating randomness can be considered. The detailed pseudo random generation works as follows:
Every MSP430 FR5969 has a random read-only device seed s, which can be readout. This seed s is the input for the 16 bit LFSR to generate a device-unique 128 bit key PRG_k. The LFSR is executed eight times with s being the initial seed, appending each output to the result PRG_k, which is later on used as encryption key. In order to generate random input for the AES encryption, we implement a PRNG discussed in the official paper [22]. The implementation utilizes the clock system and its varying clock frequency on the MCU. Two timers on the controller are set up to use different clock sources (e.g. SMCLK and ACLK) with different clock frequency. Each timer increments a counter value by 1 at each clock cycle. One of the two timers generates an interrupt as soon as the counter reaches a specific value. The delta between the two timers is now used as random number r. Due to the fact that the clock frequencies of the timers are not constant, but deviates from the source clock, the difference between the two timers cannot be predicted and differs for each subsequent call. This deviation from the presumed clock frequency is called clock jitter. Jitter is usually an undesired factor in the design and causes an random phase modulation of the source clock. The higher the jitter is, the higher the variance of the clock frequency will be. This circumstance helps us to produce unpredictable deltas between two different timers. Having the values PRG_k and r we can now generate a random number. We use r as the input for the LFSR and generate a 16-byte random value r'. Furthermore, r' is now encrypted ($AES_{PRG_k}(r')$), resulting in a 128 bit value, which can be shortened to the needed size. In order to perform Algorithm 1, we need four random variables, the mask m

and three random values a_1, a_2, a_3, each having the size of one byte and a pre-computed table P_k replacing the AddRoundKey and SubBytes operation. The tables G_1, R, RC, G_2 can now be pre-computed. However, calculation needs to be performed for all possible values, because the plaintext values are not known at time of pre-computation. This will lead to 2^8 different values for each key byte k. However, each key has a size of 16 bytes and additionally pre-computation should be done for each round of AES-128. This will eventually lead to a table of the size $10 \cdot 2^{12}$ bytes or 40 KB. Recalling the previous chapter, there are four tables of this size R, RC, G_2, and P_k. Four tables each of the size of 40 KB results in an approximately overall RAM usage of 160 KB. The used MCU is equipped with 8 KB RAM, which is not enough space to keep the tables in RAM. However, the MSP430 FR5969 is equipped with FRAM. FRAM has fast reading and writing times, this way storing the pre-computed tables in FRAM would be an option. However, the overall space of the FRAM memory is 64 KB. In order to reduce the needed memory usage of the pre-computed table, a tradeoff has to be made.

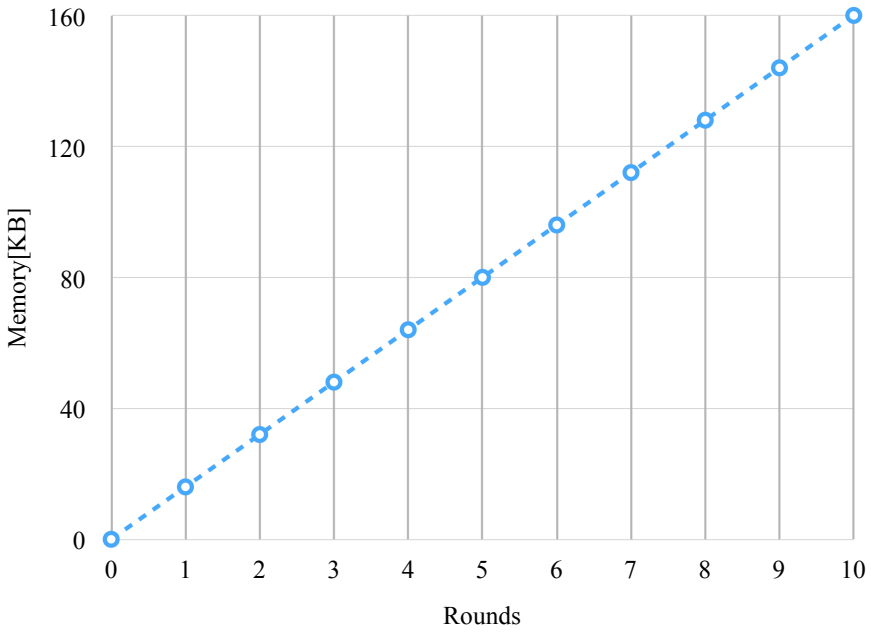

Figure 7.3: Memory needed to implement the RLUT countermeasure. The needed memory increases linearly depending on the amount of secured rounds.

In Figure 7.3 it can be seen, that the needed memory usage increases linearly to the amount of rounds being secured. Due to the fact, that the used FRAM is a unified memory, the program code and data is being stored altogether on the FRAM. This reduces the available space on the memory. Depending on the implemented AE algorithm roughly 40 KB memory can be used. Hence, we implement the countermeasure in such a way that the first round and the last round of the AES are secured. This way pre-computation is done for two rounds unlike previous computation, resulting in $2 \cdot 2^{12} = 8$KB memory usage per table. Because of this tremendous memory saving, we are able to implement this countermeasure on the given MCU. However, all pre-computed tables (P_k, R, RC, G_1, G_2)

are stored on the FRAM memory. Due to the fact that our MCU is running on 16 MHz and the FRAM memory is not capable handling access faster than 8 MHz, a wait state is required, eventually leading to a slower access, compared to SRAM.

Chapter 8

Results

The following chapter gives a detailed description of the benchmarks. Furthermore, the DPA attacks, targeting the AES implementation, performed on the MSP430 FR5969 based on Chapter 6 will be discussed. We will discuss the impact of applied countermeasures in matters of performance loss and security gain. All of the selected algorithms are benchmarked with the same initial setup referring to their preferred input values. Furthermore, we will introduce the results of the DPA attacks on the selected algorithms. Measuring the power traces was done in two different ways. First, an EM probe was used to measure the EM emanation above a capacitor in close proximity to the supply-voltage pin. Second, power traces were measured by inserting a shunt resistor into the supply-voltage path.

Each setup needed an amplifier, due to the fact, that the power consumption of this microcontroller is very low. Recording a very low power trace, will lead to the fact, that the resolution of the given trace is too small and we will need significantly more power traces to successfully mount a DPA attack. However, amplifying a signal will cause a higher noise in the power trace. Hence, a tradeoff between small resolution and increased noise has to be made. Eventually leading to the observation, that amplifying the signal, will give better results even with an increased noise in the recorded power trace. We are able to mount successful attacks on all algorithms. Results show, that collecting 2 000 traces is sufficient to reveal a sufficient amount of bytes of the secret key, if no countermeasure is in place. We will discuss the impact of the countermeasure introduced in Section 7.3 on the security of the attacked algorithms and show that simply increasing the count of recorded traces will not be sufficient in order to reveal the secret key.

8.1 Benchmark

The main focus of this section is to present a detailed description of the performance of each AE algorithm implemented on the MSP430 architecture. In order to do that, we will oppose the four algorithms to each other and furthermore compare the result to related work. This way it can be shown, if the given algorithms are performing better or worse compared to already existent and used algorithms on a microcontroller platform. All the results are summarized and visualized in Figure 8.1.

8.1.1 Initial Condition and Overall Settings

We are using the same setup for all the AE algorithms to produce comparable results. All selected AE algorithms are implemented in such a way, that each of them start their

encryption with the same function call *crypto_aead_encrypt(uint8_t* out, uint32_t* out_len, uint8_t* in, uint32_t in_len, uint8_t* ad, uint32_t ad_len, uint8_t* npub, uint32_t npub_len, uint8_t* k)*. The parameters are as follows:

- *uint8_t* out*
  The first parameter will point to the encrypted ciphertext.
- *uint8_t* out_len*
  This pointer will hold a reference to the output length of the parameter *out*.
- *uint8_t* in*
  This parameter will pass the plaintext to be encrypted.
- *uint32_t in_len*
  Additionally to the parameter *in*, the length of the given plaintext has to be passed as well.
- *uint8_t* ad*
  If there is additional data available, it will be referenced by this parameter.
- *uint32_t ad_len*
  This parameter will either be 0 if no AD is given or will hold the length of the passed AD.
- *uint8_t* npub*
  The parameter *npub* refers to the public message number of the algorithm.
- *uint32_t npub_len*
  This parameter will hold the length of passed public message number array.
- *uint8_t* k*
  The last parameter k is the key used for encryption.

Furthermore, a trigger is set right before this function call and unset right after this function returns. Due to this preparation, it can be assured, that any pre-processing of the input data, which needs to be done to start the encryption process, is not included in the benchmark results. Furthermore, measuring the execution time is performed using an oscilloscope. The area in which the trigger is set, equals the measured timespan. It should be noted, that all algorithms are using the assembly version of the AES provided by IAIK, as the used software version. Critical parts in the algorithms have already been replaced by this version.

All four AE algorithms are using their preferred settings in terms of input and output data stated in their official paper [9, 24, 32, 40] and already discussed in more detail in Chapter 5. The message data used during the benchmark is pre-set to fixed block sizes, which would be 2, 16, 17, 32, 48, 63, or 64 bytes. The associated data is set to fixed lengths as well. That would be either no header data at all (0 bytes), or 1, 16, 31, or 32 bytes. We are not using larger messages due to the assumption that the messages transmitted on an embedded system will be preferably small.

8.1.2 Comparison of the AES Versions

Each of the implemented AE algorithms are using the unmodified version of AES as underlying encryption function. A detailed description will be given how the different AES versions used throughout this master thesis are performing to each other. The underlying AES encryption (E_k) is executed more than once for each of the AE algorithms. Due to this fact, the overall speed of the algorithms are partly dependent on the speed of E_k. As seen in Figure 5.1 and discussed in Chapter 5.1 we know, that the ASM version is by far the best way to go in terms of software AES, because it is 3.28 times faster than the version provided by Texas Instruments (TI) and already optimized for speed and memory usage.
However by applying the countermeasure introduced in Section 7.3, the performance loss is immense. Execution of one single AES encryption with the ASM version, needs 91.6 ms to finish. This results in a 229 times slower computation compared to the unsecured version, which requires 0.4 ms to execute. This is mainly due to the fact, that for a proper usage of the countermeasure, different tables have to be pre-computed, before the AES encryption can be started. Pre-computation time is independent of the used key and message, therefore we can say, that by securing the AE algorithms with the countermeasure, we increase their execution time by adding a constant value c. Due to the fact, that c is constant, its impact on the overall execution time, will be smaller, the bigger the input data is, because not all AES executions need to be secured.

AD	Message							
bytes	bytes							
	2	16	17	32	33	48	63	64
0	0.77	0.86	1.04	1.14	1.32	1.42	1.70	1.70
1	0.85	0.95	1.13	1.23	1.41	1.51	1.78	1.79
16	0.91	1.00	1.18	1.28	1.46	1.56	1.83	1.84
31	1.04	1.13	1.31	1.40	1.59	1.69	1.96	1.97
32	1.05	1.14	1.32	1.42	1.60	1.70	1.97	1.98

Table 8.1: Execution time of SILC in ms

8.1.3 Benchmark Results of Unsecured/Secured AE Algorithms

Each of the selected algorithms has been measured for all combinations of the fixed input data and the results are different for each of the algorithms. The benchmark was first executed on all the algorithms without any implemented countermeasure. In Table 8.1 it can be seen, that SILC requires 0.77 ms with no associated data and a 2 byte message and requires 1.98 ms with 32 bytes associated data (AD) and a message with the size of 64 bytes. Those two data sets represent the smallest and biggest data sets in the benchmark. Having a closer look on the aligned message block sizes (16, 32, 48, and 64) it can be seen, that increasing the message size by 16 bytes increases the runtime by 0.28 ms (horizontal row in Table 8.1). For instance the execution time for AD with the size of 16 bytes and a 16 byte message needs exactly 1.00 ms, while encrypting 16 bytes more, requires 1.28 ms. However, the same scheme can be found if the header size (AD) is being increased. The delta between the aligned header sizes (0 bytes, 16 bytes, and 32 bytes) is 0.14 ms. (illustrated by the columns in Table 8.1). This leads to a steady increase of the runtime

depending on the size of the input data. To be more precise this is a linear increase of 1.75 μs per byte in case of a message increase and 0.88 μs in case of increasing size of the header data.
OTR is the only one of our selected algorithms capable of running in two modes, parallel and serial. Both versions result in different runtime behavior. As introduced in Chapter 5.4, we know that the difference between the parallel and serial version is actually the way the associated data is being processed. The parallel version allows us to process AD data in parallel, however includes a few more calculations. This behavior is reflected by Table 8.2 and Table 8.3. The performance is nearly the same on the first row, where no AD data is given at all. Two bytes message length will result in an execution time of 1.01 ms on both versions, while generating the ciphertext for a message with a size of 64 bytes, the encryption lasts for 1.97 ms for the parallel version and 1.98 ms for the serial version.

AD	Message							
bytes	bytes							
	2	16	17	32	33	48	63	64
0	1.01	1.12	1.33	1.44	1.54	1.65	1.92	1.97
1	1.23	1.39	1.61	1.72	1.81	1.92	2.19	2.25
16	1.29	1.40	1.61	1.73	1.82	1.93	2.20	2.25
31	1.54	1.64	1.86	1.97	2.06	2.17	2.45	2.50
32	1.54	1.64	1.86	1.97	2.06	2.17	2.45	2.50

Table 8.2: Execution time of parallel OTR in ms

This significantly changes as soon as AD blocks have to be processed. The more AD blocks, the worse the runtime of the parallel version gets. For example, processing data with a message length of 64 bytes and AD size of 16 bytes requires 2.25 ms to finish on the parallel version and 2.20 ms on the serial OTR version. The difference in overall execution time is 0.05 ms. However, increasing the AD data by one 16 byte block, results in 2.50 ms overall execution time on the parallel OTR and 2.33 ms, if being executed on a microcontroller with the serial version of OTR. The difference is now 0.17 ms already. This is an increase of 0.12 ms, although we increased the input data by just 16 bytes. It should be noted, that the difference of 0.17 ms is independent of the used message length. Therefore it does not matter, if 2 bytes of message data have to be processed or 64 bytes. Increasing the AD data by one full block, will always raise the runtime of the algorithm by 0.17 ms. It can be observed, that the runtime of the algorithm does not significantly change if the algorithm has to process 1 byte of AD data or 16 bytes. This is due the fact, that the algorithm, is using blocks with a size of 16 bytes. Any AD block with a smaller size, will be padded to 16 bytes first (see Chapter 5.4 for further details) and used afterwards. As an example processing data with a message length of 16 bytes and 1 byte AD data will lead to 1.39 ms execution time on the parallel version of OTR and 1.33 ms on the serial version. However, increasing the AD block to 16 bytes, will result in a runtime increase of 0.01 ms in both cases, 1.40 ms, or 1.34 ms respectively.
The impact of aligned blocks (alignment always happens to 16 bytes) on the overall execution time, can be tremendous. iFeed and COPA have both a longer execution time if they have to cope with blocks not being aligned to 16 bytes, irrelevant if the block is used for AD or message data. As illustrated in Table 8.4, encrypting a 2 byte message with

AD bytes	Message bytes							
	2	16	17	32	33	48	63	64
0	1.01	1.12	1.34	1.45	1.54	1.65	1.92	1.98
1	1.23	1.33	1.55	1.66	1.75	1.92	2.14	2.19
16	1.24	1.34	1.56	1.67	1.76	1.93	2.14	2.20
31	1.37	1.47	1.69	1.80	1.89	2.17	2.27	2.33
32	1.37	1.47	1.69	1.80	1.89	2.17	2.27	2.33

Table 8.3: Execution time of serial OTR in ms

no header data requires 0.75 ms, while encrypting a message with the size of 16 bytes, takes 0.03 ms less, resulting in an overall execution time of 0.72 ms. This effect holds for various AD sizes as well. While encrypting data with a size of 2 bytes and 31 bytes of AD requires 1.47 ms to finish, increasing the AD by one single byte, results in a 0.08 ms faster execution. This is basically due to the fact, that before the last block is being processed, it has to be padded, which will not happen if the block has already a size of 16 bytes. Due to the padding function it is irrelevant if the unaligned block is off by one byte or 15 bytes. The execution time with a 64 byte message and 32 byte AD requires 1.73 ms.

AD bytes	Message bytes							
	2	16	17	32	33	48	63	64
0	0.75	0.72	0.96	0.94	1.18	1.16	1.48	1.37
1	0.92	0.89	1.13	1.11	1.35	1.33	1.65	1.54
16	0.88	0.86	1.10	1.08	1.31	1.29	1.61	1.51
31	1.18	1.15	1.39	1.37	1.61	1.59	1.90	1.81
32	1.11	1.08	1.32	1.30	1.54	1.52	1.83	1.73

Table 8.4: Execution time of iFeed in ms

The same behavior can be observed on COPA. However, it is even worse in this case. Using input data with no header data but with a message length of 2, or 16 bytes respectively, iFeed had a difference from 0.03 ms, caused by the padding function. However, COPA requires 1.16 ms for 2 bytes and 1.02 ms for 16 bytes. This is a difference of 0.14 ms. Benchmarking with the biggest data set (32 byte AD, 64 byte message length) a single encryption requires 2.02 ms, while using one byte less, increases the needed time to 2.36 ms. The same applies for the associated data size. Aligned AD blocks need less time for a single run. For example running the benchmark with 31 bytes of AD and a message with a size of 64 bytes, requires 2.11 ms. Increasing the AD block by just one byte, in order to align the block, the overall execution time is decreased by 0.09 ms to 2.02 ms. This significant difference of overall runtime between aligned and unaligned blocks, has two reasons: First, the same scheme applies as for iFeed. Unaligned blocks have to be aligned first by applying a padding. However, the second factor is, that the used gaulois multiplication has to be called once more, if the block is not aligned, resulting in a higher execution time.

Comparing any of the given algorithms to the reference algorithm AES-GCM, it can be seen, that no matter which algorithm is being used, all of them are significantly faster than the AES-GCM. The benchmark results for AES-GCM are illustrated in Table 8.6.

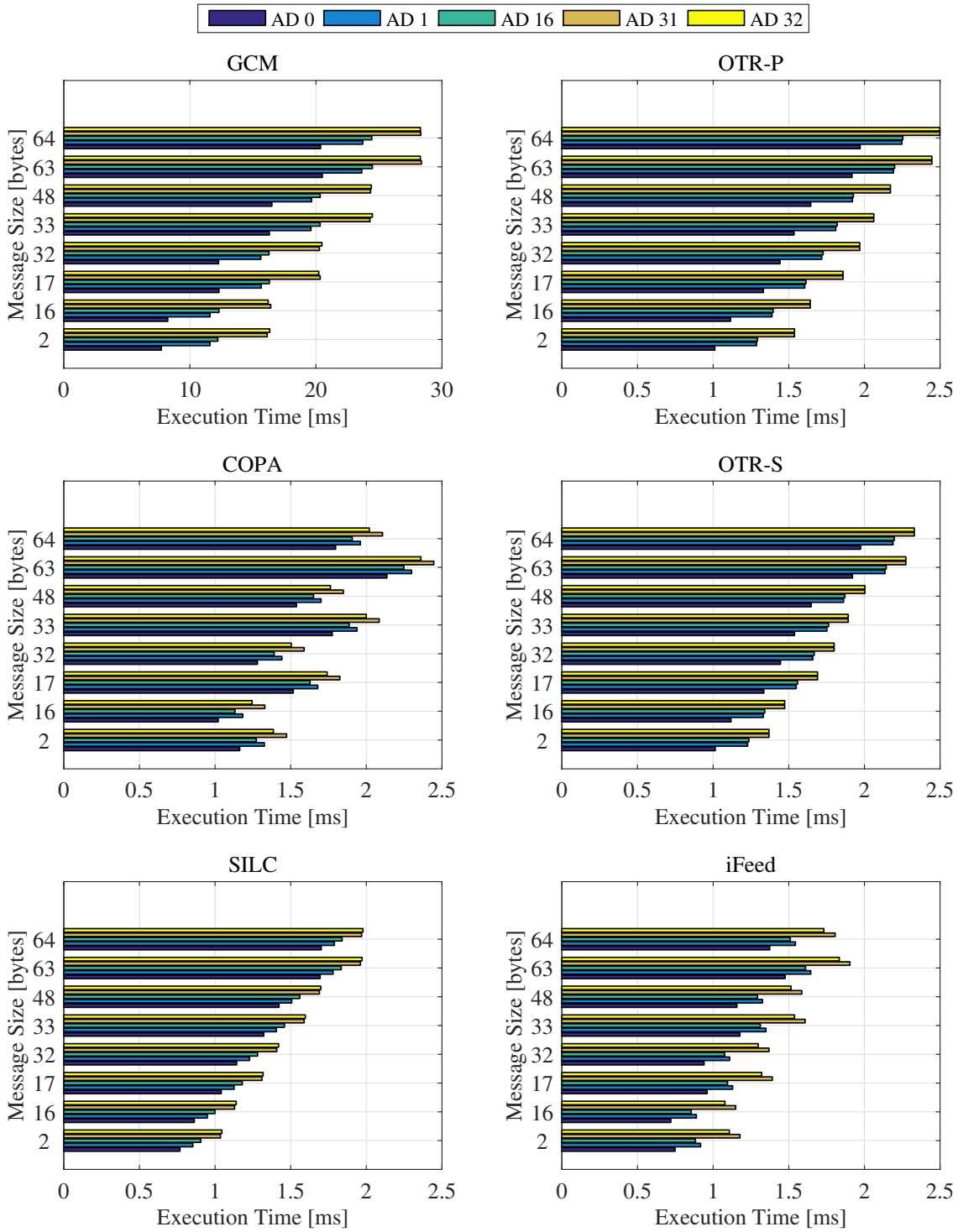

Figure 8.1: The runtime of all AE algorithms (GCM, OTR-P, COPA, OTR-S, SILC, iFeed) in descending runtime order. The runtime is denoted as ms. Header and message size are in bytes. It can be seen that COPA and iFeed are performing better if the given data set is aligned to 16 bytes, while all the others have a steady increase of the runtime

AD bytes	Message bytes							
	2	16	17	32	33	48	63	64
0	1.16	1.02	1.52	1.28	1.78	1.54	2.14	1.78
1	1.33	1.18	1.68	1.44	1.94	1.70	2.30	1.96
16	1.27	1.13	1.63	1.39	1.89	1.65	2.25	1.90
31	1.47	1.33	1.83	1.59	2.09	1.85	2.45	2.11
32	1.39	1.25	1.74	1.50	2.00	1.76	2.36	2.02

Table 8.5: Execution time of COPA in ms

For an example, encrypting a message with the smallest data set possible (no AD, 2 bytes message length) takes 7.74 ms to finish, while the slowest of the implemented algorithms, which would be COPA in this case, needed 1.16 ms for the same data set depicted in Table 8.5. This means that COPA is 6.67 times faster than GCM. Not to mention the fastest algorithm for this data set (iFeed). Execution requires 0.75 ms, resulting in a 10.32 times faster execution compared to AES-GCM. The execution time gets even worse the bigger the test data is. Running the test with 32 bytes of AD and 64 bytes of message data, GCM needed 28.29 ms in order to perform one single encryption. While the worst case of the implemented algorithms has a runtime increase of a maximum of 1.49 ms (OTR, parallel). Encrypting the same data set using AES-GCM requires 20.55 ms more time.

AD bytes	Message bytes							
	2	16	17	32	33	48	63	64
0	7.74	8.25	12.32	12.28	16.31	16.51	20.51	20.37
1	11.6	11.6	15.66	15.63	19.61	19.66	23.64	23.72
16	12.22	12.31	16.31	16.28	20.34	20.34	24.48	24.43
31	16.14	16.42	20.34	20.28	24.29	24.34	28.37	28.32
32	16.34	16.22	20.23	20.48	24.48	24.40	28.29	28.29

Table 8.6: Execution time of AES-GCM in ms

Figure 8.1 visualizes all the previously discussed benchmark results. The illustration is sorted in descending order in matters of runtime. As stated in Section 4.2.1 we chose on one hand two algorithms, which are optimized for embedded system (OTR, SILC) and on the other hand we picked two algorithms, which are not well suited for small devices (iFeed, COPA). First, it can be observed, that algorithms, which are not optimized for embedded systems, are performing better on data sets, which are aligned to 16 bytes. COPA shows a significant drop in execution time, if either the AD or the message data is well-aligned. The same observation applies for iFeed, but with a less significant drop in matters of execution time. Both algorithms, COPA and iFeed, are using a padding function in case that the last block is not aligned to 16 bytes, causing the rise in the overall execution time. Additionally, COPA even performs one more gaulois multiplication in case of a misalignment, which is the reason, why the overall execution drop is even higher compared to iFeed. In contrast to the non-optimized algorithms, OTR and SILC are not padding the last block, if it is not aligned to 16 bytes, causing a roughly linear increase of the execution time. The second interesting fact, which can be observed, is that the

assumption, that algorithms being optimized for a microcontroller are encrypting faster, does not hold. Figure 8.1 shows, that actually iFeed is slightly faster than SILC. This is mainly due to the fact, that SILC is using modulo operations in its implementation and uses more function calls for each processed block as discussed in Chapter 5.2. Due to this increased function execution, the overall runtime becomes slower, the bigger the data gets. As already discussed in Section 8.1.2 applying the countermeasure, will eventually lead to a higher execution time due to the fact, that randomized lookup tables have to be generated. However, pre-computation only needs to be done once for each secured AES call. All selected AE algorithms have one critical part, where the AES should be secured, with one exception. OTR with the serial AD processing has two sensitive locations. Nevertheless, this means, that the secure AES is called once or, in worst case, twice for each AE encryption. Independent of the used size of the used input data. Our measurements showed, that the pre-computation of the needed randomized lookup tables requires 93.79 ms in total, without taking the time for generating P_k into account. The reason for that is, that we assume the master key being fixed. Therefore the table P_k will not change for the given device. We now set our constant value c to 93.79 ms. This value is a big overhead in matters of speed. For instance, the parallel version of OTR needs 1.01 ms for the smallest data set, with 2 byte message and no associated data. The same algorithm with applied countermeasures will require 94.80 ms. We can see, that 98.93 % of the overall runtime is needed to generate the lookup tables. If the input data is increased to a message size of 64 bytes and AD is increased to a size of 32 bytes, the percentage share of pre-computation decreases to 97.04 %. It can be seen, that in order to profit of the used countermeasure, the input data should be increased.

8.2 Implementation Attacks

The focus of this section is to give a detailed overview of the results of the performed implementation attacks. All attacks have been mounted on the ASM version of the AES, because it is used throughout every implementation. Furthermore, we will discuss the impact of the implemented countermeasure in matters of security and discuss the security of the AES hardware acceleration. All of the selected AE algorithms have AES as its underlying encryption scheme. As already discussed in Chapter 6 we know that all attacks aim to reveal the key by mounting a DPA attack on the AES. This is doable for all of the AE algorithms discussed in this thesis, hence we will discuss the results of mounting the DPA attack on the core AES. We will show that 2 000 traces are enough to reveal the secret key.

8.2.1 Attack Results on Unsecured AES

As already mentioned, we will attack the underlying AES on all AE algorithms with the help of DPA. Successfully performing such a DPA attack, requires recording power traces first. This was achieved with two different measurement setups:

– *Measurement using an EM Probe*
The first measurements were taken by using an EM probe that measures the power consumption via the electromagnetic field of the device. The probe is placed over the capacitor C9, which is illustrated in Figure 8.2. In order to perform the EM measurements this setup requires no circuit modification. Therefore it is set up very

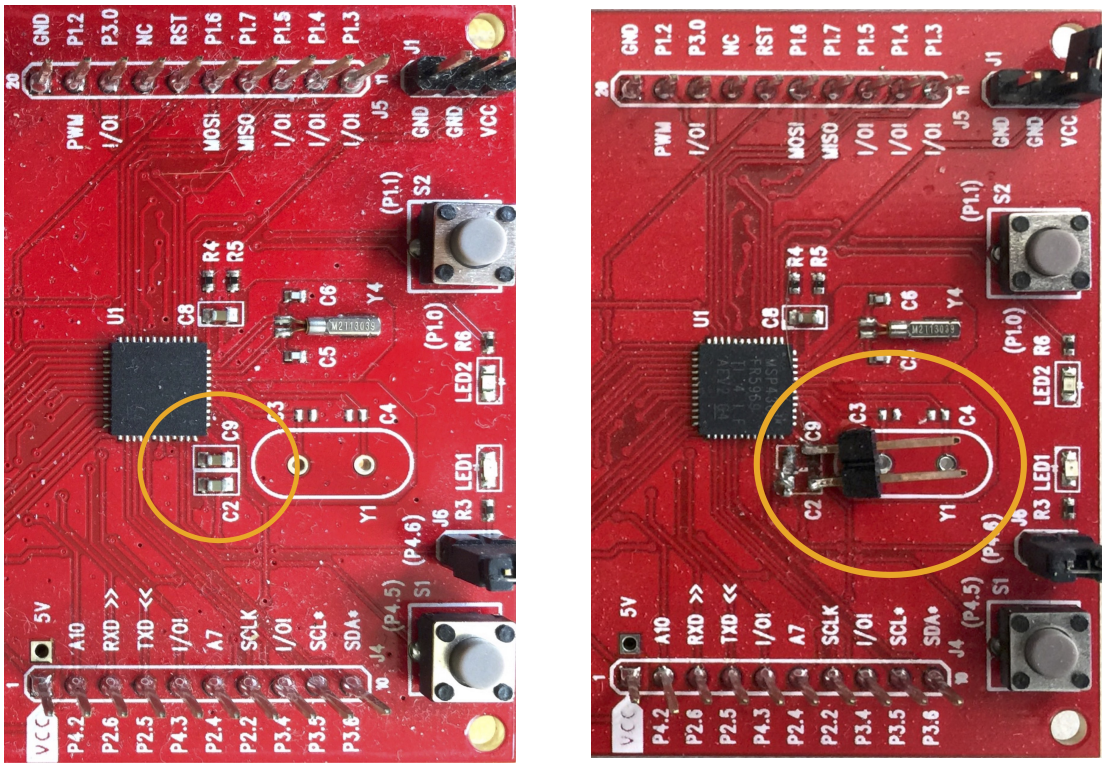

Figure 8.2: The left side shows, the position of the EM probe over capacitor C9, the right side shows the placement for the shunt resistor.

fast by just placing the probe on the position, where the maximum exploitable EM emanation can be detected. However, it has one drawback. As soon as the probe is misplaced, a recalibration to the correct position over the capacitor C9 was required.

– *Measurement using a Shunt Resistor*
Measuring the voltage drop across a shunt resistor in the supply line is the most common way to measure the power consumption. However, in most scenarios it requires a modification of the circuit to place the resistor. As seen in Figure 8.2 the capacitors C2 and C9 have been removed and a 1 Ω resistor is inserted into the DV_{CC} line of the MSP430 FR5969. The voltage drop across this resistor is now proportional to the current draw of the microcontroller.

Due to the fact, that the power consumption of the microcontroller is very low, we had to amplify the signal to get a better resolution independent of the used setup. This also leads to an increased noise level in the measured traces. As a result, the number of measurements for a successful attack can be significantly higher compared to older devices with a higher power consumption.

In order to find out how many traces are needed to get good results, we recorded 8 000 traces with a length of 80 000 samples. The recording of the power measurements was performed on a PicoScope 6404C, which is an USB oscilloscope able to perform 5 GS/s real-time sampling [5]. These power traces together with the hypothetical power model are now the input data set for the DPA attack. The correlation plot in Figure 8.3 shows the correlation for the key byte 11. It can be seen that the highest peak is at sample 34 423. Knowing the exact point in time, we now can calculate the evolution of the correlation of

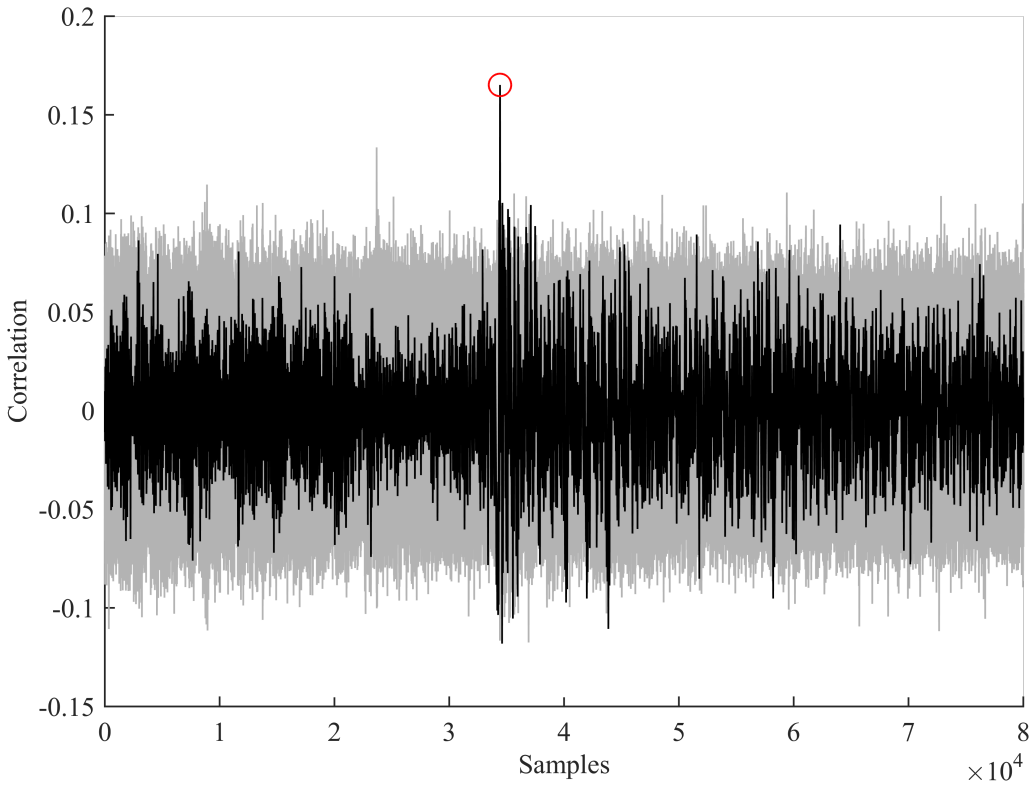

Figure 8.3: Correlation of one key byte. The highest correlation happens at sample 34 423 (red circle)

every possible key value in order to generate Figure 8.4. It can be seen that the correlation coefficient for the key value 90, which equals the correct key byte, is not converging to 0 in contrast to the other values. To be more precise, Figure 8.4 illustrates that the correlation value will not change significantly after roughly 2 000 traces. This basically means, that it is enough to record 2 000 traces to reveal the key byte.

In order to further prove the previous observation, we analyzed the success rate for different amount of traces. Calculation of the success rate is done by randomly picking x traces out of the set consisting of 8 000 traces and extracting the rank of the correct key byte ($\#(rank = 1)$). This experiment was repeated ten times ($\#experiments$). The formula to calculate the success rate, is as follows:

$$sr_x = \frac{\#(rank = 1)}{\#experiments} \tag{8.1}$$

We started by calculating the success rate with $x = 100$, the result was a success rate of 0.0. Meaning that 100 traces will not be enough to reveal the key. We increased the amount of traces to $x = 500, 1\,000, 1\,500, 2\,500, 3\,000$ and calculated the success rate of each x. Picking 500 random power measurements already revealed the correct key value once out of ten, which will result in a success rate of 0.1. 1 000 traces yields a success rate of 0.8, while collecting 1 500 traces already had a success rate of $sr_{1500} = 0.9$. However, collecting 2 000 traces or even more, will reveal the correct key value with a success rate of 1.0. This result is illustrated in Figure 8.5. This observation complies with our assumption initially assumed, that 2 000 traces are sufficient to reveal the correct key value. It should

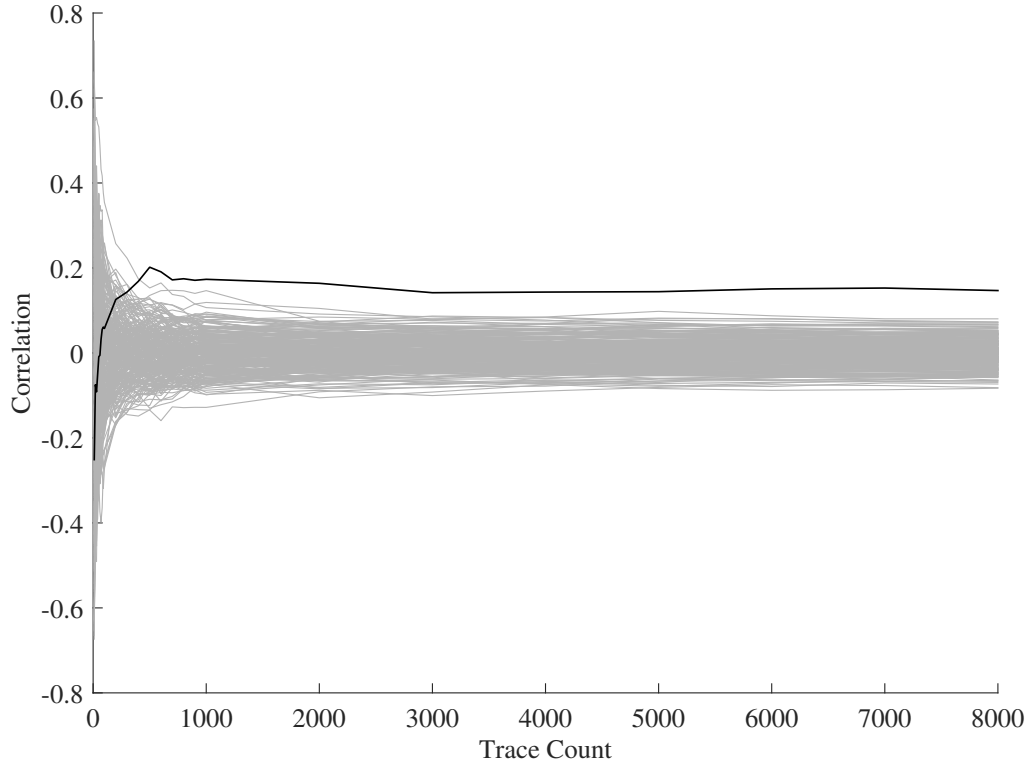

Figure 8.4: The black line shows the correlation for the correct key value 90 at sample 34 423

be noted again, that these attacks are for the ASM version of the AES. Attacking the AES acceleration of the MSP430 FR5969 would require a lot more traces. According to [33], 100 000 recorded traces are enough to reveal the correct key.
Knowing the amount of traces needed to reveal the secret information, we now can collect the power measurements for each of the implemented AE algorithms (SILC, iFeed, OTR, and COPA). As already discussed in Section 6.4 it is not always possible to reveal all key bytes. This is due to the fact, that most attacks target parts of the algorithm, where the public nonce is used. Unfortunately most of the time this nonce is less than 16 bytes long and will be padded with a constant value. Constant value will result in no exploitable leakage at all in matters of a DPA attack. However, in most cases the public nonce has a size of 12 bytes. In this case, we are able to recover 12 bytes of the secret key. The remaining 4 bytes have to be brute-forced. This can be accomplished in a conceivable timespan. The only exceptions are COPA and serial OTR. The latter is being attacked during processing associated data blocks, while the DPA attack on COPA is mounted while processing the plaintext. Starting their encryption process with the right size of input data, will eventually reveal the full secret key. COPA needs a message with at least 32 bytes and OTR one full AD data block (16 bytes).

8.2.2 Attack Results on Secured AES

The above results however differ, if the used AES version is secured with the masking countermeasure introduced in Section 7.3. This countermeasure is derived from the boolean masking scheme, with the difference, that it needs pre-computations of random lookup

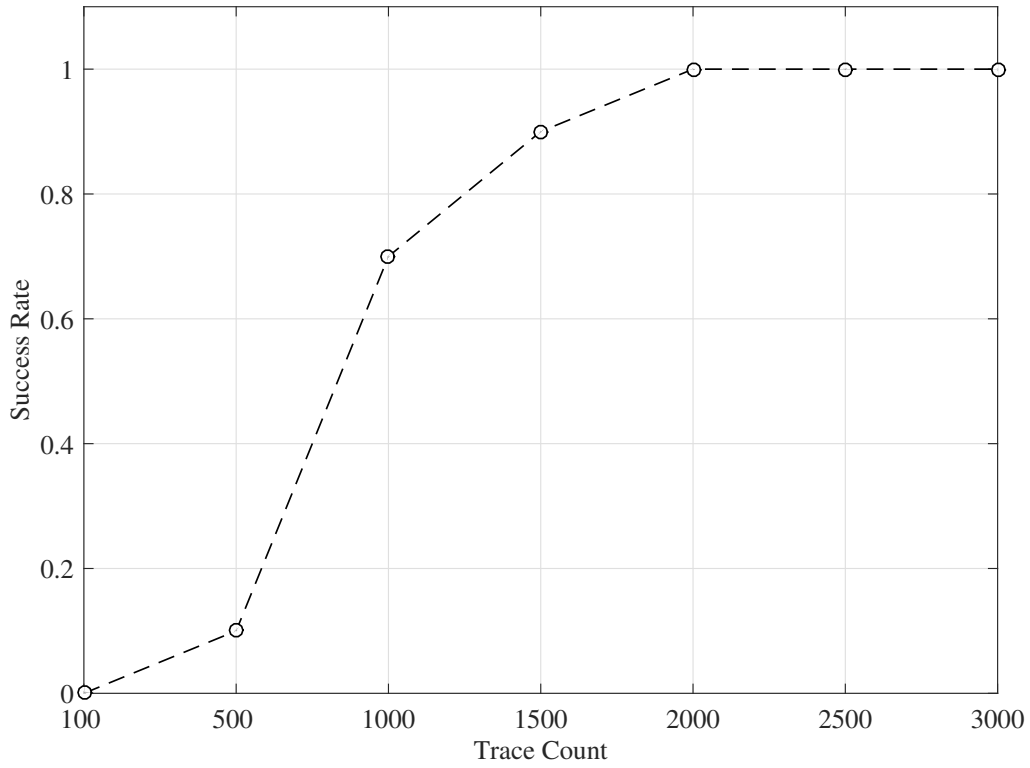

Figure 8.5: Success Rate of revealing the correct key value with different counts of traces. 2 000 traces result in a success rate of 1.0

tables (RLUT). Each of this tables are computed using a random variable independent of the other random variables used to compute the other RLUTs. Hence, this scheme is theoretically secure against attacks of any order. Observing the leakage of the tables will not reveal any information of the intermediate value, due to the fact, that all pre-computed tables are using different random values. However, in the paper [33] recently published, the authors detected a potential security flaw of this countermeasure in combination with the MSP430 FR5969. While the countermeasure scheme is theoretically secure against high-order attacks, it is the internal architecture of the MCU leaking secret information. Due to the internal structure of the FRAM memory, the device is leaking information of values saved on an address, where new values are written to. Any value already stored at this address will leak certain information about the old value. This basically means, writing a mask m on an address, where the intermediate value x was previously stored, will leak information about the value x. As a result it can happen, that during pre-computation, where the unmasked values are used, writing to the same address later on, will eventually leak information about the unmasked value x. Preventing such a leakage, requires the used address to be cleared right before writing to it.

We collected 200 000 traces with a sample length of 100 000 of the secured AES to verify that the previously used DPA attack will not leak any information about the secret value anymore. Figure 8.6 depicts the correlation values for all the possible values for key byte 11. The correct key value 90 is marked with a circle. After adding the countermeasure it is not possible anymore to identify the correct key. In order to extend the prove, we additionally calculated the evolution of correlation. In contrast to the previous calculation we did not use a specific point in time to calculate the correlation. As illustrated in Figure 8.7,

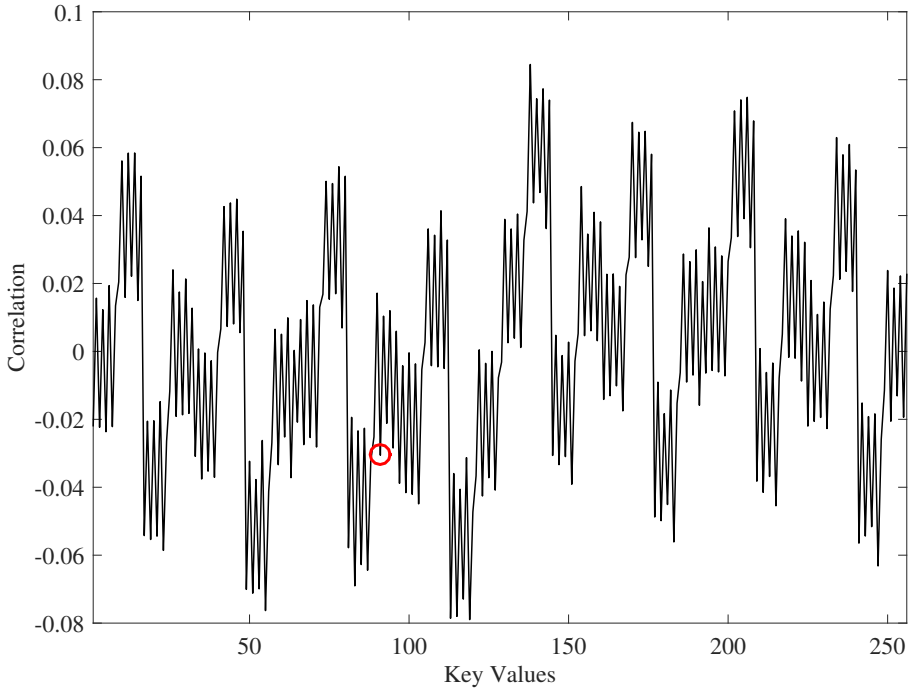

Figure 8.6: Correlation of one key byte as a result of the secured AES. 200 000 traces have been recorded. Same key value and key byte were used as for the unsecured version. The right key value is marked with a red circle

we chose the time between the sample 30 000 and 35 000, because there is a high data-dependent leakage. We calculated the correlation for the sample with the highest and with the lowest correlation, due to this fact, we have two black lines in Figure 8.7. Furthermore, it can be seen, that a lot of traces are not converging to 0. This observation results in a lot of possibilities for the right key value. This does in fact comply, with Figure 8.6. This circumstance can be explained through the changing masks of the countermeasure. Each time a new encryption is started, the masks m, a_1, a_2, a_3, needed to pre-compute the lookup tables of the masking countermeasure, are newly generated. All of those masks have a size of one byte, meaning that at some point in time, the masks will be repeated. Collecting 200 000 traces will, if the random variables are uniformly distributed, repeat each value roughly 781 times. Recalling the success rates of the unsecured version of AES, seen in Figure 8.5, we can see, that collecting roughly 781 traces will lead to a success rate of approximately $sr_{781} = 0.5$. This basically means, that 781 traces will be enough to create a high peak as seen in Figure 8.6. This observation happens for more than one key value, hence we have multiple values possible for the key byte. However, although there are various values leaking, it can be seen, that the correct key value is not present.

For further demonstration we calculated the evolution of convergence for an area, which does not have a high peak, like the area from sample 20 000 to 25 000. As depicted in Figure 8.8 all correlation traces are converging to 0.

Hence, we can say that the RLUT countermeasure is indeed not leaking any information about the intermediate values at all. However, it still has its drawbacks, because of the immense memory consumption needed in order to save the pre-computed tables. The version implemented in this thesis, masks the first and the last round of the AES, because

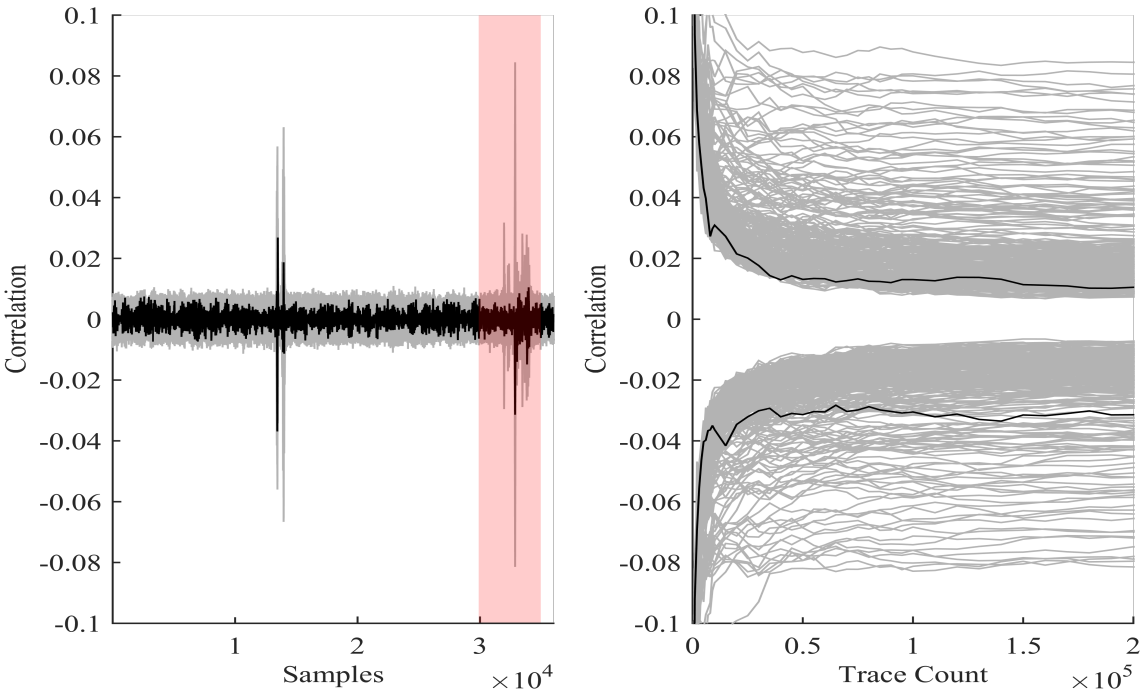

Figure 8.7: The left plot shows the correlation for all key bytes calculated with 200 000 traces. We calculated the evolution of correlation (right side) of the area marked in red (30 000 - 35 000). The correct key value is marked as black line.

64 KB of FRAM memory is not enough to fully pre-compute all tables needed to secure all rounds. Additionally, this result shows that the countermeasure using randomized lookup tables is working against a DPA attack. Nevertheless, future work has to include other state-of-the-art attacks in order to verify the resistance of the evaluated countermeasure against implementation attacks (e.g. high-order DPA, template attacks, etc.).

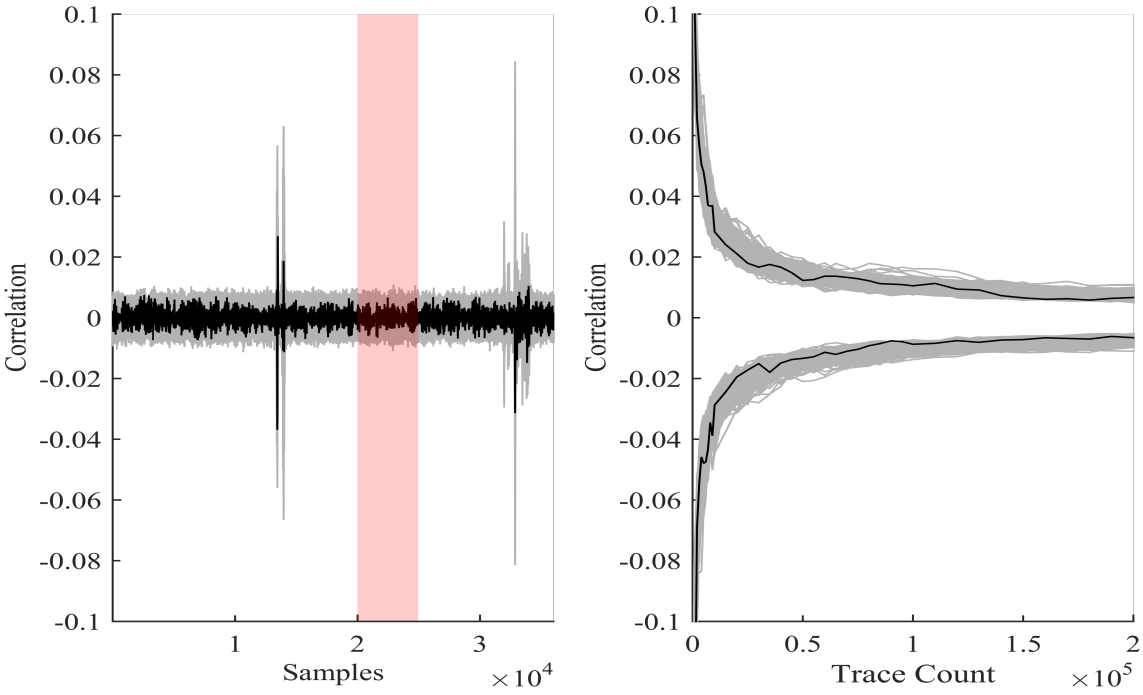

Figure 8.8: The left plot shows the correlation for all key bytes calculated with 200 000 traces. We calculated the evolution of correlation (right side) of the area marked in red (20 000 - 25 000). The correct key value is marked as black line.

Chapter 9

Conclusions

In this thesis, we have evaluated various algorithms submitted to the CAESAR competition. We have evaluated the algorithms applying AES as their default encryption scheme. Furthermore, we thinned out the remaining set of AE algorithms, by applying pre-defined evaluation criteria e.g. selecting algorithms optimized for embedded systems. The remaining four algorithms (iFeed, COPA, SILC, OTR) have been implemented and tested on a MSP430 FR5969 in matters of security and performance.

Furthermore, we analyzed the selected algorithms for possible vulnerability against differential power analysis (DPA) attacks. We have observed, that all algorithms are vulnerable against such attacks. All selected algorithms offer a location in its encryption scheme, where mounting a DPA attack revealed at least 12 bytes of the secret key, or even the full secret key. While revealing a part of the key was possible on all algorithms, this was not the case for the full key. The serial version of OTR and COPA are the only algorithms having such a weakness in their design, allowing us to reveal all bytes of the key. Furthermore, we discussed a previously proposed masking countermeasure optimized for MCUs equipped with FRAM memory. This countermeasure uses randomized lookup tables to be secure against high-order attacks of any degree. We learned, that in order to implement such a countermeasure, an immense amount of memory is needed. This is due to the fact, that all the needed lookup tables have to be pre-computed and stored. In this thesis, we showed a basic approach, how a software implementation of AES on a microcontroller can be secured with this scheme. Due to the fact, that the used MCU, has too little memory (64 KB) a tradeoff was made. In order to successfully add the masking countermeasure, we focused on securing the first and the last round of the AES. This was enough for the scope of this work, because the used DPA attack throughout this thesis was aiming for the first round of the AES. We learned, that this randomized lookup table (RLUT) countermeasure has, beside the immense memory demand, a significant overhead in matters of execution time. This is mainly because each AES execution is preceded by a re-computation of the lookup tables.

The aim of this thesis is on the one hand to evaluate the performance of current AE algorithms on an embedded system and on the other hand evaluate those algorithms for security vulnerabilities. In order to test the algorithms in matters of performance, benchmarks with pre-defined data sets have been performed. Results show, that all evaluated AE algorithms are well performing compared to the AES-GCM. We learned, that algorithms not optimized for embedded systems, perform better if the input data is aligned to the cipher's block size and additionally observed, that non-optimized algorithms, can be even faster, than optimized algorithms.

Furthermore, the vulnerability of the algorithms to implementation attacks is evaluated. As expected, all algorithms are vulnerable if no countermeasures are in place. Results show, that it is sufficient to record roughly 2 000 traces in order to reveal the secret key of any selected AE algorithm. However, if countermeasures are implemented, such DPA attacks should be harder. Due to the fact that countermeasures cannot be implemented on the hardware AES module of a MCU, the focus was to secure the software (assembler) version of the AES. This way it is possible to secure the selected AE algorithms in such a way, that vulnerable locations in the algorithms are using the secure software version of the AES, while all the other AES executions are performed by the hardware module. This way, a secure algorithm can be ensured without losing the benefit of a substantial performance increase due to the hardware acceleration. In order to secure the software version of the AES, a new masking randomized lookup table (RLUT) countermeasure was discussed. It was shown that this kind of countermeasure has to be modified in matters of memory usage. The lookup tables, needed for the countermeasure to work properly, have to be stored on memory, which should have at least the size of approximately 160 KB to save all the needed tables. That is too much for the used MCU, which has a memory size of 64 KB. As a result, the amount of needed memory was decreased, by securing only the first and last round of the software AES. As a result, we were able to show, that the previous DPA attacks are not successful anymore. 200 000 traces have been collected and the same DPA attack was applied, but as a result it was not possible to reveal the secret key.

We learned that, theoretically secure algorithms can still reveal the secret information with minimal effort, if the attacker is in possession of the actual cryptographic device and the algorithm is not properly secured against such physical attacks. Therefore, while implementing an algorithm on a MCU, the potential risk of a physical attack should be taken into account. However, securing algorithms against such attacks often come along with tradeoffs between increased execution time and level of protection. The proposed RLUT countermeasure is theoretically secure against attacks of any order, but cannot be fully implemented on the microcontroller available today, caused by the lack of sufficient memory to store the lookup tables. Future work should further investigate the RLUT countermeasure on devices with enough FRAM. Additionally, this thesis is covering basic physical attacks such as a first-order DPA attack. Therefore, future work should include the investigation of other attacks e.g. template attacks, cache attacks, or DPA attacks of higher order.

Appendix A

Definitions

A.1 Abbreviations

AE	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
CAESAR	Competition for AE: Security, Applicability, and Robustness
CBC	Cipher Block Chaining
CCM	Counter Mode with CBC-MAC
DPA	Differential Power Analysis
ECB	Electronic Codebook Mode
FRAM	Ferroelectric Random Access Memory
GCM	Gaulois/Counter Mode
GPIO	General Purpose Input/Output
LPM	Low-power Mode
MAC	Message Authentication Code
MCU	Microcontroller Unit
MPU	Memory Protection Unit
NIST	National Institute of Software and Technology
OCB	Offset Codebook Mode
RLUT	Randomized Lookup Table
SCA	Side-channel Analysis
SPA	Simple Power Analysis
UART	Universal Asynchronous Receiver/Transmitter

Bibliography

[1] Advanced Encryption Standard (AES). Available online at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, Last accessed: 2015-05-15.

[2] AES-CCM. Available online at https://tools.ietf.org/html/rfc3610, Last accessed: 2015-05-15.

[3] CAESAR Competition. Available online at http://competitions.cr.yp.to/caesar.html, Last accessed: 2015-05-15.

[4] Offset Codebook Mode (OCB). Available online at http://web.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm, Last accessed: 2015-05-15.

[5] PicoScope 6404C - Manual. Available online at https://www.picotech.com/oscilloscope/6000/picoscope-6000-manuals, Last accessed: 2015-05-18.

[6] Rijndael (AES). Available online at http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf, Last accessed: 2015-05-18.

[7] M.-L. Akkar and C. Giraud. An implementation of DES and AES, secure against some attacks. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 309–318. Springer, 2001.

[8] J. A. Ambrose, R. G. Ragel, and S. Parameswaran. RIJID: random code injection to mask power analysis based side channel attacks. In *Proceedings of the 44th annual Design Automation Conference*, pages 489–492. ACM, 2007.

[9] E. Andreeva, A. Bogdanov, A. Luykx, B. Mennink, E. Tischhauser, and K. Yasuda. AES-COPA. Available online at http://competitions.cr.yp.to/round1/aescopav1.pdf, Last accessed: 2015-05-15.

[10] Atmel. 8/16-bit Atmel XMEGA A3U Microcontroller. Available online at http://www.atmel.com/devices/ATXMEGA64A3U.aspx, Last accessed: 2015-05-15.

[11] J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). In *Financial Cryptography*, pages 162–181. Springer, 2003.

[12] M. Dworkin. Recommendation for Block Cipher Modes of Operation. Available online at http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf, Last accessed: 2015-05-15.

[13] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 251–261. Springer, 2001.

[14] J. D. Golić and C. Tymen. Multiplicative masking and power analysis of AES. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 198–212. Springer, 2003.

[15] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw. The LED block cipher. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 326–341. Springer, 2011.

[16] M. Höfler. Investigating the Vulnerabilities of Two Microcontroller Platforms to Fault Injection Attacks. Master's thesis, Technical University Graz, 2014.

[17] T. Instruments. AES Implementation. Available online at `http://www.ti.com/tool/aes-128`, Last accessed: 2015-05-15.

[18] T. Instruments. Code Composer Studio. Available online at `http://www.ti.com/tool/CcStudio`, Last accessed: 2015-05-15.

[19] T. Instruments. MSP Low-Power Microcontrollers. Available online at `http://www.ti.com/lit/sg/slab034aa/slab034aa.pdf`, Last accessed: 2015-05-15.

[20] T. Instruments. MSP430 FRAM Technology – How To and Best Practices. Available online at `http://www.ti.com/lit/an/slaa628/slaa628.pdf`, Last accessed: 2015-05-15.

[21] T. Instruments. MSP430FR59xx Mixed-Signal Microcontrollers. Available online at `http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf`, Last accessed: 2015-05-15.

[22] T. Instruments. Random Number Generation Using the MSP430. Available online at `http://www.ti.com/lit/an/slaa338/slaa338.pdf`, Last accessed: 2015-05-15.

[23] T. Instruments. User's Guide. Available online at `http://www.ti.com/lit/ug/slau535a/slau535a.pdf`, Last accessed: 2015-05-15.

[24] T. Iwata, K. Minematsu, J. Guo, S. Morioka, and E. Kobayashi. SILC: SImple Lightweight CFB. Available online at `http://competitions.cr.yp.to/round1/silcv1.pdf`, Last accessed: 2015-05-15.

[25] S. Kerckhof, F.-X. Standaert, E. Peeters, et al. From new technologies to new solutions exploiting FRAM memories to enhance physical security. In *proceedings of CARDIS*, 2013.

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

[27] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[28] O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In *USENIX workshop on Smartcard Technology*, volume 12, pages 9–20, 1999.

[29] T. Korak and M. Höfler. On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms. In *11th Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2014. in press.

[30] T. LLC. Wireless Identification and Sensing Platform (WISP). Available online at `http://keccak.noekeon.org/Keccak-specifications-2.pdf`, Last accessed: 2015-05-15.

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

[32] K. Minematsu. AES-OTR. Available online at `http://competitions.cr.yp.to/round1/aesotrv1.pdf`, Last accessed: 2015-05-15.

[33] A. Moradi and G. Hinterwälder. Side-Channel Security Analysis of Ultra-Low-Power FRAM-based MCUs. *proceedings of COSADE*, 2015.

[34] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE, 2013.

[35] E. Oswald and M. Aigner. Randomized addition-subtraction chains as a countermeasure against power attacks. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 39–50. Springer, 2001.

[36] D. Saha, D. Mukhopadhyay, and D. R. Chowdhury. A Diagonal Fault Attack on the Advanced Encryption Standard. *IACR Cryptology ePrint Archive*, 2009:581, 2009.

[37] E. Savaş. Attacks on Implementations of Cryptographic Algorithms: Side-channel and Fault Attacks. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 7–14, New York, NY, USA, 2013. ACM.

[38] J.-M. Schmidt and C. H. Kim. A probing attack on AES. In *Information Security Applications*, pages 256–265. Springer, 2009.

[39] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 2–12. Springer, 2003.

[40] L. Zhang, W. Wu, H. Sui, and P. Wang. iFeed[AES] v1. Available online at `http://competitions.cr.yp.to/round1/ifeedaesv1.pdf`, Last accessed: 2015-05-15.