

Efficient Implementation of Stream Ciphers on Embedded Processors

Gordon Meiser

07.05.2007

Master Thesis
Ruhr-University Bochum (Germany)



Chair for Communication Security
Prof. Dr.-Ing. Christof Paar
Co-Advised by Kerstin Lemke-Rust
and Thomas Eisenbarth

“They who would give up an essential liberty for temporary security, deserve neither liberty or security.”(Benjamin Franklin, 1706-1790)

STATEMENT / ERKLÄRUNG

I hereby declare, 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 references to other authors.

Hiermit versichere ich, dass ich meine Diplomarbeit selber verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Gezeichnet

Gordon Meiser

Ort, Datum

ACKNOWLEDGEMENT

It is my pleasure to express my gratitude to all the people who contributed, in whatever manner, to the success of this work.

I want to thank Prof. Dr.-Ing. Christof Paar for giving me the possibility to write my master thesis at the Chair for Communication Security at the Ruhr-University Bochum. Furthermore I'd like to thank my Co-Advisors Dipl.-Phys. Kerstin Lemke-Rust and Dipl.-Ing. Thomas Eisenbarth for advising me with writing scientific papers and answering all my questions. I also want to thank all the people sitting in my room, especially Leif Uhsadel and Sören Rinne, who helped me keep the focus on my work. I spent many nights working on my master thesis and with Leif this time was much easier to bear. Another thank you goes to all reviewers, especially to Phill Cabeen for reading, correcting, and revising my master thesis from the view of a native English speaker.

The biggest 'THANK YOU' goes to my father Guido Meiser, my mother Rita Meiser and my brother Timm Meiser for always supporting me in my ambition to complete my studies successfully. They accompanied me on my way, no matter where that led me.

ABSTRACT

In this thesis, we present the first implementation results for Dragon, HC-128, LEX, Salsa20, and Sosemanuk (which are stream ciphers in Phase 2 of Profile 1 of the eSTREAM project) on 8-bit microcontrollers.

For the evaluation process, we follow a two-stage approach and compare with efficient AES implementations. First, the C code implementation provided by the designers is ported to an 8-bit AVR microcontroller and the suitability of Dragon, HC-128, LEX, Salsa20, and Sosemanuk for the use in embedded systems is assessed. In the second stage we implement the stream ciphers in Assembly to tap the full potential of an embedded implementation. Our efficiency metrics are performance of keystream generation, key setup, and IV setup, and memory usage in flash memory and SRAM, since microcontrollers are usually strongly constrained in memory resources.

Concerning throughput, all stream ciphers outperform the AES. Sosemanuk, for instance, reaches three times the throughput of the AES. In terms of memory requirements, Salsa20 and LEX are almost as compact as AES. When considering the time-memory tradeoff metric, LEX and Salsa20 yield significantly better results than AES.

If we involve the time-memory tradeoff metric of our C language implementations and compare it with the time-memory tradeoff metric of our Assembly language implementations, we can observe immense improvements of all ciphers. For instance, the time-memory tradeoff value of LEX is more than 1,800% better in Assembly language, whereby Sosemanuk holds the least enhancement with – only – 600% increase.

Considering flash memory requirements all ciphers are executable on smaller devices in Assembly language. E.g. the Assembly implementation of the Dragon cipher counts 12% of the original codesize, allowing the execution on even the smallest device of the AVR family (ATmega8) instead of the biggest device (Atmega128).

To the best of our knowledge this thesis includes the fastest implementations world wide of Dragon, LEX, HC-128, Salsa20, and Sosemanuk for the 8-bit AVR microcontroller family.

Contents

Statement	iii
Acknowledgement	iv
Abstract	v
Table of Contents	vii
List of Figures	xi
List of Tables	xiii
Nomenclature	xv
1 Introduction	1
1.1 Motivation	1
1.2 Aim of this Thesis	2
1.3 Approach	2
1.4 Organization of this Thesis	2
2 Cryptology	5
2.1 Cryptography	5
2.1.1 Block Ciphers	6
2.1.2 Stream Ciphers	9
2.1.3 Synchronous Stream Ciphers	10
2.1.4 Self-Synchronizing Stream Ciphers	10
2.1.5 Linear Feedback Shift Registers	11
2.2 Cryptanalysis	12
2.2.1 Linear Cryptanalysis	12
2.2.2 Differential Cryptanalysis	12
3 Focused eSTREAM Profile I Candidates	15
3.1 Introduction to ECRYPT/eSTREAM	15
3.2 AES	16
3.2.1 Workflow of AES	16

3.2.2	Key expansion	18
3.2.3	AddRoundKey	18
3.2.4	SubBytes	19
3.2.5	ShiftRows	19
3.2.6	MixColumns	20
3.3	Dragon	20
3.3.1	The update function F	20
3.3.2	G and H functions	22
3.3.3	Key and IV initialization	22
3.3.4	Keystream generation	23
3.4	HC-128	23
3.4.1	Specification of the cipher	23
3.4.2	Initialization process	24
3.4.3	Keystream generation	25
3.5	LEX	26
3.6	Salsa20	28
3.6.1	The quarterround function	28
3.6.2	The rowround function	28
3.6.3	The columnround function	29
3.6.4	The doubleround function	29
3.6.5	The littleendian function	29
3.6.6	The Salsa20 hash function	29
3.6.7	The expansion function	30
3.6.8	The encryption function	31
3.7	Sosemanuk	31
3.7.1	Serpent and its derivatives	31
3.7.2	The LFSR	32
3.7.3	The FSM	33
3.7.4	Key Setup	33
3.7.5	IV Injection	34
4	Microcontroller	35
4.1	Microcontroller	35
4.2	Atmel	36
4.3	ATmega Series	36
5	Framework Set-Up and Tool Chain	39
5.1	Porting to AVR Microcontrollers	39
5.2	Development Tools	40
5.2.1	WinAVR	40
5.2.2	AVRStudio	42
5.3	Configuration for Testing	43

5.3.1	C language configuration	43
5.3.2	Assembly language configuration	43
6	Results in C	45
6.1	Objectives	45
6.2	Implementation	45
6.2.1	AES	45
6.2.2	Dragon	46
6.2.3	HC-128	48
6.2.4	LEX	48
6.2.5	Salsa20	49
6.2.6	Sosemanuk	50
6.3	Results	51
6.3.1	Memory Usage	51
6.3.2	Performance	53
7	Results in Assembly Language	55
7.1	Goals and Objectives	55
7.2	Implementation	56
7.2.1	AES	56
7.2.2	Dragon	58
7.2.3	LEX	63
7.2.4	Salsa20	66
7.2.5	Sosemanuk	70
7.3	Results	76
7.3.1	Memory Usage	76
7.3.2	Performance	77
8	Summary and Future Work	81
8.1	Summary	81
8.2	Future Work	86
A	Appendix	87
A.1	C language implementations	87
A.2	Assembly language implementations	88
B	Bibliography	89

List of Figures

2.1	Overview Cryptology	5
2.2	ECB mode encryption	7
2.3	CBC mode encryption	7
2.4	CFB mode encryption	8
2.5	OFB mode encryption	9
2.6	CTR mode encryption	9
2.7	Linear Feedback Shift Register	11
3.1	AES: Graphical round overview	17
3.2	AES: Key schedule	18
3.3	AES: SubBytes() function	19
3.4	AES: ShiftRows() function	19
3.5	AES: MixColumns() function	20
3.6	Dragon: Update function F	21
3.7	Dragon: Update function F (graphically illustrated)	21
3.8	Dragon: Key initialization function (taken from [14])	22
3.9	Dragon: Keystream generation function (taken from [14])	23
3.10	LEX: Overview	26
3.11	LEX: Intermediate round values (128 bit)	27
3.12	LEX: Extraction on odd rounds	27
3.13	LEX: Extraction on odd rounds	28
3.14	Sosemanuk: The LFSR of Sosemanuk	32
3.15	Sosemanuk: Overview	34
5.1	Snapshot of Mfile	40
5.2	Generation of project tool “extended coff”	41
5.3	Snapshot of AVR Studio 4 - watching the struct ctx of Dragon	42
7.1	AES: Memory allocation in SRAM	56
7.2	Dragon: Memory allocation in SRAM	58
7.3	Dragon: Key setup	60
7.4	Dragon: First part of the IV setup	61
7.5	LEX: Memory allocation in SRAM	63
7.6	Salsa20: Memory allocation in SRAM	66
7.7	Salsa20: Left rotation of a 32-bit value by 7 bits	68

7.8	Salsa20: Left rotation of a 32-bit value by 9 bits	68
7.9	Salsa20: Left rotation of a 32-bit value by 13 bits	69
7.10	Salsa20: Left rotation of a 32-bit value by 18 bits	69
7.11	Sosemanuk: Memory allocation in SRAM	70
7.12	Sosemanuk: Left rotation of a 32-bit value by 11 bits	71
7.13	Sosemanuk: Left shift of a 32-bit value by 7 bits	72
7.14	Sosemanuk: Fast implementation of a (32×32) -bit multiplication mod 2^{32}	74
8.1	Comparison of C and Assembly language implementations: flash memory consumption	82
8.2	Comparison of C and Assembly language implementations: SRAM con- sumption	83
8.3	Comparison of C and Assembly language implementations: throughput .	84
8.4	Comparison of C and Assembly language implementations: time memory tradeoff	85

List of Tables

3.1	HC-128: Operations used by HC-128	24
3.2	Salsa20: The quarterround function	28
3.3	Salsa20: The rowround function	29
3.4	Salsa20: The columnround function	29
4.1	Specification of the most popular AVR devices (ATmega family)	36
6.1	Memory allocation in flash memory of C implementations	51
6.2	Memory allocation in SRAM of C implementations	52
6.3	Performance of initialization, key setup, IV setup, and encryption of C implementations (all numbers given are measured CPU cycles)	53
6.4	Throughput of encryption of C implementations	54
7.1	Memory allocation in flash memory of Assembly implementations	76
7.2	Memory allocation in SRAM of Assembly implementations	77
7.3	Performance of initialization, key setup, IV setup, encryption of Assembly implementations (all numbers given are measured CPU cycles)	78
7.4	Throughput of encryption of Assembly implementations	78

Nomenclature

AES	Advanced Encryption Standard
CBC	Cipher-Block Chaining
CFB	Cipher Feedback
CPU	Central Processing Unit
CTR	Counter
DES	Data Encryption Standard
ECB	Electronic Code Book
ECRYPT	European Network of Excellence for Cryptology
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
FSM	Finite State Machine
I/O	Input/Output
IV	Initialization Vector
LE	Little Endian
LFSR	Linear Feedback Shift Register
NLFSR	Non-Linear Feedback Shift Register
OFB	Output Feedback
OTP	One Time Pad
PN	Programmers Notepad
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
SRAM	Static Random Access Memory
XOR	Exclusive Or

1 Introduction

1.1 Motivation

In 2005, the European Network of Excellence in Cryptology (ECRYPT) launched a call for stream cipher primitives [6] to identify new stream ciphers suitable for widespread adoption that may also serve as an alternative for the AES [8]. Profile I of this call asked for stream ciphers for software applications with high throughput requirements, while Profile II aims at identifying stream ciphers suitable for hardware applications with restricted resources such as limited storage, gate count, or power consumption.

The original call says that performance benchmarking for Profile I “may include 8-bit processors (as found in inexpensive smart cards), 32-bit processors (e.g., the Pentium family) to the modern 64-bit processors”. However, the current testing framework of the eSTREAM project [7, 13, 11] exclusively targets general-purpose 32-bit and 64-bit CPUs for Profile I candidates. Given the great importance of small embedded controllers in the real world (the market share of embedded processors is more than 99%), we feel that such an evaluation is of value.

This master thesis is driven by the question of how efficient candidates in the current focus of Profile I can be implemented on small 8-bit embedded microprocessors. Small 8-bit microprocessors are constrained in resources such as flash memory and RAM. Besides throughput, efficiency has a particular meaning in this context: resources needed by an implementation of a stream cipher should be kept small, since embedded applications are very often cost constrained. In fact, in many situations cost (given by memory consumption) is more crucial than throughput, in particular because many embedded applications only encrypt small payloads.

Small 8-bit embedded microprocessors are widely used in various applications, including smart cards, household appliances, industrial control, and many other systems. Modern cars, for instance, are equipped with more than fifty microcontrollers. In embedded systems, cryptography is often needed for authentication, secure messaging, and software download.

Though an 8-bit microprocessor may not be the mainstream target platform of Profile I we are confident that there is also a wide public and industrial interest in finding out whether candidates of Profile I can also serve as possible secure alternatives to AES on small 8-bit embedded microprocessors.

The results presented in this thesis are the first published benchmarking numbers of eSTREAM candidates on 8-bit AVR microcontrollers. This thesis thereby contributes to the evaluation of the focused Profile I candidates.

1.2 Aim of this Thesis

In this work, we evaluate performance aspects of stream ciphers that are in the focus of Profile I of Phase 2 of eSTREAM and not yet broken. In detail, this work covers Dragon, HC-128, LEX, Salsa20, and SOSEMANUK. Because of recently reported key recovery attacks, Py/Pypy [25] and Phelix [24] are not included in our testing framework. The aim of this thesis is the efficient implementation of the above mentioned ciphers on 8-bit AVR microcontrollers and the resulting performance benchmarks.

1.3 Approach

For the evaluation of Profile I candidates, the C code implementation provided by the designers is ported to an 8-bit AVR microcontroller. For comparison, we also implement the byte-oriented AES taken from Gladman [15]. Our comparison metric includes (i) throughput of keystream, (ii) time needed for key setup, (iii) time needed for IV setup, (iv) memory allocation in flash (program code), and (v) memory allocation in SRAM (variables).

Furthermore we carry out an evaluation of the most promising candidates in Assembly language. We implement the AES (for comparison reasons), DRAGON, LEX, Salsa20, and Sosemanuk. We excluded HC-128 from this list of Assembly implementations because of its huge amount of required SRAM and the hence resulting used ATmega1281 device.

1.4 Organization of this Thesis

This thesis is organized as follows: in Chapter 2 a general introduction to cryptology focusing on block and stream ciphers is given. Then in Chapter 3 we provide the specifications for the implemented ciphers in this work. For comparison reasons, we also included the AES. In Chapter 4 an introduction to microcontrollers, the Atmel Corporation and their product, the ATmega family of 8-bit microcontrollers is given. Afterwards in Chapter 5 we present our used tool-chain and some important adjustments we made. Chapter 6 gives an introduction to our C language implementations. First, the separate ciphers and their implementations are presented, followed by a summarization of their

performance results and memory requirements. Almost the same is done in the following Chapter 7. Here our results with the ciphers efficiently implemented in Assembly language are presented. Finally, we conclude the thesis with a summary and the future work in Chapter 8.

2 Cryptology

Cryptology is the science of information security. It is widely diversified and also covers several fields like authentication, access control, network security, and information confidentiality, to mention just a few. Cryptology can be divided into two principal parts, namely cryptography (e.g. the development of new encryption techniques) and cryptanalysis (e.g. the science of breaking cryptographic tools).

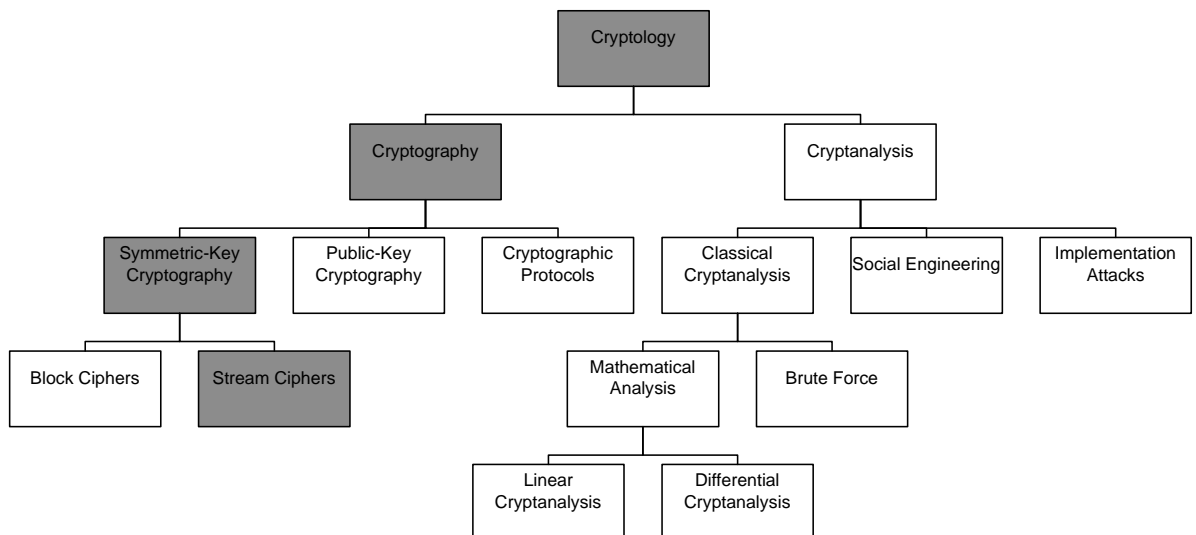


Figure 2.1: Overview Cryptology

2.1 Cryptography

Cryptography can be segmented into three main disciplines: symmetric-key cryptography, public-key/asymmetric-key cryptography and cryptographic protocols. Symmetric-key cryptography ensuring confidentiality deals with encryption methods in which the sender and the receiver use the same key to encrypt and decrypt messages. Problems occur if the number of participants increases. Public-key cryptography has been publicly known since a paper by Whitfield Diffie and Martin Hellman was published in the year 1976. Here the sender and the receiver are no longer forced to use the same key for

encryption and decryption. There exist a private key and a public key which are mathematically related, but the private key cannot be derived from the public key within any practical time. The private key is kept secret, while the public key can be distributed without restrictions, but may be certified by a trusted third party. Cryptographic protocols are procedures where participants can exchange sensible data over unprotected channels.

The field of symmetric-key cryptography can be divided into three sub-disciplines: block ciphers, hash functions and stream ciphers. Block ciphers and stream ciphers will be discussed in the following sections. A short introduction to hash functions is given in section 3.6, as part of the Salsa20 cipher.

2.1.1 Block Ciphers

When using block ciphers the plaintext must be subdivided in blocks of fixed length (typically 64, 128, and 256 bits), because block ciphers encrypt and decrypt block-wise. A block cipher requires a key and the plaintext as input and outputs the corresponding ciphertext. Mathematically described, a block cipher consists of two parts: an encryption function E and a decryption function $D = E^{-1}$. Let n be the size of a plaintext block, let k denote the key, m is the plaintext and c the corresponding ciphertext, then:

$$c = E_k(m) \text{ and } m = D_k(c) = E_k^{-1}(c) = E_k^{-1}(E_k(m))$$

Many block ciphers follow a simple construction schema. They use simple operations like rotation, shift, XOR and substitutions (also called S-boxes), but repeatedly apply this operations for n iterations (n normally lies between 4 and 32). The iteration is called a ‘round’ and the simple construction that is applied in every round is called a ‘round function’. Dependent on the length of the plaintext, it may be necessary to expand the plaintext to a multiple of the block size. There are different modes available when using a block cipher for enciphering multiple blocks. In the following sections we take a closer look to some dependent modes of operation, especially the ones that are able to build a stream cipher.

ECB Mode

The electronic code book mode (ECB) is the simplest mode of operation. Each block of ciphertext is encrypted separately. One huge disadvantage of this mode is the fact that identical blocks of plaintext lead to identical blocks of ciphertext, while the key stays untouched. Similarly, a ciphertext block could be exchanged without detection on the side of the receiver.

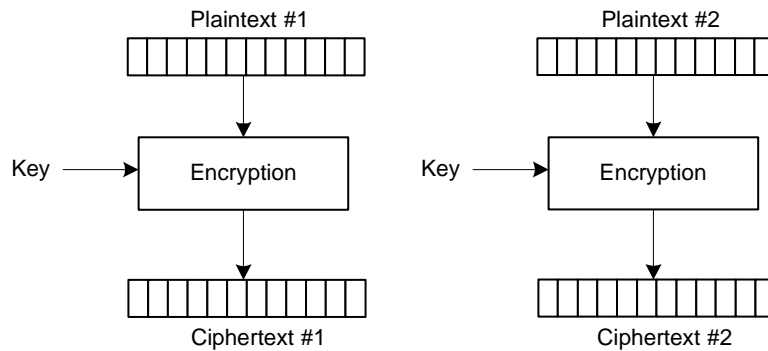


Figure 2.2: ECB mode encryption

CBC Mode

The cipher-block chaining mode (CBC) requires the ciphertext block of the previous round to encrypt the plaintext block of the current round. As there is no previous ciphertext in the first round an initialization vector (IV) is needed. In this mode the previous ciphertext block is XORed with the plaintext before encryption. The mathematical formula for the CBC encryption is:

$$c_0 = IV, c_i = E_k(m_i \oplus c_{i-1})$$

and for the decryption:

$$c_0 = IV, p_i = E_k^{-1}(c_i) \oplus c_{i-1}$$

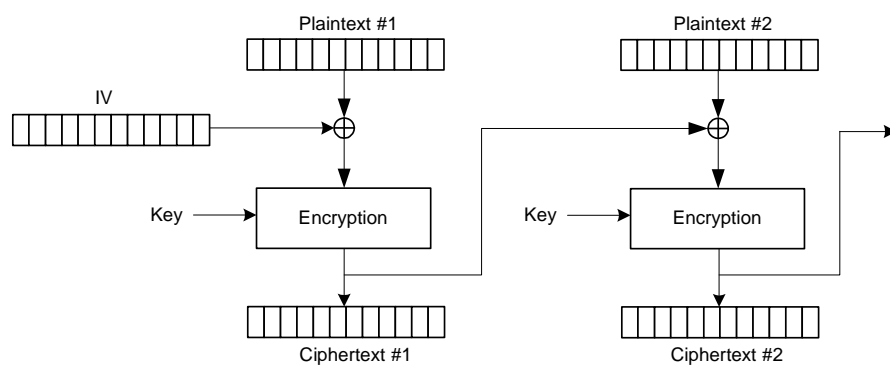


Figure 2.3: CBC mode encryption

NOTE: A flipped bit in the plaintext affects all following ciphertexts. Decryption can be parallelized, which is not possible for encryption.

CFB Mode

The cipher feedback mode (CFB) works similarly to the CBC mode. In the first round the IV serves as input to the encryption function. In the following rounds the previous ciphertext is used as input. The output of the encryption function is XORed with the plaintext. The hence resulting block is both, the ciphertext and the input for the encryption function in the next round as well. A block cipher in CFB mode builds a self-synchronizing stream cipher (see section 2.1.4). The mathematical formulas are the following:

$$\begin{aligned} \text{encryption: } c_0 &= IV, c_i = E_k(c_{i-1}) \oplus m_i \\ \text{decryption: } c_0 &= IV, m_i = E_k(c_{i-1}) \oplus c_i \end{aligned}$$

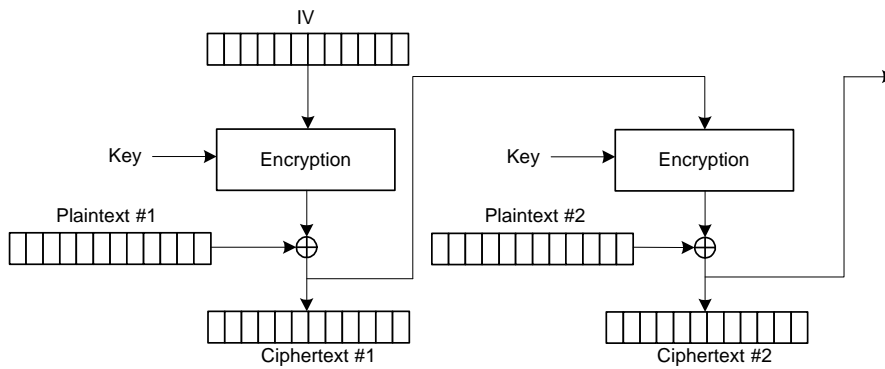


Figure 2.4: CFB mode encryption

NOTE: Only the encryption function is required here. A change in the plaintext propagates forever in the ciphertext and decryption can be parallelized.

OFB Mode

The output feedback mode (OFB) encrypts the previous encrypted block and passes it as input to the encryption function in the next call. Subsequently this value is XORed with the plaintext and builds the ciphertext. This transforms a block cipher, running in OFB mode, into a synchronous stream cipher (see section 2.1.3). If o_i is the output of the encryption function of the i -th iteration, then:

$$\begin{aligned} o_0 &= IV \text{ and } o_i = E_k(o_{i-1}) \\ c_i &= m_i \oplus o_i \text{ and } m_i = c_i \oplus o_i \end{aligned}$$

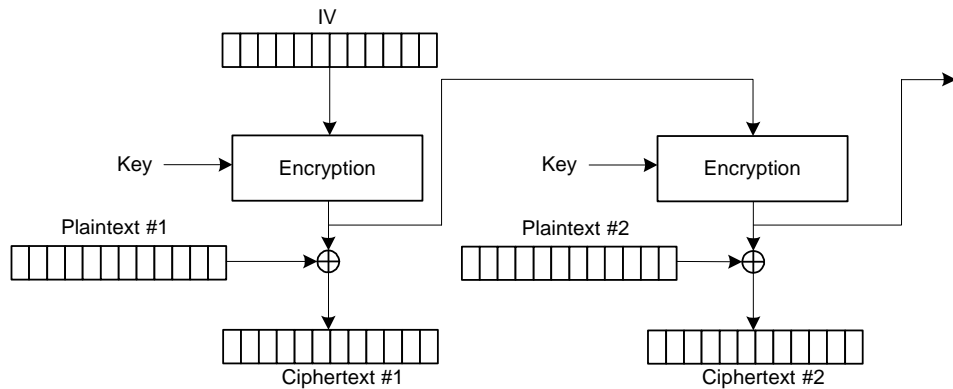


Figure 2.5: OFB mode encryption

CTR Mode

The counter mode (CTR), like the OFB mode, turns a block cipher into a synchronous stream cipher. Here a combination of an initialization vector (or nonce) and a counter is encrypted by the encryption function. The output of the encryption function is then XORed with the plaintext to generate the ciphertext. The combination of the nonce and the counter can be an exclusive-or, an addition, or the concatenation of the two values. The counter must be a function that creates different output at every call, whereby the output should only repeat itself after a very long time.

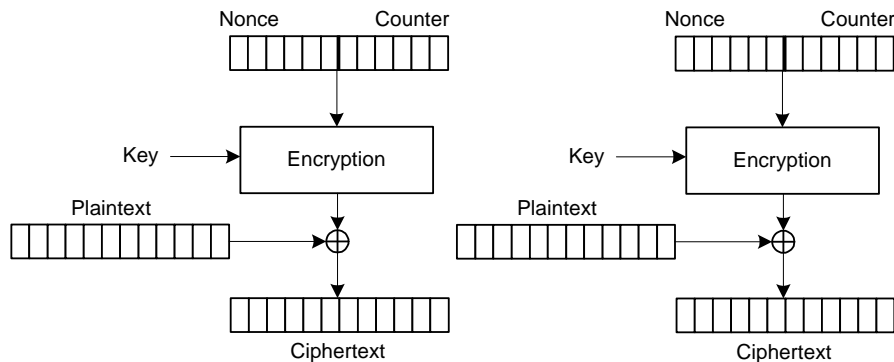


Figure 2.6: CTR mode encryption

2.1.2 Stream Ciphers

A stream cipher is a symmetric cipher which encrypts individual characters one at a time, whereby the keystream transformation varies with time, dependent on the current

internal state of the cipher. All stream ciphers are attempts to realize a cipher similar to the One Time Pad (OTP) cipher but without its disadvantages, mentioned below.

The OTP is a symmetric cipher as well. The big difference to other ciphers lies in the key length and its creation. The big disadvantages of the OTP are the following: the length must be at least as long as the plaintext, the key shall only be used once and must be created truly randomly. After key creation, the key is added to the plaintext (for example by XOR). If these restrictions are fulfilled, the OTP possesses the characteristic of ‘perfect secrecy’. Another disadvantage of the OTP additionally to the huge key length is the key transport to the communication partner.

Stream ciphers use a key and an initialization vector to build a ‘pseudo-random’ keystream before the keystream is XORed with the plaintext. Depending on how a stream cipher updates its internal state, stream ciphers can be grouped into two classes: synchronous stream ciphers and self-synchronizing stream ciphers. If the update function of the internal state is independent of the plaintext or ciphertext message, the cipher is classified as a synchronous stream cipher. In the contrary case, in which the cipher updates the inner state depending on either the previous plain- or ciphertext, the cipher is called a self-synchronizing stream cipher. Often, a stream cipher runs at a higher speed and has a lower hardware complexity if compared to block ciphers.

2.1.3 Synchronous Stream Ciphers

Using a synchronous stream cipher implies that the sender and the receiver are synchronized. If digits are lost, added or changed during transmission, the synchronization gets lost and the plaintext or ciphertext cannot be correctly assembled. But there are various options to retain synchronization. For instance, by tagging the ciphertext with markers at constant points in the stream or by using offsets to obtain the correct decryption. The advantage of this mode is the fact that, if a bit is defective, only the corresponding ciphertext bit is erroneous. The rest of the message is not affected. This property is of high value if the error rate is very large. But on the other hand, it is more difficult to prevent such errors without further endeavors. Hence, synchronous stream ciphers are more vulnerable to active attacks.

2.1.4 Self-Synchronizing Stream Ciphers

Stream ciphers using multiples of the previous N ciphertext bits to compute the keystream belong to the category of self-synchronizing stream ciphers. The advantage of this

method is that the receiver automatically synchronizes with the keystream after receiving of N bits. This type of error handling is much better than the error handling of synchronous stream ciphers and also leads to better resistance against active attacks. Self-synchronizing stream ciphers are also susceptible to one bit errors, but here a one bit error affects not only one bit, but N bits. A frequently-used kind of self-synchronizing stream ciphers are block ciphers in cipher-feedback mode (CFB).

2.1.5 Linear Feedback Shift Registers

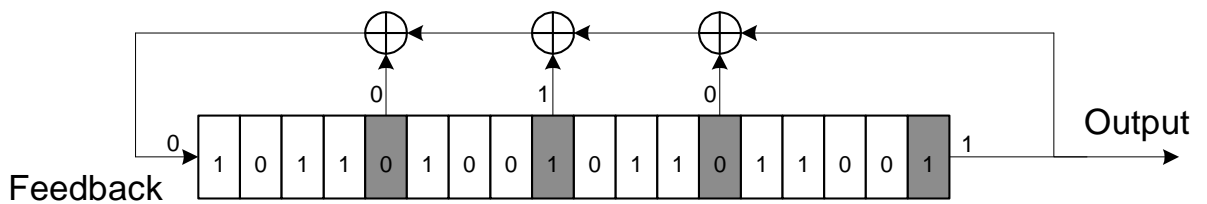


Figure 2.7: Linear Feedback Shift Register

A linear feedback shift register is a shift register whose output bit is XORed with some other bits of the register and then becomes the last bit of the LFSR again. The initial value of the LFSR is called the seed or initialization vector and leads to a deterministic sequence of bytes. If the LFSR is based on an irreducible polynomial and the seed is non-zero, the LFSR has maximum period, i.e. $2^n - 1$ (where n denotes the number of FlipFlops). The LFSR generates a linear stream of bits and therefore a LFSR cannot be directly used in a stream cipher¹.

But there are several methods to bring non-linearity into a LFSR. The first possibility is the combination of the output of at least two LFSRs to a non-linear combining function. This can be done by feeding a non-linear boolean function with the outputs of two or more LFSRs to create a combination generator.

Another way to create non-linearity is the irregular clocking of the LFSR. Normally a LFSR delivers an output at every clock signal. One attempt to break linearity is the use of two LFSRs. The first LFSR is clocked regularly and the second is only clocked if the output of the first one is a '1'. If a '0' occurs, the second LFSR repeats the previous output. To enhance the security level, this output of the second LFSR can be combined with a third LFSR. This method is called 'stop-and-go'.

Another method is called the 'shrinking generator'. Here as well two LFSRs are used, but both are clocked regularly. If the output of the first LFSR is a '1', the output of the second LFSR is taken as the output. If the first LFSR delivers a '0', there is no output

¹There exist algorithms which are able to find the polynomial of a LFSR when an output sequence of particular length is given (i.e. the Berlekamp-Massey algorithm [10])

at this state. This method is vulnerable to timing attacks against the second LFSR, but this hole can be plugged by buffering the output.

There is also a possibility to get by with only one LFSR to bring non-linearity in an LFSR. Different bits (e.g., 5 or 6) are taken from the actual state of the LFSR and serve as input to a non-linear filtering function (for instance a 6-to-1 boolean function), whose output becomes the overall output of the LFSR. This method is called nonlinear combining function.

2.2 Cryptanalysis

In the sector of cryptanalysis we only take a closer look at the two most important attacks: linear and differential cryptanalysis. These two attack methods are special mathematical attacks against block ciphers and stream ciphers. They work if a cipher runs multiple rounds to execute an encryption/decryption. When designing new block or stream ciphers the developer(s) should keep these two attacks in mind and, ideally, the ciphers should be resistant to these attacks.

2.2.1 Linear Cryptanalysis

The linear cryptanalysis is a ‘known-plaintext’ attack. This means, the attacker holds several plaintexts as well as ciphertexts and uses these pairs to attack the cipher. The idea behind this kind of attack is the linear approximation of some operations/parts of the cipher with a linear expression. In most cases a *mod 2*-operation like XOR is chosen. Let $A = \{a_1, a_2, \dots, a_m\}$ denote the input of an operation, a_i the i -th element of the input and $B = \{b_1, b_2, \dots, b_n\}$ the output of an operation, b_j the j -th element of the output. Then the expression could look like this:

$$a_{i_1} \oplus a_{i_3} \oplus a_{i_4} \oplus b_{j_2} \oplus b_{j_4} \oplus b_{j_5} = 0$$

The goal of the linear cryptanalysis is to find linear approximations like the one above, which exhibit a very low or very high probability of occurrence. In an ideal cipher the result of the formula above would be 0.5, because the number of zeros and ones is equally large. If we find now an expression where the result differs from the ideal value 0.5, for example $|0.5 - \delta|$ with bias δ , we are able to do a forecast for some parts of the cipher.

2.2.2 Differential Cryptanalysis

The differential cryptanalysis works similarly to the linear cryptanalysis, but here the point of attack is not a linear approximation, but instead high probabilities of certain

occurrences of differences of inputs and outputs of an operation (like the round function per instance). Let $X = \{x_0, x_1, \dots, x_m\}$ denote the input and $Y = \{y_0, y_1, \dots, y_n\}$ the output of an operation. Let x' and x'' be two inputs and y' and y'' the corresponding outputs. Then $\Delta(X) = (x' \oplus x'')$ is the input difference and $\Delta(Y) = (y' \oplus y'')$ the output difference. In an ideal, completely randomized cipher the probability that given an input difference $\Delta(X)$, an output difference $\Delta(Y)$ appears is 0.5^n where n denotes the bit length of the input/output. If we are able to find input differences and corresponding output differences which deviate from the probability 0.5^n , we can make predictions about the input or the output bits, whatever we are searching for.

3 Focused eSTREAM Profile I Candidates

In this chapter we provide the specifications for the implemented ciphers in this work. For comparison, we implemented the byte-oriented AES (Advanced Encryption Standard) in C, taken from Gladman [15], and a fast AES Assembly language version. Hence we describe the AES as well.

3.1 Introduction to ECRYPT/eSTREAM

ECRYPT (European Network of Excellence for Cryptology) is a European research initiative launched on February 1st, 2004. The project has an duration of four years. The objective is “to intensify the collaboration of European researchers in information security, and more in particular in cryptology and digital watermarking”.

One of the projects of ECRYPT is called eSTREAM, which aims on identifying “new stream ciphers that might become suitable for widespread adoption”. ECRYPT launched the call for papers in November 2004. This call divided the eSTREAM candidates into two groups, or better two profiles:

- Profile 1: ‘Stream ciphers for software applications with high throughput requirements.’
- Profile 2: ‘Stream ciphers for hardware applications with restricted resources such as limited storage, gate count, or power consumption.’

The timeline of the project is divided into three main parts called phases. Phase 1 started immediately after the deadline for submission in April 2005. Phase 1 aimed on general analysis of all (34) submissions with the goal of determining a subset of interesting candidates to build the Phase 2 cipher list. Phase 1 ended in February 2006 and 5 months later Phase 2 began. The candidates of Phase 2 were scrutinized using criteria like performance, resistance against improved attacks and, of course, deeper cryptanalysis.

The software candidates in Phase 2 are:

- Phase 2
 - ABC
 - CryptMT
 - DICING
 - NLS
 - Polar Bear
 - Rabbit
- Focus Phase 2
 - Dragon
 - HC-256
 - LEX
 - Phelix
 - Py
 - Salsa20
 - Sosemanuk

The current active Phase 3 started in April 2007, but when this thesis was created, Phase 2 was the actual phase. Therefore, in this work we evaluate performance aspects of the stream ciphers that were in the focus of Phase 2 and Profile 1 and not broken, yet. In detail, this work covers Dragon, HC-128, LEX, Salsa20, and Sosemanuk. Because of reported key recovery attacks we have not included Py/Pypy [25] or Phelix [24] in our testing framework.

NOTE: The ciphers that are described in the following sections are ordered alphabetically.

3.2 AES

The AES, also known as Rijndael, is a block cipher and the official predecessor of DES (Data Encryption Standard). The developers of AES are two Belgian cryptographers Joan Daemen and Vincent Rijmen. ‘Rijndael’ is a combination of the names of the inventors. The AES version we use (AES-128) is not exactly Rijndael, because Rijndael supports a larger range of key sizes, whereby AES-128 only allows use of a 128 bit key. The AES uses a block size of 128 bits and delivers an 128 bit output.

3.2.1 Workflow of AES

The workflow of the AES can be well described by the following pseudo code:

- Key expansion
- First round

- AddRoundKey()
- Encryption rounds (repeat 10 rounds)
 - SubBytes()
 - ShiftRows()
 - MixColumns()
 - AddRoundKey()
- Last round
 - SubBytes()
 - ShiftRows()
 - AddRoundKey()

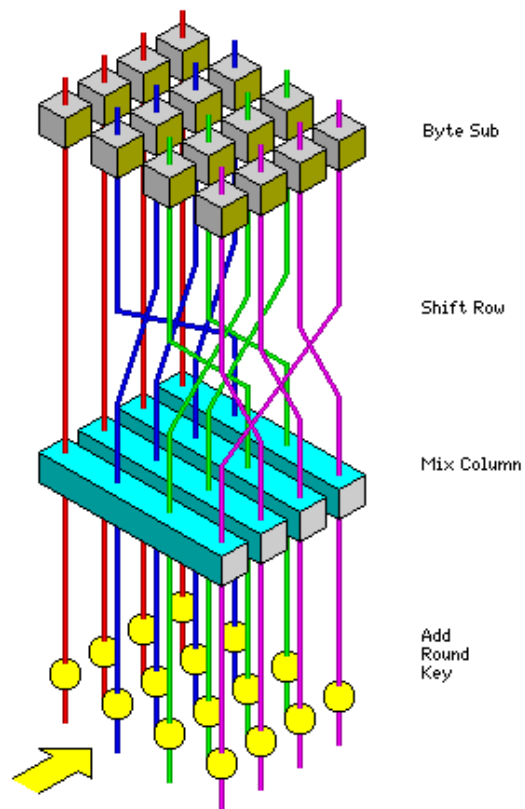


Figure 3.1: AES: Graphical round overview (taken from [16])

Figure 3.1 gives a graphical overview of one AES round.

3.2.2 Key expansion

First some declarations: $r = 10$ is the number of rounds, $b = 128$ is the block size (4 words of size 32 bit). Now let us take a look at the key expansion. The key must be extended to $(r + 1)$ subkeys. The subkeys need to have the same size as the block size. Thus the key must be expanded to the size of $b * (r + 1)$ bits. The keys will be arranged in 2-dimensional arrays with 4 rows and 4 columns, so that a single entry has a size of one byte. The first subkey is the key itself. The remaining subkeys are computed in the following manner:

To compute the first column (w_i) of the next subkey, the last column of the parent subkey (w_{i-1}) is required. The values of this column are processed by the *RotWord()* function and after that each byte of the column is replaced by the corresponding value in the S-box. This column w_{i-1} is then XORed with the column w_{i-4} . For words in positions that are a multiple of $b/\text{wordsize}$ (4,8,12,...), an additional XOR with the *rcon* table must be executed. The *rcon* table is a table containing pre-calculated values (for details on the mathematical background refer to [22] or [8]). The next three columns follow the same construction pattern but without the application of the substitution function, the table lookup in the *rcon* table, and the *RotWord()* function.

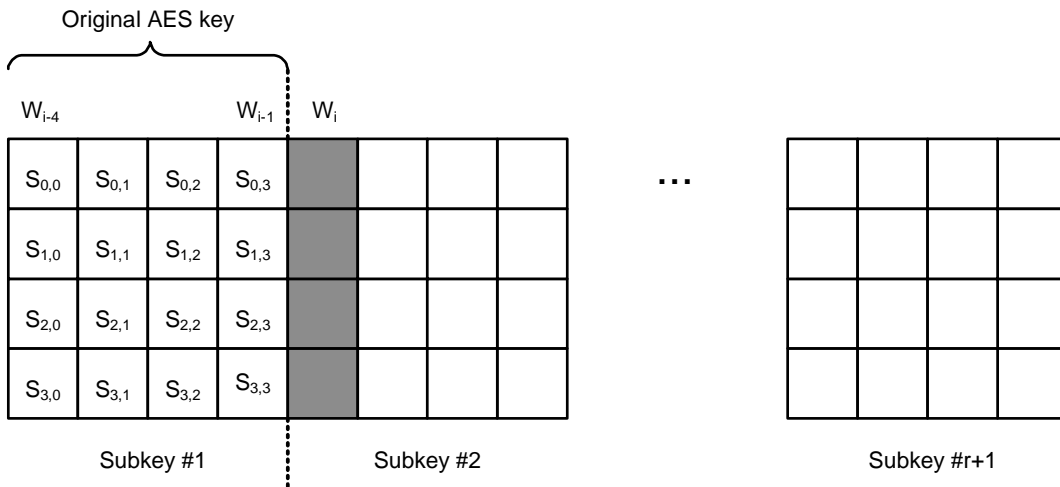


Figure 3.2: AES: Key schedule

3.2.3 AddRoundKey

The *AddRoundKey* function combines the current state with the corresponding subkey. It is a XOR between the bytes of the state and the subkey.

3.2.4 SubBytes

The SubByte function is the only non-linear function in the AES. Each byte in the (4×4) array is updated by an 8-bit S-box. This S-box is derived from the multiplicative inverse of \mathbb{F}_{2^8} with proven good non-linearity properties. The S-box consists of 16 rows and 16 columns. Each byte of the state has two digits in hexadecimal notation. For instance, the actual byte has the value '0x73' = 115. The corresponding value in the S-box can be found in the 3rd row and 7th column. In our case we find '0x8F' = 143.

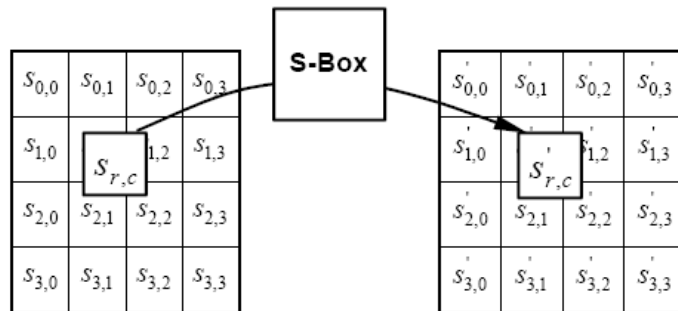


Figure 3.3: AES: SubBytes() function (taken from [8])

3.2.5 ShiftRows

The ShiftRows function affects the rows of the state. The i -th row will be rotated i byte(s) to the left.

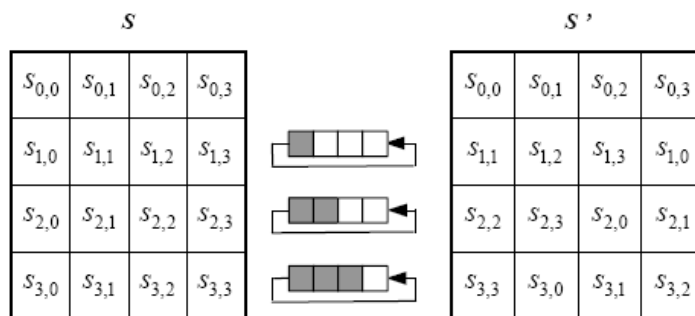


Figure 3.4: AES: ShiftRows() function (taken from [8])

3.2.6 MixColumns

In the MixColumns function the four bytes of each column are combined using an invertible linear transformation. At each step of the MixColumns function four bytes (one column) are used as input and each of these four bytes affects all four output bytes. The MixColumns function can be viewed as a matrix multiplication. Each column can be treated as a polynomial over \mathbb{F}_{2^8} , which is then multiplied with a fixed polynomial $c(x) = (3x^3 + x^2 + x + 2)$ modulo $(x^4 + 1)$.

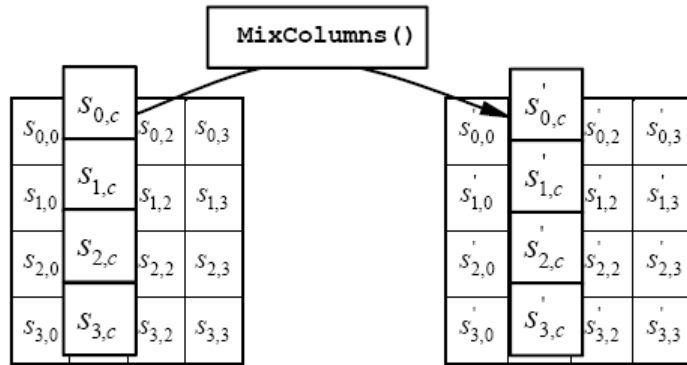


Figure 3.5: AES: MixColumns() function (taken from [8])

3.3 Dragon

Dragon is a stream cipher using a single word based non-linear feedback shift register and a non-linear filter function with memory. Its developers are K. Chen, M. Henricksen, W. Millan, J. Fuller, L. Simpson, E. Dawson, H. Lee and S. Moon. Dragon can be executed either with a key and IV-size of 128 bits or a key and IV-size of 256 bits. It produces a 64 bit keystream. The core of Dragon consists of two (8×32) -bit S-boxes and makes frequent use of its update function labeled F. Furthermore Dragon exhibits a large NLFSR (Non-Linear Feedback Shift Register) of 1024 bits and a 64-bit counter, named M.

3.3.1 The update function F

The update function of Dragon is an essential part of the cipher because it is called at two positions during one regular cycle of the whole cipher, more precisely in the key setup and during the keystream generation. It takes 192 bits as input and provides 192 bits of output, ordered in six 32 bit words.

Input = { a, b, c, d, e, f }					
Pre-mixing Layer:					
1. $b = b \oplus a;$	$d = d \oplus c;$	$f = f \oplus e;$			
2. $c = c \boxplus b;$	$e = e \boxplus d;$	$a = a \boxplus f;$			
S-box Layer:					
3. $d = d \oplus G_1(a);$	$f = f \oplus G_2(c);$	$b = b \oplus G_3(e);$			
4. $a = a \oplus H_1(b);$	$c = c \oplus H_2(d);$	$e = e \oplus H_3(f);$			
Post-mixing Layer:					
5. $d' = d \boxplus a;$	$f' = f \boxplus c;$	$b' = b \boxplus e;$			
6. $c' = c \oplus b;$	$e' = e \oplus d;$	$a' = a \oplus f;$			
Output = { a', b', c', d', e', f' }					

Figure 3.6: Dragon: Update function F (taken from [14])

In Figure 3.6 the input parameters are named a, b, c, d, e, f and the output words are denoted a', b', c', d', e' and f'. The update function makes use of six component functions overall: G_1, G_2, G_3, H_1, H_2 and H_3 (described in 3.3.2). The G and H functions are the non-linear components of the cipher. Before and after the G and H functions, several binary and modular additions are executed. The F function can be divided into three phases: the pre-mixing, the substitution (G and H functions) and the post-mixing phase. Each phase is developed in such a way that parallelization is possible. This is shown in Figure 3.7. Here a \oplus stands for an XOR and \boxplus stands for addition modulo 2^{32} .

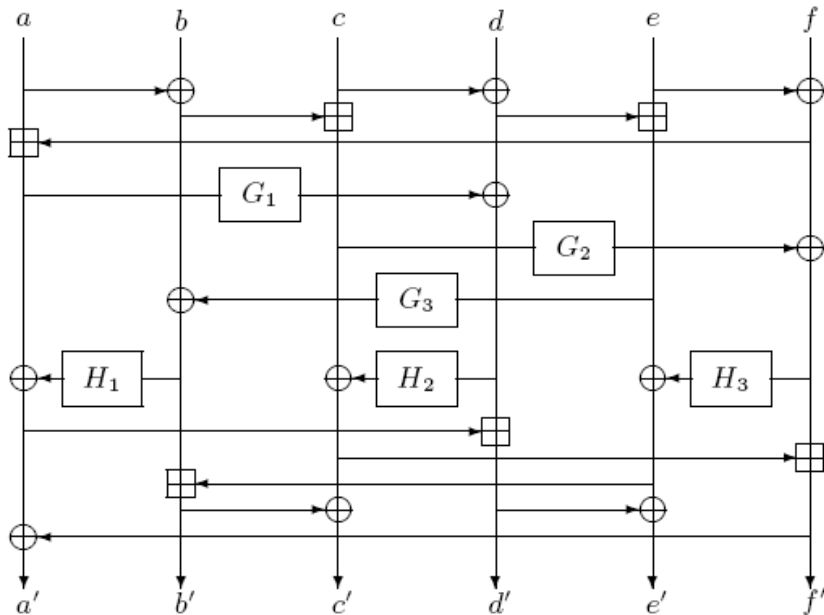


Figure 3.7: Dragon: Update function F (graphically illustrated) (taken from [14])

3.3.2 G and H functions

The G and H functions are constructed as a combination of the two (8×32) -bit S-boxes S_1 and S_2 . Hence this combination composes virtual (32×32) -bit S-boxes. The G functions use three times the S_1 S-box and one time the S_2 S-box. The H functions use three times the S_2 S-box and one time the S_1 S-box. Each byte of the 32 bit input value x is taken to determine the value at the corresponding position in the S-box ($x = x_0 || x_1 || x_2 || x_3$). The G and H functions are defined as follows:

$$\begin{aligned} G_1(x) &= S_1(x_0) \oplus S_1(x_1) \oplus S_1(x_2) \oplus S_2(x_3) \\ G_2(x) &= S_1(x_0) \oplus S_1(x_1) \oplus S_2(x_2) \oplus S_1(x_3) \\ G_3(x) &= S_1(x_0) \oplus S_2(x_1) \oplus S_1(x_2) \oplus S_1(x_3) \\ H_1(x) &= S_2(x_0) \oplus S_2(x_1) \oplus S_2(x_2) \oplus S_1(x_3) \\ H_2(x) &= S_2(x_0) \oplus S_2(x_1) \oplus S_1(x_2) \oplus S_2(x_3) \\ H_3(x) &= S_2(x_0) \oplus S_1(x_1) \oplus S_2(x_2) \oplus S_2(x_3) \end{aligned}$$

3.3.3 Key and IV initialization

<p>Input = $\{ K, IV \}$ (256-bit) Input = $\{ k, iv \}$ (128-bit)</p> <ol style="list-style-type: none"> $W_0 \dots W_7 = K K \oplus IV \overline{K \oplus IV} IV$ (256-bit) $W_0 \dots W_7 = k k' \oplus iv' iv k \oplus iv' k' k \oplus iv iv' k' \oplus iv$ (128-bit) $M = 0x0000447261676F6E$ <p>Perform steps 3-8 16 times</p> <ol style="list-style-type: none"> $a b c d = (W_0 \oplus W_6 \oplus W_7)$ $e f = M$ $\{a', b', c', d', e', f'\} = F(a, b, c, d, e, f)$ $W_0 = (a' b' c' d') \oplus W_4$ $W_i = W_{i-1}$, for $i = 7$ down to 1 (shifting the state by one word) $M = e' f'$ <p>Output = $\{ W_0 \dots W_7 \}$</p>
--

Figure 3.8: Dragon: Key initialization function (taken from [14])

The initialization of the internal state of Dragon follows a simple strategy. It's nothing but many concatenations of the (private) key and the (public) initialization vector, denoted as k and iv . The 1024 bit internal state is divided into eight 128 bit words. These words are labeled W_0 to W_7 . The F function is used in the initialization process several times. More precisely, the F function is called 16 times during initialization. Figure 3.8 shows the initialization process. Here \bar{x} denotes the complement of x , and x'

stands for a modified x whereby the upper and the lower halves of x are swapped. The authors recommend a rekeying every 2^{64} bits of generated keystream.

3.3.4 Keystream generation

Input = { $B_0 \parallel \dots \parallel B_{31}, M$ } 1. $(M_L \parallel M_R) = M$ 2. $a = B_0, b = B_9, c = B_{16}, d = B_{19}, e = B_{30} \oplus M_L, f = B_{31} \oplus M_R$ 3. $(a', b', c', d', e', f') = F(a, b, c, d, e, f)$ 4. $B_0 = b', B_1 = c'$ 5. $B_i = B_{i-2}, 2 \leq i \leq 31$ 6. $M = M + 1$ 7. $k = a' \parallel e'$ Output = { $k, B_0 \parallel \dots \parallel B_{31}, M$ }

Figure 3.9: Dragon: Keystream generation function (taken from [14])

Dragon possesses a NLFSR of 1024 bit size. This value is divided into thirty-two 32-bit values called B_0 to B_{31} . At each round six words from the internal state are taken to feed the F function. The words of the first round are taken from the positions 0, 9, 16, 19, 30 and 31 and every round these values are scaled down by 2 (mod 32). So the words of the second round are 30, 7, 14, 17, 28 and 29 and so on. Furthermore, the 64-bit memory counter M acts as a counter during keystream generation. The counter M is initialized during the initialization process. After 64 bits of keystream are generated the output is called k and B and M are updated.

3.4 HC-128

The stream cipher HC-128[23] is developed by Hongjun Wu working at the Katholieke Universiteit in Leuven, Belgium. The main part of HC-128 is made up of two secret tables with 512 32-bit entries each. At every step of the cipher one element from one of these two tables is updated by a non-linear feedback function. So within 1024 steps both tables (that means the internal state of the cipher) are updated completely. The output is a 32-bit word, which is generated at each step by the non-linear output filtering function.

3.4.1 Specification of the cipher

First we describe some operations and declarations which are necessary to understand the following details:

+	:Addition modular 2^{32}
\boxminus	:Subtraction mod $512 = 2^9$
\oplus	:Bitwise exclusive OR
\parallel	:Concatenation
\gg	:Right shift operator
\ll	:Left shift operator
\ggg	:Right rotation operator
\lll	:Left rotation operator

Table 3.1: HC-128: Operations used by HC-128

P	:an array with 512 32-bit entries. Single elements are written $P[i]$, whereby $0 \leq i \leq 511$
Q	:an array with 512 32-bit entries. Single elements are written $Q[i]$, whereby $0 \leq i \leq 511$
K	:the 128-bit secret key
IV	:the 128-bit public injection vector
s	:the generated keystream from HC-128

There exist six relevant functions in HC-128. h_1 uses the P-array as S-box, h_2 uses the Q-array.

$$\begin{aligned}
 f_1(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \ggg 3) \\
 f_2(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \ggg 10) \\
 g_1(x, y, z) &= ((x \ggg 10) \oplus (z \ggg 23)) + (y \ggg 8) \\
 g_2(x, y, z) &= ((x \lll 10) \oplus (z \lll 23)) + (y \lll 8) \\
 h_1(x) &= Q[x_0] + Q[256 + x_2] \\
 h_2(x) &= P[x_0] + P[256 + x_2]
 \end{aligned}$$

Here x is made up of 4 one byte values: $x = x_3 \parallel x_2 \parallel x_1 \parallel x_0$ whereby x_0 denotes the least significant byte.

3.4.2 Initialization process

During the initialization process the two large arrays must be initialized. This is done in several steps.

1. A new array called W_i ($0 \leq i \leq 1279$) must be filled by the expanded key and IV.

$$W_i = \begin{cases} K_i & 0 \leq i \leq 7 \\ IV_{i-8} & 8 \leq i \leq 15 \\ f_2(W_{i-2}) + W_{i-7} + f_1(W_{i-15}) + W_{i-16} + i & 16 \leq i \leq 1279 \end{cases}$$

2. After that initialization, the arrays P and Q are updated with the values in the W-array.

$$\begin{aligned} P[i] &= W_{i+256} & \text{for } 0 \leq i \leq 511 \\ Q[i] &= W_{i+768} & \text{for } 0 \leq i \leq 511 \end{aligned}$$

3. HC-128 is executed for 1024 steps and the output words replace the table entries as follows:

for $i = 0$ to 511, do

$$P[i] = (P[i] + g_1(P[i \boxplus 3], P[i \boxplus 10], P[i \boxplus 511])) \oplus h_1(P[i \boxplus 12]);$$

for $i = 0$ to 511, do

$$Q[i] = (Q[i] + g_2(Q[i \boxplus 3], Q[i \boxplus 10], Q[i \boxplus 511])) \oplus h_2(Q[i \boxplus 12]);$$

3.4.3 Keystream generation

During the keystream generation one entry of one of the arrays P or Q is updated. Each of the S-boxes is used to generate 512 bits of output and then the S-box is updated in the next 512 steps. Below, the keystream generation algorithm of HC-128 is given:

$i = 0;$

repeat until plaintext is encrypted

{

$j = i \bmod 512;$

 if $(i \bmod 1024) < 512$

 {

$$P[j] = (P[j] + g_1(P[j \boxplus 3], P[j \boxplus 10], P[j \boxplus 511]));$$

$$s_i = h_1(P[j \boxplus 12]) \oplus P[j];$$

 }

 else

 {

$$Q[j] = (Q[j] + g_2(Q[j \boxplus 3], Q[j \boxplus 10], Q[j \boxplus 511]));$$

$$s_i = h_2(Q[j \boxplus 12]) \oplus Q[j];$$

 }

}

```

    end-if
    i++
}

```

3.5 LEX

The LEX (Leak EXtraction) stream cipher is a modification of the AES Block Cipher [12] and is developed by Alex Biryukov. It uses internal states of each round to create the keystream. Amongst the researched ciphers, it is the one with the smallest output size: 16 bits. For further information regarding AES, refer to [8] or see section 3.2 above. The standard version of LEX uses a 128-bit key and a 128-bit initialization vector.

The setup phase consists of a key setup, followed by the IV initialization. The key setup of LEX is exactly the same as the key setup of AES. The IV initialization is implemented by AES-encrypting the IV under the secret key K :

$$S = AES_K(IV).$$

This value S together with the secret key K forms the internal state of LEX. S is updated every round while K remains unchanged. The developer of LEX approves to change the key every 500 AES encryptions to enhance the security level.

Figure 3.10 provides an overview of LEX.

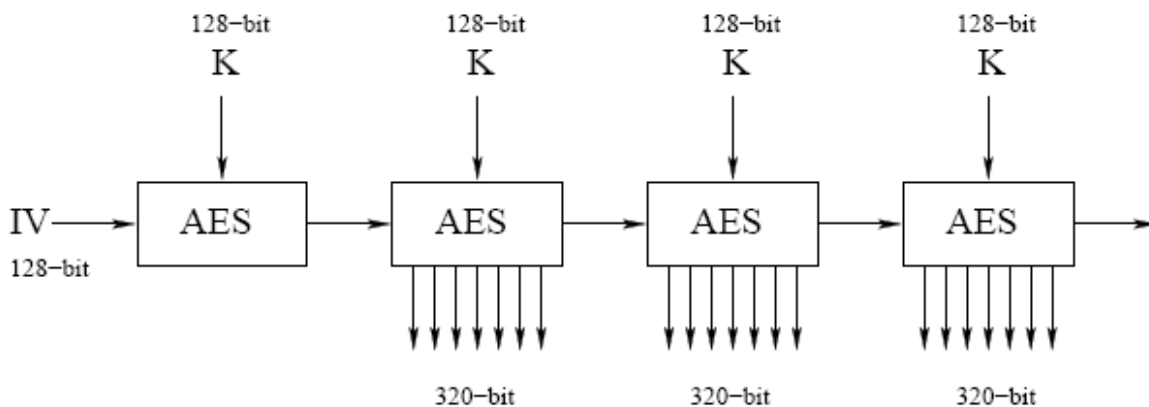


Figure 3.10: LEX: Overview (taken from [12])

The encryption of LEX is the repeated extraction of four certain bytes from the output of each of the sixteen AES rounds. Every round of AES produces 128 bit intermediate

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{0,0}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

Figure 3.11: LEX: Intermediate round values (128 bit) (taken from [12])

values. In Figure 3.11 the intermediate values are shown in a squared form and each value possesses an index, derived from its position in the square (row- and column-number).

The developer of LEX proposes to use the bytes $b_{0,0}$, $b_{2,0}$, $b_{0,2}$, $b_{2,2}$ (see figure 3.12) at every odd round and the bytes $b_{0,1}$, $b_{2,1}$, $b_{0,3}$, $b_{2,3}$ (see figure 3.13) at every even round. Only four steps are necessary to extract these 32 bits from the two 32 bit variables t_0 and t_2 :

$$out32 = ((t_0 \& 0xFF00FF) \ll 8) \oplus (t_2 \& FF00FF).$$

In the formula above t_i is a row of four bytes: $t_i = (b_{i,0}, b_{i,1}, b_{i,2}, b_{i,3})$. There is no need to use any filter function before the bytes are sent to the output.

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{0,0}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

Figure 3.12: LEX: Extraction on odd rounds (taken from [12])

NOTE: We encountered a discrepancy between the specification and the implementation of LEX. In the specification Biryukov suggests to use the bytes $b_{0,0}$, $b_{2,0}$, $b_{0,2}$, and $b_{2,2}$ for extraction in odd rounds and the bytes $b_{0,1}$, $b_{2,1}$, $b_{0,3}$, and $b_{2,3}$ for extraction in even rounds. In contrast to the specification, the implementation indicates the bytes $b_{1,1}$, $b_{3,1}$, $b_{1,3}$, and $b_{3,3}$ for extraction in even rounds. The test vectors fit to the bytes extracted by the implementation. An email (sent to Biryukov) including a request to clarify the situation stayed unanswered.

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{0,0}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

Figure 3.13: LEX: Extraction on even rounds (taken from [12])

3.6 Salsa20

Salsa20 is a stream cipher developed by Daniel J. Bernstein. To be accurate, the cipher is actually a hash algorithm (hashing of the key, nonce and a block number) with a subsequent XOR of the hash value and the plaintext. Salsa20 takes 64 bytes as input, produces 64 bytes of output and operates on 32-bit words. Only three basic operations are needed by the different functions in Salsa20: addition mod 2^{32} (denoted as $+$), exclusive OR of two words (denoted as \oplus) and left rotation of word x by n bytes (denoted as $x \lll n$).

3.6.1 The quarterround function

The most important part of Salsa20 is the quarterround function. Nearly all subsequent functions use the quarterround function, and consist simply of multiple calls of the quarterround function. The function takes 4 words as input and delivers 4 words as output. If we name the input $y = (y_0, y_1, y_2, y_3)$ and the output is designated as $z = (z_0, z_1, z_2, z_3)$, then the quarterround function can be defined as follows:

$$\begin{aligned}
 z_1 &= y_1 \oplus ((y_0 + y_3) \lll 7) \\
 z_2 &= y_2 \oplus ((z_1 + y_0) \lll 9) \\
 z_3 &= y_3 \oplus ((z_2 + z_1) \lll 13) \\
 z_0 &= y_0 \oplus ((z_3 + z_2) \lll 15)
 \end{aligned}$$

Table 3.2: Salsa20: The quarterround function

3.6.2 The rowround function

The rowround function takes 16 bytes as input and delivers 16 bytes as output. The input is called $y = (y_0, y_1, \dots, y_{15})$, the output is named $z = (z_0, z_1, \dots, z_{15})$ and the function is defined as:

$$\begin{aligned}
(z_0, z_1, z_2, z_3) &= \text{quarterround}(y_0, y_1, y_2, y_3) \\
(z_5, z_6, z_7, z_4) &= \text{quarterround}(y_5, y_6, y_7, y_4) \\
(z_{10}, z_{11}, z_8, z_9) &= \text{quarterround}(y_{10}, y_{11}, y_8, y_9) \\
(z_{15}, z_{12}, z_{13}, z_{14}) &= \text{quarterround}(y_{15}, y_{12}, y_{13}, y_{14})
\end{aligned}$$

Table 3.3: Salsa20: The rowround function

3.6.3 The columnround function

The columnround function takes 16 bytes as input and delivers 16 bytes as output. The input is called $\mathbf{x} = (x_0, x_1, \dots, x_{15})$, the output is named $\mathbf{z} = (z_0, z_1, \dots, z_{15})$ and the function is defined as:

$$\begin{aligned}
(z_0, z_4, z_8, z_{12}) &= \text{quarterround}(x_0, x_4, x_8, x_{12}) \\
(z_5, z_9, z_{13}, z_1) &= \text{quarterround}(x_5, x_9, x_{13}, x_1) \\
(z_{10}, z_{14}, z_2, z_6) &= \text{quarterround}(x_{10}, x_{14}, x_2, x_6) \\
(z_{15}, z_3, z_7, z_{11}) &= \text{quarterround}(x_{15}, x_3, x_7, x_{11})
\end{aligned}$$

Table 3.4: Salsa20: The columnround function

3.6.4 The doubleround function

The doubleround function takes 16 words as input and delivers 16 words as output. The doubleround function first calls the column round function followed by a call of the rowround function.

$$\text{doubleround}(\mathbf{x}) = \text{rowround}(\text{columnround}(\mathbf{x}))$$

3.6.5 The littleendian function

The littleendian functions simply swaps the byteorder of a 4-byte word. If $b = (b_0, b_1, b_2, b_3)$ then $\text{littleendian}(b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3 = (b_3, b_2, b_1, b_0)$.

3.6.6 The Salsa20 hash function

A short introduction to hash functions:

Hash functions use reproducible methods to shrink data of arbitrary length to a fixed length. Hash functions must fulfill 3 important properties:

- Preimage resistant: Given a hash value h it should be very difficult to find the corresponding message m , such that $h = \text{hash}(m)$.
- Second preimage resistant: Given a message m_1 it should be hard to find a message m_2 (different from m_1) with $\text{hash}(m_1) = \text{hash}(m_2)$.
- Collision resistant: Given a $\text{hash}(m_1)$ it should be hard to find a message m_2 such that $\text{hash}(m_1) = \text{hash}(m_2)$.

The Salsa20 hash function takes 64 bytes as input and delivers a 64-byte sequence as output.

$$\text{Salsa20}(x) = x + \text{doubleround}^{10}(x).$$

Let us denote x as $(x[0], x[1], \dots, x[63])$ then

$$\begin{aligned} x_0 &= \text{littleendian}(x[0], x[1], x[2], x[3]) \\ x_1 &= \text{littleendian}(x[4], x[5], x[6], x[7]) \\ x_2 &= \text{littleendian}(x[8], x[9], x[10], x[11]) \\ &\vdots \\ x_{15} &= \text{littleendian}(x[60], x[61], x[62], x[63]) \end{aligned}$$

Define $z = (z_0, z_1, \dots, z_{15}) = \text{doubleround}^{10}((x_0, x_1, \dots, x_{15}))$. Then $\text{Salsa20}(x)$ is the concatenation of:

$$\begin{aligned} &\text{littleendian}^{-1}(z_0 + x_0) \\ &\text{littleendian}^{-1}(z_1 + x_1) \\ &\text{littleendian}^{-1}(z_2 + x_2) \\ &\vdots \\ &\text{littleendian}^{-1}(z_{15} + x_{15}) \end{aligned}$$

3.6.7 The expansion function

The input k of this function can be a 32-byte or a 16-byte sequence, the input n must be a 16-byte value. The output of $\text{Salsa20}_k(n)$ is then a 64-byte value.

Define $\sigma_0, \sigma_1, \sigma_2$ and σ_3 as 4-byte values, k_0, k_1 and n as 16-byte values then $\text{Salsa20}_{k_1, k_2}(n) = \text{Salsa20}(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3)$. Define τ_0, τ_1, τ_2 and τ_3 as 4-byte values, k and n as 16-byte values then $\text{Salsa20}_k(n) = \text{Salsa20}(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3)$.

3.6.8 The encryption function

Define

$$\begin{aligned} k &= \text{a 16-byte or 32-byte value (secret key),} \\ v &= \text{a 8-byte value (public initialization vector),} \\ m &= \text{an l-byte sequence (plaintext message),} \end{aligned}$$

then $\text{Salsa20}_k(v) \oplus m$ is the corresponding encrypted ciphertext message. In the opposite case m is the ciphertext and $\text{Salsa20}_k(v) \oplus m$ is the decrypted plaintext message.

3.7 Sosemanuk

The Sosemanuk [9] stream cipher uses basic design principles from the stream cipher Snow 2.0 [17] and the partly modified block cipher Serpent [19] and is developed by C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin and H. Sibert. It is based on 32-bit words and uses a 128-bit size for the key and the IV. The cipher consists of a 10-word LFSR and a 2-word FSM. Sosemanuk could be regarded as an improvement of Snow 2.0, because it uses a faster IV Setup and also requires a reduced amount of static data which leads to a better performance and a gain in security. The key length is variable between 128 and 256 bit, but a larger key (≥ 128 bit) is no warranty for an increase of security. The key setup procedure of Sosemanuk is based on a modified version of Serpent where only 24 rounds are executed instead of 32 in the original cipher.

3.7.1 Serpent and its derivatives

Serpent is a block cipher and was a candidate for the AES. Serpent operates on 128 bit blocks which are split into four 32 bit blocks (in Little Endian (LE) mode). There are two functions derived from the original cipher called *Serpent1* and *Serpent24*.

While a Serpent round consists of a subkey addition (bitwise exclusive OR), a S-box application and a linear bijective transformation, the Serpent1 function only consists of the S-box application. More precisely, the third S-box S_2 of Serpent is used. Serpent1 needs four 32 bit words as input and delivers four 32 bit words as output.

Instead of the 32 rounds of the full Serpent version, Serpent24 is reduced to 24 rounds. The 24th round of Serpent4 is equivalent to the 32th round of Serpent. The only difference is the fact that Serpent24 contains the linear transformation and the XOR (instead of the 32th and 33th subkey, the 24th and 25th are used). Serpent24 needs only the first 25 subkeys of the Serpent key schedule. Expressed as a formula:

$$R_{23}(X) = L(\hat{S}_{23}(X \oplus \hat{K}_{23})) \oplus \hat{K}_{24}$$

3.7.2 The LFSR

Now a short description of the underlying finite field of the LFSR is given. Most of the internal state of Sosemanuk is held in a LFSR that contains ten elements of $\mathbb{F}_{2^{32}}$. This is the field with 2^{32} elements. The representation of the elements of $\mathbb{F}_{2^{32}}$ is exactly the same as in Snow 2.0. Here is a short summary.

\mathbb{F}_2 is the finite field with 2 elements. Let β be a root of the primitive polynomial:

$$Q(X) = X^8 + X^7 + X^5 + X^3 + 1$$

in \mathbb{F}_2 . Then \mathbb{F}_{2^8} is the quotient $\mathbb{F}_2[X]/Q(X)$. That means each element of \mathbb{F}_{2^8} can be illustrated as β^k for some integers k ($0 \leq k \leq 254$). So any element in \mathbb{F}_{2^8} can be identified with an 8-bit integer on the basis of the following bijection:

$$\begin{aligned} \phi: \quad \mathbb{F}_{2^8} &\rightarrow 0, 1, \dots, 255 \\ x = \sum_{i=0}^7 x_i \beta^i &\mapsto x = \sum_{i=0}^7 x_i 2^i \end{aligned}$$

where each x_i is either 0 or 1. Now let α be a root of the primitive polynomial:

$$P(X) = X^4 + \beta^{23}X^3 + \beta^{245}X^2 + \beta^{48}X^1 + \beta^{239}$$

on $\mathbb{F}_{2^8}[X]$. The field $\mathbb{F}_{2^{32}}$ is now defined as the quotient $\mathbb{F}_{2^8}[X]/P(X)$.

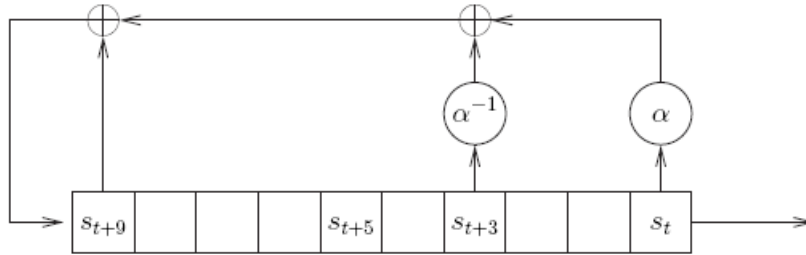


Figure 3.14: Sosemanuk: The LFSR of Sosemanuk (taken from [9])

This section is important because the LFSR operates over these elements of $\mathbb{F}_{2^{32}}$. At time $t = 0$ the LFSR is filled with the values s_1 to s_{10} . That is called the initial state. From now on, each new value is computed with the following recurrence formula:

$$s_{t+10} = s_{t+9} \oplus \alpha^{-1}s_{t+3} \oplus \alpha s_t, \quad \forall t \geq 1$$

After that the register is shifted. The LFSR possesses the following feedback polynomial:

$$\pi(X) = \alpha X^{10} + \alpha^{-1} X^7 + X + 1 \in \mathbb{F}_{2^{32}}[X]$$

3.7.3 The FSM

The FSM consists of two 32-bit registers R1 and R2 which realize a 64 bit memory. At every call, it takes as input some words from the LFSR and then updates the memory and delivers 32 bits as output. For time $t \geq 1$ the FSM works as follows on the LFSR state:

$$FSM_t : (R1_{t-1}, R2_{t-1}, S_{t+1}, S_{t+8}, S_{t+9}) \mapsto (R1_t, R2_t, f_t).$$

The single operations are defined as follows:

$$\begin{aligned} R1_t &= (R2_{t-1} + \text{mux}(\text{lsb}(R1_{t-1}), s_{t+1}, s_{t+1} \oplus s_{t+8})) \bmod 2^{32} \\ R2_t &= \text{Trans}(R1_{t-1}) \\ f_t &= (s_{t+9} + R1_t \bmod 2^{32}) \oplus R2_t \end{aligned}$$

where $\text{mux}(c, x, y)$ is the same as: ‘choose x if $c = 0$, choose y if $c = 1$ ’ and $\text{lsb}(x)$ denotes the least significant bit of x. The Trans function on $\mathbb{F}_{2^{32}}$ is defined by:

$$\text{Trans}(z) = (M \times z \bmod 2^{32}) \lll 7$$

at which M is a given constant with the value 0x54655307. These are the first ten decimals of the constant π in hexadecimal notation. \lll stands for a bitwise rotation of a 32 bit value (in the case above: by 7 bits).

The last 4 outputs of the FSM are grouped and Serpent1 is applied to these groups. This groups will be combined (by XOR) with the corresponding output values from the LFSR to produce the final output bytes:

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = \text{Serpent1}(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t)$$

3.7.4 Key Setup

The key setup of Sosemanuk is similar to the Serpent24 key schedule. It produces 25 subkeys with a size of 128 bit, that are 100 32-bit words. The key length may vary from 1 to 256 bits but because Sosemanuk aims at 128-bit security a key length smaller than 128 bit is not allowed. Adding further bits to the 128 bit key is possible up to the maximum, but using a longer key does not necessarily provide more security.

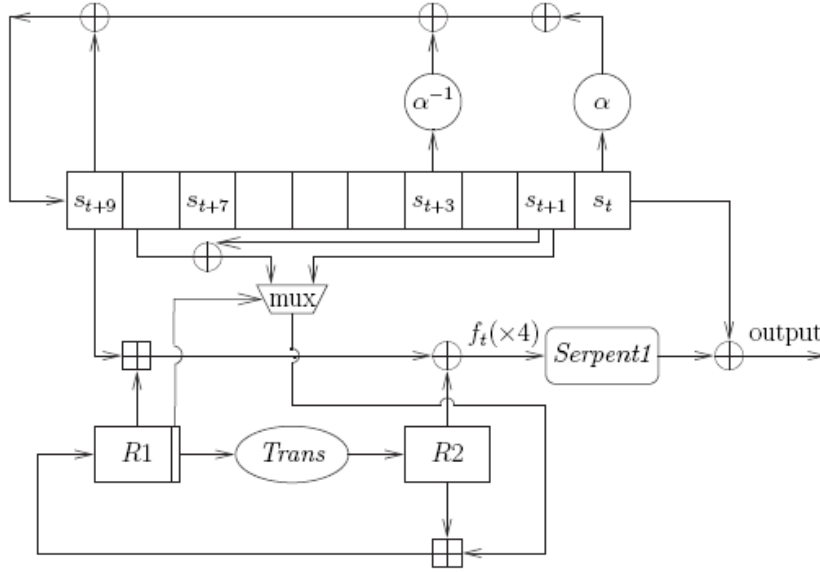


Figure 3.15: Sosemanuk: Overview (taken from [9])

3.7.5 IV Injection

The IV has a size of 128 bits. This value is used as input to the SERPENT24 function, which is already initialized by the key schedule. For the IV injection only the outputs of the 12th, 18th and 24th round are used. These values are marked as follows:

- The output of the 12th round: $(Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12})$
- The output of the 18th round: $(Y_3^{18}, Y_2^{18}, Y_1^{18}, Y_0^{18})$
- The output of the 24th round: $(Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24})$

These values initialize the internal state of SOSEMANUK in the following manner:

$$\begin{aligned}
 (s_7, s_8, s_9, s_{10}) &= (Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12}) \\
 (s_5, s_6) &= (Y_1^{18}, Y_3^{18}) \\
 (s_1, s_2, s_3, s_4) &= (Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24}) \\
 R1_0 &= Y_0^{18} \\
 R2_0 &= Y_2^{18}
 \end{aligned}$$

4 Microcontroller

4.1 Microcontroller



Nowadays in almost every home you will find at least one or more personal computer systems. But it is likely that you can find ten times more embedded systems in your own household as you possess personal computers. For instance, microcontrollers can be found in nearly every electrical device including washing machines, microwaves, telephones, toasters, digicams, and so on. In a typical mid-range automobile, 50 or more microcontrollers are installed to control, monitor and regulate the settings of the motor or other devices. In the year 2002 only 2% of all worldwide produced micro computers were attached to general purpose computer systems, the remaining 98% were used in embedded applications.

A microcontroller can be described as a 'computer on a chip'. All necessary components are included on one single chip: the processor (the CPU), non-volatile memory for the program (ROM, EPROM, EEPROM or flash), volatile memory for input and output (RAM), a clock generator (quartz timing crystal), and an I/O control unit. Microcontrollers do not need an external address or a data bus, because all components are integrated on the same chip as the CPU. Microcontrollers are very restricted in resources like SRAM and flash memory.

If the microprocessor is dedicated to performing one or a few special tasks, the whole system is called an embedded system. An embedded system does not possess the usual I/O units like a keyboard, a screen, or a printer. It runs in a restricted environment which is optimized according to the requirements that are to be fulfilled.

4.2 Atmel



The Atmel Corporation is a producer of semiconductors with about 8,000 employees. Founded in 1984 by George Perlegos, it has grown to an industry leader in security systems like smart cards. Atmel produces microcontrollers, radio frequency devices, EEPROM, flash memory, and scores of application-specific devices. Amongst other things Atmel produces its own AVR architecture and the ATmega series. Atmel holds a great intellectual property library with over 1,300 analog and digital patents and on account of this it is able to provide electronics systems manufacturers with complete system solutions. Annual revenue in the year 2005 aggregates to \$ 1,676 million.

4.3 ATmega Series

An unproven rumor says that the abbreviation AVR stands for Alf-Egil Bogen and Vegard Wollan RISC (Reduced Instruction Set Computer), the two founders of Atmel Norway. Some others say the acronym stands for Advanced Virtual RISC, but Atmel itself says that the name AVR is not an acronym and has no further meaning.

AVR microprocessors are a family of 8-bit RISC microcontrollers. The individual device classes differ in SRAM and flash memory size, as listed in Table 4.1. Its memory is organized as a Harvard architecture with a 16-bit word program memory and an 8-bit word data memory. Most of the microcontroller's instructions are one-cycle. All of the microcontrollers listed in Table 4.1 can be clocked at up to 16 MHz.

Due to its easy usage, its low power consumption, and its comparatively low price, the AVR microcontrollers have reached a high popularity in embedded system design.

Table 4.1: Specification of the most popular AVR devices (ATmega family)

Device	Flash [kbyte]	SRAM [byte]
ATmega8	8	1024
ATmega16	16	1024
ATmega32	32	2048
ATmega64	64	4096
ATmega128	128	4096
ATmega1281	128	8192

The AVR microcontrollers do not require external memory because all memory types (flash memory, EEPROM and SRAM) are located on the same chip as the CPU. The non-volatile flash memory is used to store the program instructions. Each instruction takes 16 bits for execution, divided into an 8-bit opcode followed by 8 bits of data or an address (although all devices of the ATmega series are 8-bit microcontrollers). The most important characteristics of the AVR family are:

- 130 to 135 instructions (most of them require only 1 cycle),
- 32 general purpose working 8-bit registers (3 of them could be used as 16-bit pointers),
- 8k to 128k bytes of in-system self-programmable flash memory with an average endurance of 10k write/erase cycles,
- 0.5k to 4k bytes of EEPROM with an average endurance of 100k write/erase cycles,
- 1k to 128k bytes of internal SRAM,
- 23 to 54 programmable I/O lines,
- very small power consumption (for example an ATmega16 at 1MHz, 3V, 25°C):
 - in active state: 1.1 mA,
 - in idle state: 0.35mA,
 - in power-down state: $< 1\mu\text{A}$.

5 Framework Set-Up and Tool Chain

5.1 Porting to AVR Microcontrollers

Initially we simply ported the published eSTREAM API implementations into a C-version that runs on one of the devices of the ATmega series. Afterwards we implemented these versions in Assembly language. The eSTREAM ciphers come with a set of associated files according to the eSTREAM API. In order to reduce the size of the code and to solve the dependencies we move only the parts of each file that are required for execution into one *cipher-avr.c* file. One problem in porting code to an AVR microcontroller is the limited amount of SRAM. A solution for saving valuable SRAM lies in moving S-boxes or comparable big static data arrays into flash memory. We accomplish that by using the *progmem* construction. It is first required to include the necessary .h-file by writing `#include <avr/pgmspace.h>` at the beginning of the .c-file. Subsequently the static tables can be saved in flash memory using the *PROGMEM* command. For instance, in the case of the LEX cipher this may look like this:

```
static const u32 Te0[256] PROGMEM = {  
2   0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU, ...
```

Reading of the values (in this case 32-bit) can be done by using the *pgm_read_dword* command. To simplify the usage of this command and to spare paperwork we defined a macro called RFF (Read From Flash):

```
/* Special define cause of progmem */  
2 #define RFF(v) pgm_read_dword(&v) /* RFF stands for ReadFromFlash */
```

Another issue is the different integer variable size in a 32-bit-oriented environment and the 8-bit-oriented environment of an AVR. Thus all variables used have to be adapted to the standard integer sizes of the AVR microcontroller. All ciphers use the *u8-u32* definitions. So we are able to force the using of the AVR standard integer sizes by redeclaring the *u*-types:

```
#define u8 uint8_t  
2 #define u16 uint16_t  
#define u32 uint32_t
```

Additionally the file *stdint.h* must be included to enable the compiler to understand what `uintX_t` is.

5.2 Development Tools

For the software development we used the tools ‘WinAVR’ [5] and ‘AVR Studio 4’ [3], which are described in the following sections.

5.2.1 WinAVR

WinAVR is a suite of executable, open source software development tools for the Atmel [1] AVR series of RISC microprocessors hosted on the Windows platform. WinAVR contains `avr-gcc` (compiler), `avrdude` (programmer), `avr-gdb` (debugger) and a tool for automatic makefile generation. There is also included an editor named ‘Programmers Notepad’ (PN). The version of PN, included in the WinAVR suite, comes with a built-in to compile the C-code by pressing a hot-key. But before this can be done, the gcc-compiler needs a makefile with specified instructions and options to compile the C-file. This can be easily done with the makefile generator Mfile, developed by Jörg Wunsch [4].

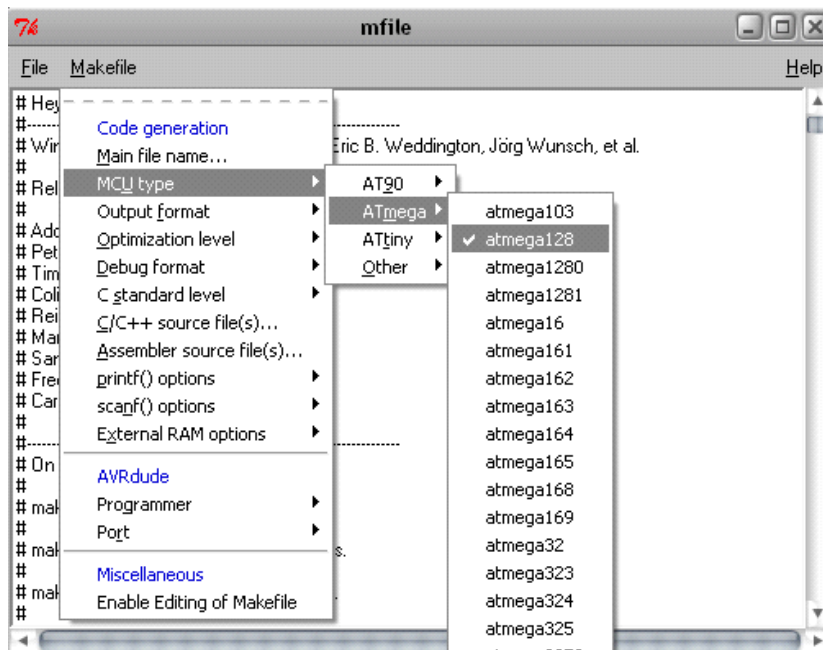


Figure 5.1: Snapshot of Mfile

Mfile comes with a very good standard template for the makefile. One merely has to make some important settings as shown in the following listing:

```
# MCU name
2 MCU = atmega128
# Target file name (without extension).
```

```

4 TARGET = dragon-avr
# List C++ source files here. (C dependencies are automatically generated.)
6 CPPSRC = dragon-avr.cpp

```

NOTE: Although the *CPPSRC* directive must be defined, the corresponding C++ file does not actually need to exist. The name of the target file must be written in lower case. If not done in this way, the compiler does not work and displays error messages.

There is another very useful tool named *avrsizesx* [2]. This tool shows a short summary of the used memory of flash memory, SRAM and EEPROM in percentages. In the case of the Dragon cipher, additional to

```

Size after:
2 dragon-avr.elf :
section      size      addr
4 .data        40        8388864
  .text       57434         0
6 .bss         384        8388904
  .noinit      0        8389288
8 .eeprom      0        8454144
  .stab        42348         0
10 .stabstr     3364         0
Total      103570

```

we get the following output:

```

Flash      SRAM      EEPROM
2  -----
  46%      10%      0%

```

Last but not least, we have to do a further little tweak to have the ability to watch the separate parts of structs in AVRStudio later on.

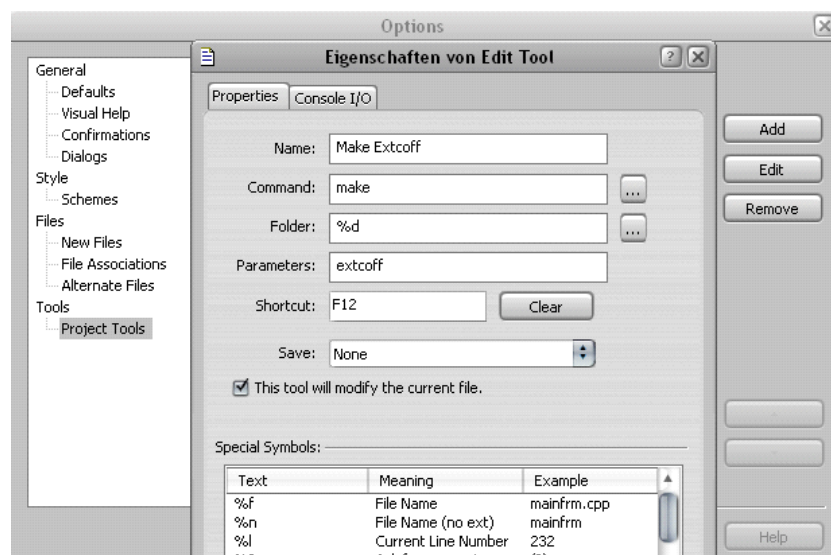


Figure 5.2: Generation of project tool “extended coff”

To accomplish this we have to add a project tool with special parameters to PN, which creates an *extended coff*-file [20]. This file format writes additional debug information in the output file, considerably more than can be found in a *elf*-file [21].

Once the code compiles without errors in PN, we use the output file (extcoff) from WinAVR to execute the code in ‘AVR Studio 4’ and simulate it on the chosen AVR device.

NOTE: It must be pointed out that SRAM size as provided by WinAVR includes only static variables that are initialized at the beginning. Because of this, WinAVR returns a smaller value than the actual required SRAM size. In particular, the cipher specific structure `ECRYPT_ctx` is not included in this value, because it is initialized during runtime and is non-static. To provide a better statement on actual SRAM size the byte size of the cipher specific structure `ECRYPT_ctx` is calculated manually and added to the size of the static variables.

5.2.2 AVRStudio

AVR Studio 4 is an Integrated Development Environment (IDE) for writing and debugging AVR applications on the Windows platform. We are able to use all the functions familiar from common debugging tools such as watching registers and variables. At every state we can obtain the number of CPU cycle counts, which enables us to measure clock cycles for benchmarking throughput. For the implementations in C-language, AVR Studio 4 is only used to simulate the AVR device. For the Assembly implementations, AVR Studio 4 is also used as development environment. The Assembly code is AVR Assembly code, not gcc-Assembly code, but can be easily transformed into it.

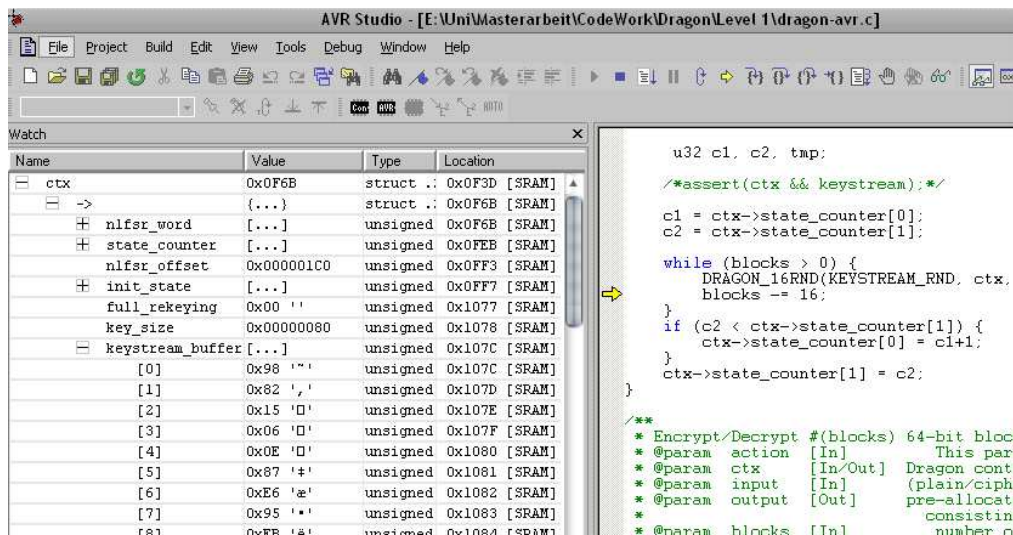


Figure 5.3: Snapshot of AVR Studio 4 - watching the struct `ctx` of Dragon

As we use the extended coff files to simulate the ciphers on the AVR devices, we can observe the values of parts of a struct. This is shown in Figure 5.3.

5.3 Configuration for Testing

Here we explain the general structure of the performance evaluation based on the ECRYPT API functions for both cases, C and Assembly language. The exact configuration will be shown in the chapters describing the implementation of the ciphers.

5.3.1 C language configuration

The test sequence in C language is the following (all data in pseudo code):

```
ECRYPT_init()
ECRYPT_keysetup(key)
ECRYPT_ivsetup(iv)
ECRYPT_process_bytes(encryption, blocksize)
ECRYPT_ivsetup(iv)
ECRYPT_process_bytes(decryption, blocksize)
```

The parameter `blocksize` is adapted for each cipher. After each call of a function the CPU cycles needed are recorded by using AVR Studio 4. This configuration is very close to the one in the eSTREAM API. We encrypt one block of size `blocksize` under the key `key`, the IV `iv` and a plaintext with solely zeros. After encryption we do a new keysetup and decrypt the created ciphertext. If we get the (empty) plaintext, the test succeeds.

5.3.2 Assembly language configuration

The common configuration in Assembly language looks like this:

```
call PREINIT
call INIT
call KEYSETUP
call IVSETUP
call ENCRYPT
call END
```

In the `PREINIT` phase we initialize the stack pointer and define the start address of the SRAM. After that, in the `INIT` phase we write the *key*, *IV* and *plaintext* in the flash memory. Key setup, IV setup and encryption are the same as in the C configuration. Because a microcontroller is a finite state machine (FSM), the algorithm ends in an endless loop, jumping all the time to the marker `END`.

6 Results in C

6.1 Objectives

The final objective of this master thesis is the efficient implementation of stream ciphers on embedded 8-bit AVR microcontrollers. The first obvious step is logically to determine whether or not it is even possible to get the ciphers running on a device of the ATmega family. This chapter describes our approach to create an executable version of the focused ciphers in C language.

6.2 Implementation

In this section we describe the configuration/structure of the focused ciphers. We will not describe the assembling of the *cipher-avr.c* file unless significant changes are made. When that is the case, the information can be found in the following sections.

6.2.1 AES

Our version of the AES is the byte oriented AES implementation developed by Brian Gladman, complemented by the ability to perform an encryption in CBC mode. Otherwise the IV has been unused. Outside this small change, the rest of the cipher is untouched. For information referring to CBC modus please see Chapter 2.1.1. In CBC modus, before encryption is done, the plaintext is XORed with the IV, respectively the last encrypted ciphertext block.

```
void xor16(u8 value1[], u8 value2[], u8 offset1, u8 offset2) {  
2   for(u8 j=0;j<16;j++) {  
       value1[offset1 + j] ^= value2[offset2 + j];  
4   }  
}
```

We do this by using a small function called *xor16()* which takes 4 parameters as input: an 8-bit array with the name *value1*, a second 8-bit array with the name *value2* and two 8-bit values called *offset1* and *offset2*. The function builds the XOR conjunction of *value1* (beginning at *offset1*) and *value2* (beginning at *offset2*) for the next 16 bytes.

The speed measurement is done by a very simple main function which does the encryption of one plaintext block and afterwards a decryption of the encrypted block in CBC modus.

```

1 u8 input[16] = {16*0};
2 u8 output[16] = {16*0};
3 u8 key[16] = {16*0};
4 u8 iv[16] = {16*0};
5 u8 o_key[16] = {16*0};
6 u8 keysize = 128;

8 int main(void){
    aes_context ctx;
10 aes_set_key(key, keysize, &ctx); // key setup
    xor16(input, iv, 0, 0); // IV setup
12 aes_encrypt(input, output, &ctx); // encryption
    xor16(output, iv, 0, 0); // IV setup
14 aes_decrypt(output, input, &ctx); // decryption
    return 0;
16 }

```

If more than 1 block should be encrypted, the main function must be slightly modified:

```

int main(void){
2 aes_context ctx;
  u32 count_blocks = 24;
4 for(u8 i=0;i<count_blocks;i++){
    if (i==0) xor16(input, iv, 0, 0);
6    else xor16(input, output, (i*16), ((i-1) * 16));
    aes_set_key(key, keysize, &ctx);
8    aes_encrypt(input + (i*16), output + (i*16), &ctx);
  }
10 return 0;
}

```

6.2.2 Dragon

Dragon can be made executable on an ATmega device with a modicum of effort. One merely has to make some small changes like adapting the used variables to the standard integer variables used by the microprocessor, and storing big static data arrays in the flash memory. This can easily be done by adding the following three lines at the top of the C file:

```

#define u8 uint8_t
2 #define u16 uint16_t
#define u32 uint32_t

```

All ciphers already use the identifiers *u8*, *u16*, and *u32* but they are defined to be platform-dependent in the file 'encrypt-config.h'. We do not include this file and add the three lines above instead. So the cipher variables can stay untouched and work well for us anyway.

As mentioned above, the S-boxes called *sbox1* and *sbox2* must be written to the flash memory to save space in the SRAM. Corresponding to Chapter 5.1 this is accomplished by the following two small lines:

```
static const u32 sbox1[256] PROGMEM = { ... };
2 static const u32 sbox2[256] PROGMEM = { ... };
```

The loading of the values stored by this method works as specified in Chapter 5.1.

Now we have to create a small main function processing only a minimal encryption, followed by a decryption of the encrypted ciphertext. This minimal main function and the necessary variable declarations and initializations to run Dragon are listed below:

```
u8 input[128] = {128*0};
u8 output[128] = {128*0};
u8 keystream[128] = {128*0};
4 u8 key[16] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA,
               0xBB, 0xCC, 0xDD, 0xEE, 0xFF };
u8 iv[16] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA,
             0xBB, 0xCC, 0xDD, 0xEE, 0xFF };
6 u32 keysize = 128;
u32 ivsize = 128;
8
int main(void){
10  ECRYPT_ctx ctx;
   ECRYPT_keysetup(&ctx, key, keysize, ivsize); // key setup
12  ECRYPT_ivsetup(&ctx, iv); // IV setup
   ECRYPT_process_bytes(0, &ctx, input, output, 128); // encryption
14  ECRYPT_ivsetup(&ctx, iv); // IV setup
   ECRYPT_process_bytes(1, &ctx, output, input, 128); // decryption
16  return 0;
}
```

NOTE: We discovered some problems while trying to compile the optimized reference code. The resulting keystream begins with ‘7D7256A3....’ using ‘0x000011...667777’ as key vector instead of ‘99B3AA14...’. We compared the baseline reference code with the optimized reference code and found out a difference in the use of the endianess macros. In *dragon-ref.c* only ‘..._BIG()’ macros are used. In *dragon-opt.c* only ‘..._LITTLE()’ macros are used instead. Hence we replaced the appropriate appearances of the ‘..._LITTLE()’ macros with the ‘..._BIG()’ macros. Afterwards we got the correct keystream but in wrong endianess as outcome (‘14AAB3992FD03BB6...’). We solved this problem by modifying the following lines in the KEYSTREAM_RND and the PROCESS_RND macros (in *dragon-opt.c*):

In KEYSTREAM_RND replace

```
*(out++) = a ^ (f + c); \
2 *(out++) = e ^ (d + a);
```

with

```
*(out++) = U32TO32_BIG(a ^ (f + c)); \
2 *(out++) = U32TO32_BIG(e ^ (d + a));
```

and in PROCESS_RND

```
*(out++) = *(in++) ^ a ^ (f + c); \
2 *(out++) = *(in++) ^ e ^ (d + a);
```

with

```

*(out++) = U32TO32_BIG(*(in++) ^ a ^ (f + c)); \
2 *(out++) = U32TO32_BIG(*(in++) ^ e ^ (d + a));

```

After the modification of the code in the above described way the optimized reference code produces the correct keystream.

6.2.3 HC-128

For the correct execution of HC-128 on the ATmega128l only 2 small changes must be made. As mentioned in the Dragon section the standard integer types must be adapted to the microprocessor. The second change is the main function shown below:

```

u8 input[64] = {64*0};
2 u8 output[64] = {64*0};
u8 keystream[64] = {64*0};
4 u8 key[16] = {16*0};
u8 iv[16] = {16*0};
6 u32 keysize = 128;
u32 ivsize = 128;
8
int main(void){
10 ECRYPT_ctx ctx;
ECRYPT_keysetup(&ctx, key, keysize, ivsize); // key setup
12 ECRYPT_ivsetup(&ctx, iv); // IV setup
ECRYPT_process_bytes(0, &ctx, input, output, 64); // encryption
14 ECRYPT_ivsetup(&ctx, iv); // IV setup
ECRYPT_process_bytes(1, &ctx, output, input, 64); // decryption
16 return 0;
}

```

HC-128 possesses no S-boxes or other big static arrays. Therefore there is no need to include the file *pgmspace.h*.

6.2.4 LEX

In contrast to HC-128, LEX makes extensive use of static arrays. Hence we have to include the *pgmspace.h* to store the five 32-bit arrays with 256 entries to the flash memory. To accomplish that, the following lines are necessary:

```

#include <avr/pgmspace.h>
2 ...
static const u32 Te0[256] PROGMEM = { ... };
4 static const u32 Te1[256] PROGMEM = { ... };
static const u32 Te2[256] PROGMEM = { ... };
6 static const u32 Te3[256] PROGMEM = { ... };
static const u32 Te4[256] PROGMEM = { ... };

```

Of course, the standard integer types must be adapted to the microprocessor and finally there is a main function that encrypts and decrypts one 40 bit block:

```

u8 input[40]      = {40*0};
2 u8 output[40]   = {40*0};
u8 keystream[40] = {40*0};
4 u8 key[16]      = {16*0};
u8 iv[16]        = {16*0};
6 u32 keysize     = 16;
u32 ivsize       = 16;
8
int main(void){
10  ECRYPT_ctx ctx;
    ECRYPT_keysetup(&ctx, key, keysize, ivsize);           // key setup
12  ECRYPT_ivsetup(&ctx, iv);                               // IV setup
    ECRYPT_process_bytes(0, &ctx, input, output, 40);     // encryption
14  ECRYPT_ivsetup(&ctx, iv);                               // IV setup
    ECRYPT_process_bytes(1, &ctx, output, input, 40);     // decryption
16  return 0;
}

```

6.2.5 Salsa20

As with all the other ciphers, the integer types used in Salsa20 must be adapted to the microprocessor. We implemented two versions of Salsa20. One direct conversion of the original C code and one using improved rotation functions. We explain these four functions on the basis of the first function for left rotation by 7 bits. The standard rotation function requires 32 shifts to rotate a 32-bit value, no matter which value n possesses.

```

#define ROTATE(v, c) (ROTL32(v, c))
2 #define ROTL32(v, n) (U32V((v) << (n)) | ((v) >> (32 - (n))))

```

Our improved left rotation by 7 bits is done by a rearrangement of the single bytes, followed by a right rotation by 1 bit. Let the single bytes of a 32-bit value be denoted as A , B , C and D , where A is the most significant byte and D the least significant byte. Then $(A|B|C|D)$ is permuted to $(B|C|D|A)$ and afterwards the whole 32-bit value is right-rotated by 1 bit. In C this looks as follows:

```

u32 rot7(u32 value32) {
2  u8 value8 [4], tmp;
    U32TO8_LITTLE(value8, value32);
4  tmp = value8[3];
    value8[3] = value8[2];
6  value8[2] = value8[1];
    value8[1] = value8[0];
8  value8[0] = tmp;
    tmp = 0x01 & value8[0];
10  value32 = U8TO32_LITTLE(value8);
    value32 = value32 >> 1;
12  if(tmp == 0x01) {
        value32 ^= 0x80000000;
14  }
    return value32;
16 }

```

The remaining three functions work in a similar way (more information on this can be found in Chapter 7.2.4). The main function of Salsa20 is the same for both versions and looks as follows:

```

1  u8 input[64]      = {64*0};
2  u8 output[64]    = {64*0};
3  u8 keystream[64] = {64*0};
4  u8 key[16]       = {16*0};
5  u8 iv[8]         = {8*0};
6  u32 keysize      = 128;
7  u32 ivsize       = 64;
8
9  int main(void){
10     ECRYPT_ctx ctx;
11     ECRYPT_keysetup(&ctx, key, keysize, ivsize); // key setup
12     ECRYPT_ivsetup(&ctx, iv); // IV setup
13     ECRYPT_encrypt_bytes(&ctx, input, output, 64); // encryption
14     ECRYPT_ivsetup(&ctx, iv); // IV setup
15     ECRYPT_decrypt_bytes(&ctx, output, input, 64); // decryption
16     return 0;
17 }

```

6.2.6 Sosemanuk

Sosemanuk uses two 32-bit arrays with 256 elements. These two tables are stored to the flash memory by the following commands:

```

1  static u32 mul_a[] PROGMEM = { ... };
2  static u32 mul_ia[] PROGMEM = { ... };

```

The main function of Sosemanuk looks like this:

```

1  u8 input[80] = {80*0};
2  u8 output[80] = {80*0};
3  u8 key[] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0
  xBB, 0xCC, 0xDD, 0xEE, 0xFF};
4  u8 iv[] = {0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x00, 0x11, 0x22, 0
  x33, 0x44, 0x55, 0x66, 0x77};
5
6  int main(void)
7  {
8     ECRYPT_ctx ctx;
9     ECRYPT_init();
10    ECRYPT_keysetup(&ctx, key, (sizeof key)*8, (sizeof iv)*8); // key setup
11    ECRYPT_ivsetup(&ctx, iv); // IV setup
12    ECRYPT_process_bytes(0, &ctx, input, output, 80); // encryption
13    ECRYPT_ivsetup(&ctx, iv); // IV setup
14    ECRYPT_process_bytes(1, &ctx, output, input, 80); // decryption
15    return 0;
16 }

```

6.3 Results

This section provides the results for efficiency of the implementations in C language.

6.3.1 Memory Usage

A microcontroller is restricted by the the size of available flash memory and SRAM. Flash memory is used to store static information like program code or huge look-up tables. The (usually) even smaller SRAM is used for dynamic access during program execution. The memory requirements of each cipher's implementation determine the smallest possible AVR device.

Table 6.1 shows the memory allocation in flash memory. More specifically, Table 6.1 provides (i) the size of the flash memory which is used to store the program code, (ii) the required size of flash memory for the storage of static arrays, like S-boxes for instance, (iii) the total size of program code and static arrays, and (iv) the associated AVR device, i.e. the smallest device on which the implementation of the cipher can be executed without errors.

Table 6.1: Memory allocation in flash memory of C implementations

Cipher	Program Code	Static Arrays	Memory (Total)		Device
	[byte]	[byte]	[byte]	[percentage]	
AES	4616	2048	6664	40,67%	ATmega16
Dragon	55386	2048	57434	43,82%	ATmega128
HC-128	23100	0	23100	17,62%	ATmega1281
LEX	16278	5120	21398	65,30%	ATmega32
Salsa20	4478	0	4478	54,66%	ATmega8
Salsa20 V2	3842	0	3842	46,90%	ATmega8
Sosemanuk (M)	42656	2048	44704	68,21%	ATmega64
Sosemanuk (F)	22600	2048	24648	75,22%	ATmega32

In terms of flash memory consumption, the less memory is needed the better. A good indicator is the corresponding device on which the cipher is executable. As we see in Table 6.1, all devices of the ATmega family are represented. In nearly all cases the consumption of flash memory determines the associated device. Here, the only exception is HC-128. HC-128 can only be executed on an ATmega1281 device because of its huge usage of SRAM as we see in Table 6.2. We can observe that Salsa20 and AES take the lead in this area, followed by LEX and Sosemanuk. Dragon and HC-128 are only executable on the two biggest devices of the ATmega family.

Table 6.2 has nearly the same structure as Table 6.1, but focuses on the amount of SRAM needed by the ciphers. Column 2 shows the requirement of the cipher specific

structure `ECRYPT_ctx` in bytes, which represents the internal state of the cipher. This value is important because WinAVR disregards dynamic variables in the value given in Column 3 as discussed in Chapter 5.2. Column 4 provides the total size of used SRAM (based on the sum of columns 2 and 3) and the following Column 5 displays this value in percentage in relation to the associated AVR device (Column 6).

Table 6.2: Memory allocation in SRAM of C implementations

Cipher	ECRYPT_ctx [byte]	Static Variables [byte]	Total SRAM		Device
			[byte]	[percentage]	
AES	241	88	329	32,13%	ATmega16
Dragon	405	424	829	20,24%	ATmega128
HC-128	4324	232	4556	55,62%	ATmega1281
LEX	232	200	432	21,09%	ATmega32
Salsa20	64	258	322	31,45%	ATmega8
Salsa20 V2	64	258	322	31,45%	ATmega8
Sosemanuk (M)	448	192	640	15,63%	ATmega64
Sosemanuk (F)	448	192	640	31,25%	ATmega32

While all ciphers possess moderate values in the total SRAM usage, HC-128 sticks out from the others with a consumption of 4556 bytes. The ranking in this area is the same as in the area before.

NOTE: In the tables there exists a cipher named ‘Salsa20 V2’. This is our second Salsa20 implementation using the improved rotation functions.

As shown in Table 6.3 with our improved version of Salsa20, we can significantly save cycles when we replace the rotation macro by a rotation function that uses permutation of bytes prior to rotations and additionally adapts better to an 8-bit microcontroller. The original macro does 32 shifts, no matter how many bits should be rotated. Our improvement saves nearly 75% of cycles needed by the original macro.

AES, Dragon, LEX, Salsa20, Salsa20 V2, Sosemanuk (M) and Sosemanuk (F)¹ cannot be reduced to smaller AVR devices because of their consumption of flash memory². HC-128 instead has to use the ATmega1281 device because of its immense usage of SRAM shown in Table 6.2.

¹If it is necessary to get the Sosemanuk cipher running on a smaller device than an ATmega64, this goal can be achieved by replacing all macros with functions. In this case the needed size of flash memory shrinks to 24,648 bytes. The big drawback of this modification is the reduced encryption speed as visible in the Tables 6.3 and 6.4.

²Though the flash size entries of Table 6.1 seem to indicate that AES, Dragon and Salsa20 V2 can be implemented on a smaller AVR device, actually this is not possible.

6.3.2 Performance

In the following performance benchmarks we use a key and IV size of 128 bit. Input and output arrays are equal to the block size of each cipher. This means that we encrypt one block with each cipher.

Table 6.3 shows the number of cycles for the initialization, key setup, IV setup and encryption for each cipher. As seen in Table 6.3, HC-128 consumes very much cycles in the *iv_setup()* function. However, HC-128 achieves the lowest number of cycles for encryption of one block of data.

Table 6.3: Performance of initialization, key setup, IV setup, and encryption of C implementations (all numbers given are measured CPU cycles)

Cipher	Initialization	Key Setup	IV Setup	Encryption
AES	586	6953	196	12574
Dragon	2700	2136	24052	24227
HC-128	1452	460	2082876	10804
LEX	1426	2619	7367	8061
Salsa20	1700	249	71	90802
Salsa20 V2	1700	248	70	48942
Sosemanuk (M)	1282	32851	33972	14134
Sosemanuk (F)	1282	56327	61149	19938

Remarkably in Table 6.3 is the amount of required cycles for the IV setup of HC-128 in contrast to Salsa20, which requires nearly no cycles for this function. The cycle count for encryption in Column 5 lacks significance because the blocksize is not included. This circumstance has been kept in mind while creating Table 6.4. Note that the amount of cycles needed for initialization (given in Column 2) is composed of the required activities to initialize the microprocessor and the *ECRYPT_ctx* struct.

Table 6.4 focuses on the throughput of the encryption function for each cipher while Table 6.3 gives the number of cycles for one block size. More specifically, Table 6.4 provides (i) the corresponding block size, (ii) the count of cycles from Table 6.3, (iii) the quotient of the count of cycles and block size, and (iv) the throughput of the encryption function. The throughput is computed by dividing the CPU clock (assuming 8 MHz) by the quotient of the count of cycles and the block size.

As shown in Table 6.4, the ciphers can be classified into two groups regarding throughput. HC-128, Sosemanuk (M), Dragon, LEX and Sosemanuk (F) belong to the fast group. Salsa20 V2, AES and Salsa20 reside in the slow group. Important is the fact that the improved Salsa20 implementation produces nearly twice the output of the original Salsa20 implementation. Note further that for long keystreams the encryption is the dominant factor (but remember the huge time for IV setup of HC-128). When only a

Table 6.4: Throughput of encryption of C implementations

Cipher	Block Size [byte]	Encryption [cycles]	Ratio [cycles/byte]	Throughput [bytes/sec] @8MHz
AES	16	12574	785,88	10180
Dragon	128	24227	189,27	42267
HC-128	64	10804	168,81	47390
LEX	40	8061	201,53	39697
Salsa20	64	90802	1418,78	5639
Salsa20 V2	64	48942	764,72	10461
Sosemanuk (M)	80	14134	176,68	45281
Sosemanuk (F)	80	19938	249,23	32100

small amount of keystream has to be generated, it can be seen from Table 6.4 that LEX is the most efficient cipher. Up to an output of approximately 48 bytes of keystream, AES is the second most efficient that is then surpassed by Dragon.

7 Results in Assembly Language

7.1 Goals and Objectives

This chapter provides the results on efficiency of our implementations in Assembly. Details on the framework used are provided in Chapter 5.3. We used an Assembly implementation of the AES cipher [18] to be able to adequately compare our Assembly implementations of Dragon, LEX, Salsa20 and Sosemanuk. We did not implement HC-128 in Assembly because its huge consumption of SRAM memory prohibits the implementation on any small AVR device. For each cipher, two different implementations have been created. Dragon, Salsa20 and Sosemanuk are implemented in a function-based version and a macro-based version, respectively. The function-based versions are realized with the goal of minimizing the use of flash memory. In contrast, the macro-based versions are optimized to reach high throughput rates. LEX is treated as a special case. The version called ‘LEX’ is the transformation of the C version of LEX in Assembly language using five big static arrays. By contrast, the version named ‘LEX V2’ is an Assembly language implementation derived from our Assembly language implementation of the AES.

NOTE: It is important to mention that the savings in cycles mentioned below are only possible if the C code is translated into Assembly language in a native and unoptimized way. However, the C compiler partly optimizes the code during translation. Because of this we can not save as much cycles as the numbers below may indicate in practice.

7.2 Implementation

This section contains implementation details of AES, Dragon, LEX, Salsa 20 and Sosemanuk (alphabetically ordered).

7.2.1 AES

The AES cipher is implemented in Assembly language for comparison reasons. As a matter of fairness we did not want to compare the C language version of AES with the adjusted Assembly versions of the other ciphers. So we decided to use the AES implementation of Christian Roepke [18], which adapts well to an 8-bit microcontroller. This implementation makes use of on-the-fly subkey computation, whereby the computation of the subkeys is included in the encryption function. We modified the given implementation in such a way, that the generation of the subkeys is entirely done before encryption. The IV setup is a simple XOR of the 16 plaintext bytes and the 16 IV bytes. Figure 7.1 shows the memory allocation of AES in SRAM.

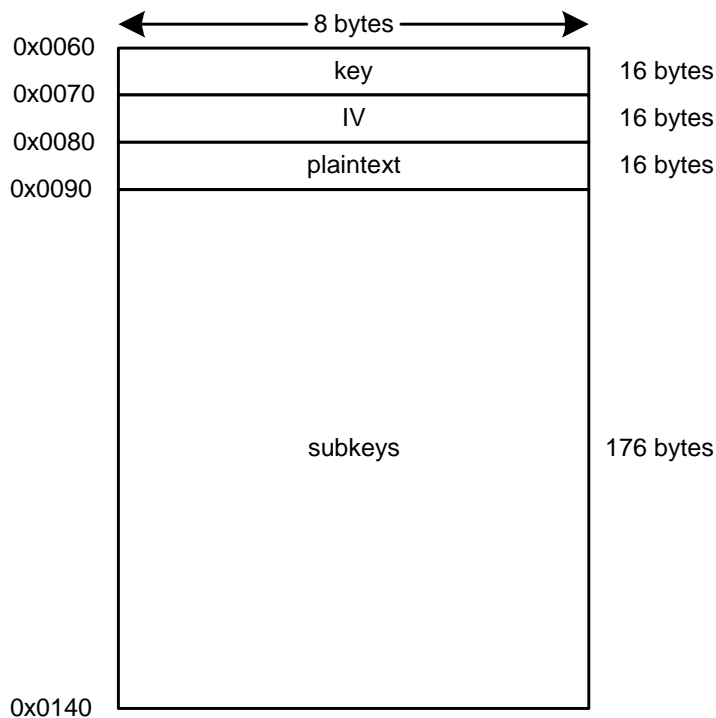


Figure 7.1: AES: Memory allocation in SRAM

Key Setup

The key setup computes the subkeys by the application of the minimal S-box with 2^8 elements, the *rcon* table and the *RotWord()* function as given in the specification of the AES [8].

IV Setup

As already noted, the IV setup consists of a simple XOR of the 16 plaintext bytes and the 16 IV bytes to transform the AES from ECB to CBC mode.

Encryption

As given in the specification of AES, the encryption consists of 10 rounds. Preceding to the first round, the first subkey is added to the plaintext. Subsequently 9 full rounds are applied. This means that the following 4 functions are called in this order:

- S_BOX()
- SHIFT_ROWS()
- MIX_COLUMNS()
- SKEYADD()

The last round is a full round without the application of the *MIX_COLUMNS()* function. The *SHIFT_ROWS()* function rotates values by 1, 2 and 3 bytes. We make no use of the rotation functions but move the bytes directly to the destination register. This makes it possible to save a lot of cycles in contrast to the C language implementation.

7.2.2 Dragon

The C source code of Dragon offers a lot of possibilities to enhance the speed of the cipher and to minimize the code size. The speed of the C version is good, but an ATmega128 as running device is not the first choice when a stream cipher should be implemented on a 8-bit microcontroller. The S-boxes G and H are frequently used in the Dragon cipher, during IV setup, as well as during the encryption and the decryption functions. So the focus lies on the fast implementation of these virtual 32×32 S-boxes. Further improvements can be made through intelligent register and SRAM handling, so that values which are changed frequently are held in the registers instead of being written back into SRAM. Figure 7.2 shows the memory allocation of Dragon in SRAM.

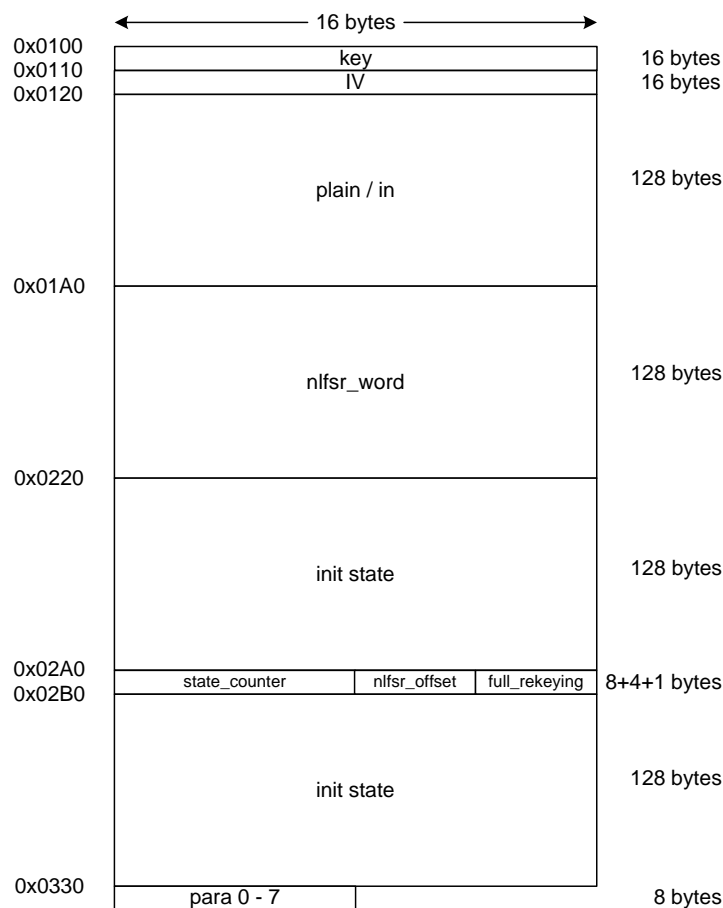


Figure 7.2: Dragon: Memory allocation in SRAM

Key Setup

During key and IV setup the LFSR is primarily initialized using the key and the IV in the following manner: $(k|(k' \wedge iv')|iv|(k \wedge iv')|k'|(k \wedge iv)|iv'|(k' \wedge iv))$, where $|$ denotes a concatenation. If the 128 bit key is divided into 4 parts, the key can be formalized as $k = (k_0|k_1|k_2|k_3)$ and $k' = (k_2|k_3|k_0|k_1)$. The identifiers k' and iv' denote the key and the IV, whereby the first and the last half of the key, respectively the IV, are swapped. In the key setup only k and k' are stored to the relevant positions (for further details see Figure 7.3 and Figure 7.4). In the Assembly language version of the key setup the `U8TO32_BIG()` macro is not needed, because the separate bytes can be addressed directly. This saves a lot of cycles. The `U8TO32_BIG()` macro looks like this:

```

2 #define U32TO32_BIG(v) SWAP32(v)
   #define SWAP32(v) \
4   ((ROTL32(v, 8) & U32C(0x00FF00FF)) | \
   (ROTL32(v, 24) & U32C(0xFF00FF00)))
6 #define ROTL32(v, n) \
   (U32V((v) << (n)) | ((v) >> (32 - (n))))

```

That means that the `U8TO32_BIG()` macro calls the `SWAP32()` macro which calls the `ROTL32()` macro. It does a swapping of the four bytes of a 32-bit value. On an 8-bit microcontroller we have the ability to access the separate bytes directly and so we simply read the bytes in reverse order. This completely avoids the use of the `U8TO32_BIG()` macro which saves 276 CPU cycles ($(2 \cdot 32 \cdot 4)$ cycles for shift operations, $(2 \cdot 4) + 4$ cycles for binary ORs and $(2 \cdot 4)$ binary ANDs). The `U8TO32_BIG()` macro is used eight times during the key setup and so we save a total of 2,208 cycles.

Regardless of the explicit implementation, the key setup can be displayed as follows (`ctx→nlfsr_word[]` is an 32-bit array of size 32):

k_0 is written at the positions 0, 6, 12, 18, 20 and 30 of `ctx→nlfsr_word[]`.
 k_1 is written at the positions 1, 7, 13, 19, 21 and 31 of `ctx→nlfsr_word[]`.
 k_2 is written at the positions 2, 4, 14, 16, 22 and 28 of `ctx→nlfsr_word[]`.
 k_3 is written at the positions 3, 5, 15, 17, 23 and 29 of `ctx→nlfsr_word[]`.

Afterwards the current state of `ctx→nlfsr_word[]` is copied to `ctx→init_state[]`. The saving of cycles arise from the unrolling of the for-loops.

NOTE: The positions 8 to 11 and 24 to 27 of the `ctx→nlfsr_word[]` array stay unassigned in the key setup. This positions will be filled in the IV setup.

IV Setup

The first part of the IV setup of Dragon is either a continuation of the key initialization, or a fresh rekeying. In the second case, first `ctx→init_state[]` is copied to

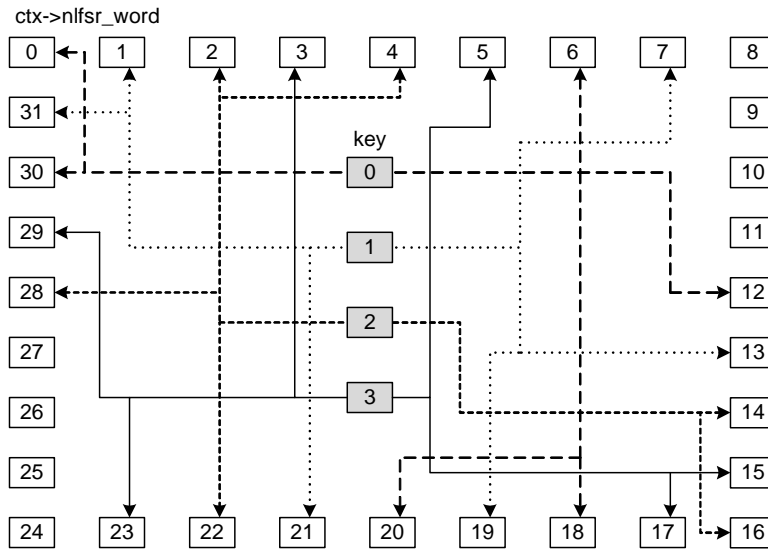


Figure 7.3: Dragon: Key setup

`ctx->nlfsrc_word[]`. The second phase of the IV setup consists of the mixing process, where the values of the elements of the `ctx->nlfsrc_word[]` array are randomized.

The first part of the IV setup is composed of the writing and XORing of `iv` and `iv'`. Similar to the key setup, the IV setup can be displayed in a very easy manner. Let us denote $iv = (iv_0|iv_1|iv_2|iv_3)$ and $iv' = (iv_2|iv_3|iv_0|iv_1)$. Then the IV setup can be illustrated as followed:

iv_0 is written at the positions 8 and 26 of `ctx->nlfsrc_word[]`,
 iv_1 is written at the positions 9 and 27 of `ctx->nlfsrc_word[]`,
 iv_2 is written at the positions 10 and 24 of `ctx->nlfsrc_word[]`,
 iv_3 is written at the positions 11 and 25 of `ctx->nlfsrc_word[]`,

iv_0 is XORed with and written at the positions 6, 14, 20 and 28 of `ctx->nlfsrc_word[]`,
 iv_1 is XORed with and written at the positions 7, 15, 21 and 29 of `ctx->nlfsrc_word[]`,
 iv_2 is XORed with and written at the positions 4, 12, 22 and 30 of `ctx->nlfsrc_word[]`,
 iv_3 is XORed with and written at the positions 5, 13, 23 and 31 of `ctx->nlfsrc_word[]`.

We save a lot of cycles because we are able to access the separate bytes of the 32-bit values directly and because we do not use loops like in the C version.

At the second part of the IV setup, the elements of the array `ctx->nlfsrc_word[]` are scrambled to complete the initialization process. Therefore the macros with the names `DRAGON_NLFSR_WORD()`, `DRAGON_OFFSET()` and `DRAGON_UPDATE()` are used. The `DRAGON_OFFSET()` macro specifies and alters the current offset. This

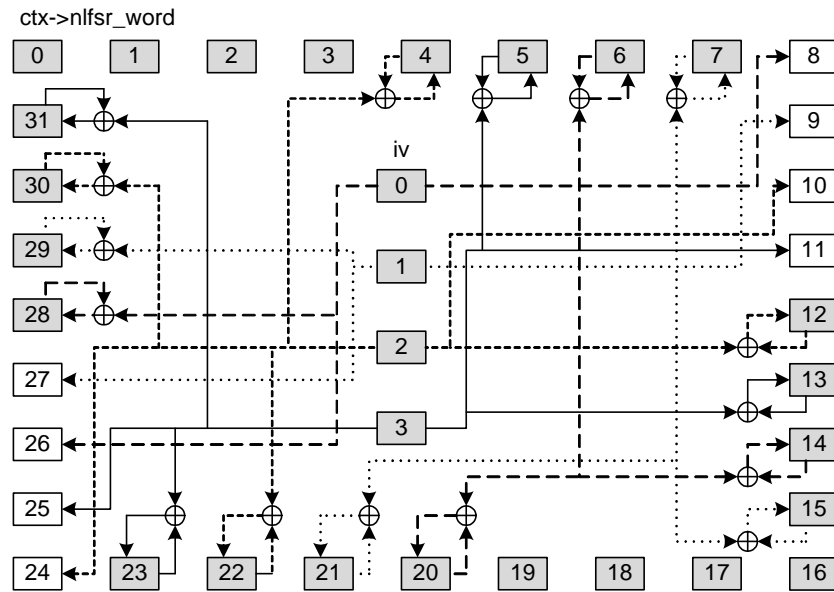


Figure 7.4: Dragon: First part of the IV setup

offset is added to the current element of `ctx->nlfsw_word`] (given as parameter) and afterwards the resulting value modulo 32 is computed. The `DRAGON_NLFSR_WORD()` macro changes the value of the element given in the parameter (the offset is included in this computation). The `DRAGON_UPDATE()` macro uses XORs, additions modular 2^{32} and the virtual S-boxes G_1 to G_3 and H_1 to H_3 . The S-box G_1 is defined as follows (in C notation):

```
#define G1(x) \
2  RFF(sbox2[x & 0xFF]) ^ \
   RFF(sbox1[(x >> 8) & 0xFF]) ^ \
4  RFF(sbox1[(x >> 16) & 0xFF]) ^ \
   RFF(sbox1[(x >> 24) & 0xFF])
```

NOTE: The `RFF()` macro is explained in Chapter 5.1.

To access the separate bytes of the 32-bit value x , 48 right-shifts, 4 ANDs and 3 XORs are needed, which leads to a total saving of $(8 \cdot 4) + (16 \cdot 4) + (24 \cdot 4) + (4 \cdot 4) + (3 \cdot 4) = 220$ cycles in Assembly language. This corresponds to 1,320 cycles at each call of the `DRAGON_UPDATE()` macro (6 S-box calls are used in the macro) and altogether 21,120 cycles within the 16 rounds during the IV setup. As in the key setup the `U8TO32_BIG()` macro is used 8 times, so we save another 2,208 cycles. The remaining instructions do not offer great opportunities for enhancing the speed or scaling down the code size.

Encryption

The encryption function is an iterated call of the *KEYSTREAM_ROUND()* macro which includes the *BASIC_ROUND()* macro. This call of the *KEYSTREAM_ROUND()* macro is done 16 times.

```

#define BASIC_RND(ctx, a, loc_a, b, loc_b, c, loc_c, \
2      d, loc_d, e, loc_e, f, loc_fb1, c1, c2)\
4      a = ctx->nlsr_word[loc_a]; \
      c = ctx->nlsr_word[loc_c]; \
      e = ctx->nlsr_word[loc_e] ^ c1; \
6      b = ctx->nlsr_word[loc_b] ^ a; \
      d = ctx->nlsr_word[loc_d] ^ c; \
8      f = (ctx->nlsr_word[loc_e+1] ^ e) ^ (c2++); \
      c += b; \
10     e += d; \
      a += f; \
12     f ^= G2(c); b ^= G3(e); d ^= G1(a); \
      e ^= H3(f); a ^= H1(b); c ^= H2(d); \
14     ctx->nlsr_word[loc_fb1] = b + e; \
      ctx->nlsr_word[loc_fb1+1] = c ^ (b + e);

```

The essential part of this macro is the application of the S-boxes. These are used in exactly the same manner as in the *DRAGON_UPDATE()* macro. Hence, we can save the same amount of cycles here. We save 220 cycles with every call of a G or H function. These functions are called 6 times in each round and the encryption function runs over 16 rounds. So we save $(220 \cdot 6 \cdot 16) = 21,120$ cycles through this. Furthermore, the *BASIC_ROUND()* macro is programmed in such way that all data can be held in the registers. There is no need for outsourcing data into the SRAM if it is not specified by the algorithm.

```

#define KEYSTREAM_RND(ctx, a, loc_a, b, loc_b, c, loc_c, \
2      d, loc_d, e, loc_e, f, loc_fb1, c1, c2, in, out)\
      BASIC_RND(ctx, a, loc_a, b, loc_b, c, loc_c, \
4      d, loc_d, e, loc_e, f, loc_fb1, c1, c2) \
      tmp = a ^ (f + c); \
6      *(out++) = U32TO32_BIG(tmp); \
      tmp = e ^ (d + a); \
8      *(out++) = U32TO32_BIG(tmp);

```

In the rest of the *KEYSTREAM_ROUND()* macro we are able to save another $(276 \cdot 2) = 552$ cycles per round (because of the *U32TO32_BIG()* macro), which accumulates to 8,832 saved cycles.

7.2.3 LEX

As we already mentioned in the introduction of this chapter we implemented two versions of LEX in Assembly language. In this section we describe the implementation of the ‘LEX’ version. As the version ‘LEX V2’ is in principle an improved and slightly modified AES, we do not take a closer look at this version. Figure 7.5 shows the memory allocation of LEX in SRAM.

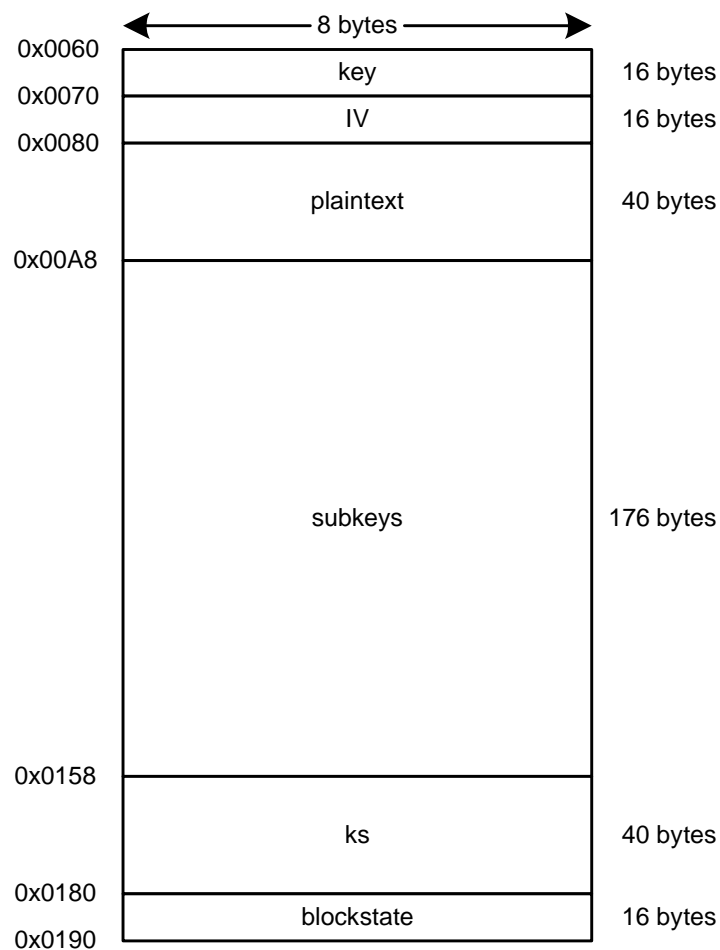


Figure 7.5: LEX: Memory allocation in SRAM

Key Setup

LEX makes extensive use of 5 static arrays with 256 4-byte values. The fourth of these 5 tables, named $Te4//$, is used for the key setup. So the key setup only consists of load

operations from this array, the *rcon* table lookups and some XOR operations. These operations do not offer great possibilities for improvements or reduction of code size.

NOTE: The *Te4[]* array is solely an extended version of the original S-Box with 2^8 elements of the AES. All 4 bytes of one element of this array have the same value. This simplifies the handling with the values of the S-box on a computer with 4-byte word size, but is not necessary on an 8-bit microcontroller. To save flash memory and to enhance the speed of the key setup of LEX this array can be reduced to 256 entries of 8-bit size. A similar reduction is possible for the *rcon* table.

IV Setup

The IV setup of LEX consists of the encryption of the IV under the secret key *k*. Therefore the IV serves as input/plaintext for the encryption function, which is also used for the normal encryption. The implementation of the encryption function of LEX is explained in the next section.

Encryption

The encryption function uses the 4 precomputed arrays *Te0[]* to *Te3[]* to make table lookups instead of using the functions *S_BOX()*, *SHIFT_ROWS()*, *MIX_COLUMNS()* and *SKEYADD()*. First, the plaintext is XORed with the first subkey and afterwards the lookup tables are used. This looks like this:

```

/* round 1: */
2 t0 = RFF(Te0[s0 >> 24]) ^ RFF(Te1[(s1 >> 16) & 0xff]) ^ RFF(Te2[(s2 >> 8) & 0xff]) ^
  RFF(Te3[s3 & 0xff]) ^ rk[ 4];
t1 = RFF(Te0[s1 >> 24]) ^ RFF(Te1[(s2 >> 16) & 0xff]) ^ RFF(Te2[(s3 >> 8) & 0xff]) ^
  RFF(Te3[s0 & 0xff]) ^ rk[ 5];
4 t2 = RFF(Te0[s2 >> 24]) ^ RFF(Te1[(s3 >> 16) & 0xff]) ^ RFF(Te2[(s0 >> 8) & 0xff]) ^
  RFF(Te3[s1 & 0xff]) ^ rk[ 6];
t3 = RFF(Te0[s3 >> 24]) ^ RFF(Te1[(s0 >> 16) & 0xff]) ^ RFF(Te2[(s1 >> 8) & 0xff]) ^
  RFF(Te3[s2 & 0xff]) ^ rk[ 7];
6 ctx->ks[0] = (t0 & 0xFF00FF00) ^ ((t2 & 0xFF00FF00)>>8); /* Leak for odd rounds */
/* round 2: */
8 s0 = RFF(Te0[t0 >> 24]) ^ RFF(Te1[(t1 >> 16) & 0xff]) ^ RFF(Te2[(t2 >> 8) & 0xff]) ^
  RFF(Te3[t3 & 0xff]) ^ rk[ 8];
s1 = RFF(Te0[t1 >> 24]) ^ RFF(Te1[(t2 >> 16) & 0xff]) ^ RFF(Te2[(t3 >> 8) & 0xff]) ^
  RFF(Te3[t0 & 0xff]) ^ rk[ 9];
10 s2 = RFF(Te0[t2 >> 24]) ^ RFF(Te1[(t3 >> 16) & 0xff]) ^ RFF(Te2[(t0 >> 8) & 0xff]) ^
  RFF(Te3[t1 & 0xff]) ^ rk[10];
s3 = RFF(Te0[t3 >> 24]) ^ RFF(Te1[(t0 >> 16) & 0xff]) ^ RFF(Te2[(t1 >> 8) & 0xff]) ^
  RFF(Te3[t2 & 0xff]) ^ rk[11];
12 ctx->ks[1] = ((s1 & 0xFF00FF) << 8) ^ (s3 & 0xFF00FF); /* Leak for even rounds */

```

Obviously the C implementation uses eight 32-bit values. These values are called s_0 to s_3 (group *s*), respectively t_0 to t_3 (group *t*) and one of the groups is alternately used as a storage for the values of the previous or the current round. This means that four 32-bit values must be saved in order to compute the values of the present round. These are 16

bytes of the last round and 16 bytes of the current round. As the ATmega8 possesses only 32 registers, whereby we need some of them for pointer handling and temporary values, we can not hold all these 32 bytes in the registers at the same time. The C version swaps out the values of the last round to the SRAM, but we were able to devise a more efficient solution. We analyzed the algorithm and found out that if we want to store the values of the s and the t group in the same registers, then it is only necessary to swap out six 8-bit values. If we divide a 32-bit value X to single bytes and we use the notation $X = (A|B|C|D)$ then the 6 bytes to store can be denoted as: $S_0(B)$, $S_0(C)$, $S_0(D)$, $S_1(C)$, $S_1(D)$ and $S_2(D)$. The same applies to the values of the t group. The remaining necessary bytes are still accessible in the registers. This improvement saves a lot of cycles, because we do not have to store the values to the SRAM and then reload them later.

7.2.4 Salsa20

The most important part of Salsa20 is the quarterround function. At this point, chances are good to save CPU cycles. The rest of the cipher can partly be optimized but these changes do not affect the cycle count as much as the optimization of the quarterround function. To alleviate the understanding of the setup of the cipher and the memory allocation in SRAM, take a look at Figure 7.6.

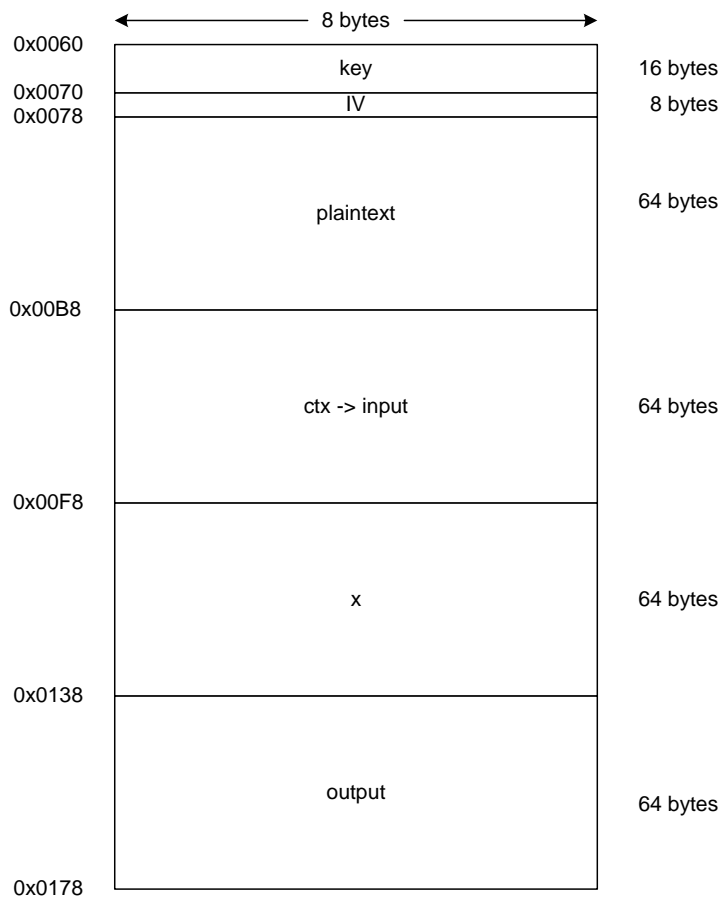


Figure 7.6: Salsa20: Memory allocation in SRAM

Initialization

During the initialization phase, the key, the IV, and the plaintext are stored into the SRAM. This part is not included in the cycle count calculation of the key setup. The 16 bytes of the key are stored to the start address of the SRAM at position 0x0060, followed by the 8 bytes of the initialization vector starting from 0x0070. Another 64 bytes of the

plaintext are written into the SRAM, beginning at start address 0x0078. Furthermore, the stack is initialized and the Y pointer is filled with the start address of the SRAM.

Key Setup

In the key setup the 32-bit array `x->input[]` of size 16 is filled with certain parts of the key k and the constant τ . In the C version the macro `U8TO32_LITTLE()` is used to get the 32 bits out of an 8-bit array in little endian order. In Assembly language, there exists the possibility to access the 4 bytes of a 32-bit value directly and therefore a few cycles can be saved because the reordering of the bytes can be done by reading in the values backwards. For this, the key setup in Assembly language needs fewer cycles than the C language version. The constant τ (τ) is loaded from flash by the following code:

```

; bend over Z pointer to tau
2  ldi ZL,LOW(tau*2)
   ldi ZH,HIGH(tau*2)
4
; U8TO32_LITTLE(tau + 0)
6  lpm reg3,Z+
   lpm reg2,Z+
8  lpm reg1,Z+
   lpm reg0,Z+

```

IV Setup

In the key setup only 12 of the 16 entries of the array `x->input[]` are filled. The remaining 4 bytes are filled with the IV and zeros. Here the `U8TO32_LITTLE()` is used as well and therefore the IV setup in Assembly language is a little bit faster than in the C version.

Encryption

The encryption of Salsa20 is done by calling the doubleround function 10 times. This amounts to 8 calls of the quarterround function in every of the 10 rounds. All in all, the encryption consists of 80 calls of the quarterround function. So the main focus of speeding up the entire encryption lies in optimizing the quarterround function.

The quarterround function

As shown in Chapter 3.6.1, the quarterround function makes extensive use of left rotations, more precisely, left rotation by 7, 9, 13 and 18 bits.

```

#define ROTATE(v, c) (ROTL32(v, c))
2 #define ROTL32(v, n) (U32V((v) << (n)) | ((v) >> (32 - (n))))

```

The *ROTATE()* macro does 32 shifts and an OR operation, no matter how many bits should be rotated. There is a great saving of cycles when using permutations of bytes in favor of rotations which is very easy to realize in Assembly language.

Here the explanation of the realizations of the fast rotations by 7, 9, 13 and 18 bits follows. Let the single bytes of a 32-bit value be denoted as *A*, *B*, *C* and *D*, where *A* is the most significant byte and *D* the least significant byte. The original *ROTATE* macro needs 132 cycles for execution. This means that alone the rotations in the quarterround function require 528 CPU cycles.

The left rotation by 7 bits is done by a rearrangement of the single bytes, followed by a right rotation by 1 bit. This is shown in Figure 7.7. (*A|B|C|D*) is permuted to (*B|C|D|A*) and afterwards the whole 32-bit value is right-rotated by 1 bit. This improved rotation needs only 11 cycles in average, only 8% of the cycles required by the original macro.

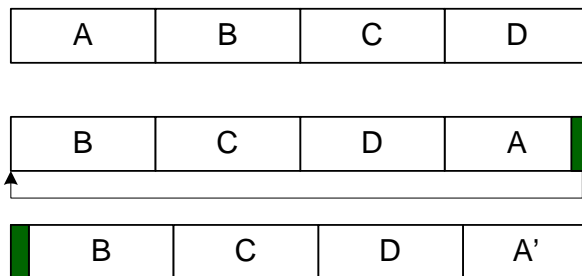


Figure 7.7: Salsa20: Left rotation of a 32-bit value by 7 bits

The left rotation by 9 bits is also done by a rearrangement of the single bytes, but now followed by a left rotation by 1 bit. This is shown in Figure 7.8. (*A|B|C|D*) is permuted to (*B|C|D|A*) and afterwards the whole 32-bit value is left-rotated by 1 bit. This improved rotation needs only 11 cycles in average. Again, this is only 8% of the cycles required by the original macro.

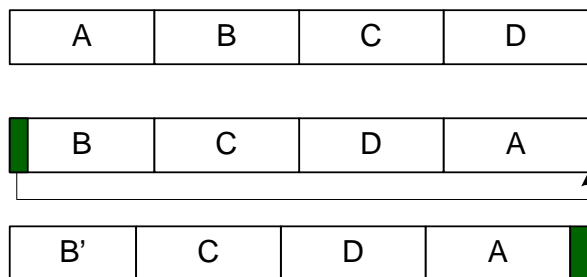


Figure 7.8: Salsa20: Left rotation of a 32-bit value by 9 bits

The left rotation by 13 bits is done by a rearrangement of the single bytes, followed by a right rotation by 3 bits. This is shown in Figure 7.9. (*A|B|C|D*) is permuted

to (C|D|A|B) and afterwards the whole 32-bit value is right-rotated by 3 bits. This improved rotation needs only 29 cycles in average, 21% of the cycles required by the original macro.

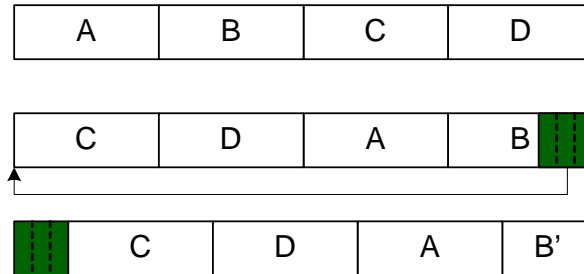


Figure 7.9: Salsa20: Left rotation of a 32-bit value by 13 bits

The left rotation by 18 bits is done by a rearrangement of the single bytes, followed by a left rotation by 2 bits. This is shown in Figure 7.10. (A|B|C|D) is permuted to (C|D|A|B) and afterwards the whole 32-bit value is left-rotated by 2 bits. This improved rotation needs only 22 cycles in average 16% of the cycles required by the original macro.

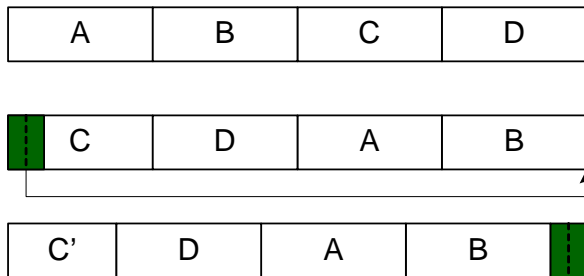


Figure 7.10: Salsa20: Left rotation of a 32-bit value by 18 bits

The application of these 4 improved rotation versions for rotations by 7, 9, 13 and 18 bits consumes only 73 cycles instead of 528 of the original rotation version. This saves 36,400 cycles in the 80 calls of the quarterround function. The rest of the quarterround function provides no significant optimization potential. The first 16 registers are filled by values taken from the SRAM. The registers 16 to 19 are used as temporary memory. Therefore there is just a little overhead, which does not exist in the C version. After the call of the quarterround function, the values are stored back to the SRAM.

7.2.5 Sosemanuk

The Sosemanuk cipher uses a reduced version of the Serpent cipher for IV initialization and the Snow 2.0 LFSR with a reduced internal state. Consequently, this increases the size of the code. The Sosemanuk implementation features 2 inline functions and 20 macros to realize a successful encryption. Four of these 20 macros use the *ROTL()* macro of which we know that we can save cycles by using improved rotation functions. Another possibility to save cycles is the fast implementation of a (32×32) -bit multiplication, which is used in the *FSM()* macro.

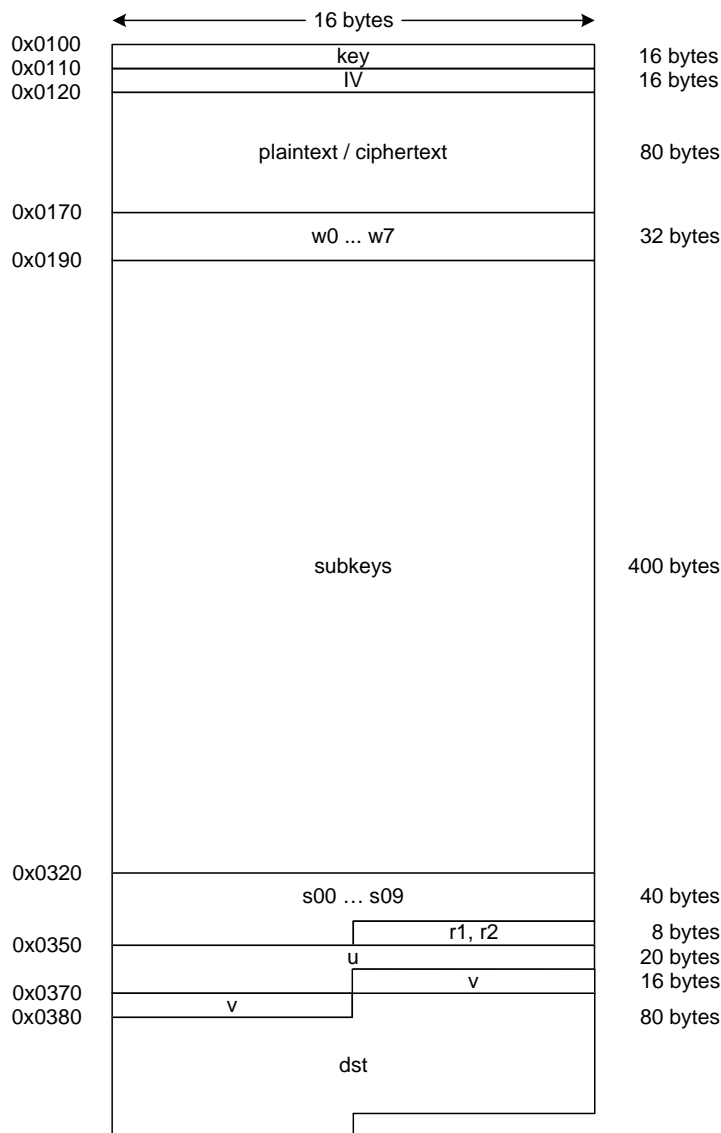


Figure 7.11: Sosemanuk: Memory allocation in SRAM

The cycle count can also be reduced by efficient resource handling. In the Assembly code of Sosemanuk all macros are implemented in such a way that they only need 4 additional registers for the saving of temporary values. Furthermore, if possible, back writing of values is tried to be avoided and replaced by movement of variables.

Key Setup

During the key setup the following macros are used:

- $S0()$ to $S7()$,
- $SKS0()$ to $SKS7()$ and
- $WUP0()$ and $WUP1()$.

The S-box macros $S0()$ to $S7()$ hold no great potential of optimization. They make use of only very trivial operations like exclusive OR, binary inversion, binary OR, and binary AND. These operations can be efficiently executed on a microcontroller. The S-box take five 32-bit values as input and is implemented by using only one additional 32-bit value for storing intermediate results.

The macros $SKS0()$ to $SKS7()$ use the corresponding S-box macro and store four of the five 32-bit values as sub-keys.

Rotations by 11 bit are used 4 times in the $WUP0()$ macro, respectively in the $WUP1()$ macro. During the key setup the $WUP0()$ macro is called 13 times, the $WUP1()$ macro is called 12 times. This is an overall sum of 100 calls of the rotation macro. Instead of 132 cycles the improved rotation version needs only 28 cycles. This is a saving of $104 \cdot 4 = 416$ cycles per call of the macro and an overall saving of 10,400 in the whole key setup. The rotation by 11 bits is implemented as shown in Figure 7.12.

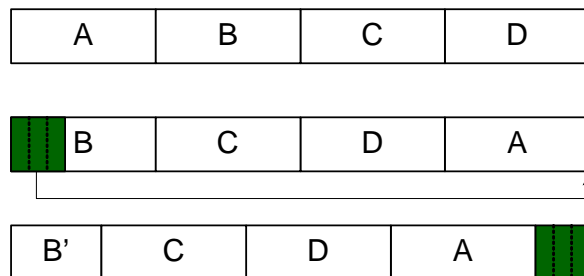


Figure 7.12: Sosemanuk: Left rotation of a 32-bit value by 11 bits

IV Setup

During IV setup the following macros are used:

- *FSS()*,
- *KA()*,
- *S0* to *S7* and
- *SERPENT_LT()*.

Several macros are called by the *FSS()* macro, namely the *KA()* macro, the *S0* to *S7* macros and the *SERPENT_LT()* macro.

The *KA()* macro performs solely a XOR association between its 4 input values and the subkey at the position *offset*.

Dependent on the input of the *FSS()* macro, one of the 8 *S()* macros is called and the input values pass through the corresponding S-box.

Rotations and also shift operations are contained in the *SERPENT_LT()* macro. Here 32-bit values will be rotated by 1, 3, 5, 7, 13 and 22 bit and shifted by 3 and 7 bit. The rotations by 1 and 3 bit are processed as usual. Rotation by 5 and 7 bits are done by reordering of the separate bytes ((A|B|C|D) becomes (B|C|D|A)) and a subsequent right rotation by 3 bits, respectively 1 bit. The rotation by 13 bits is already shown in Figure 7.9. The left rotation by 22 bits is a right rotation by 10 bits and so the 32 bit value is right rotated by 2 bits after reordering from (A|B|C|D) to (D|A|B|C).

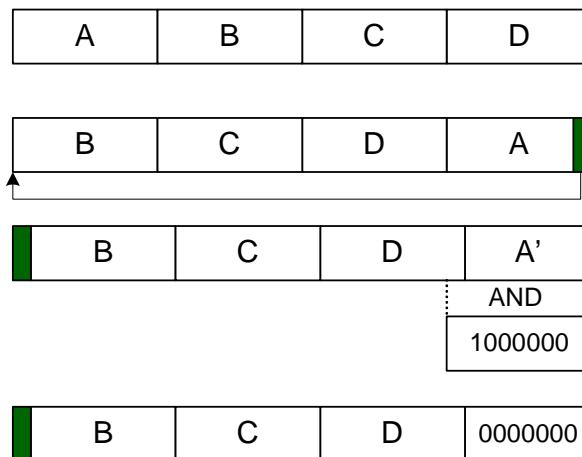


Figure 7.13: Sosemanuk: Left shift of a 32-bit value by 7 bits

The shifting operation by 3 bits is done in the natural way again and the shifting by 7 bits is shown in Figure 7.13. First the order of the separate bytes is changed

from (A|B|C|D) to (B|C|D|A). Afterwards, the whole 32-bit value is right shifted by 1 bit and finally the least significant byte is AND-associated with the hexadecimal value $0x80$. This leads to the deletion of the 7 least significant bits of the 32-bit value. This method needs only 12 cycles in average, whereby the normal left shift requires 43 cycles. So we can save 72% of the cycles used so far. All in all during the *SERPENT_LT()* macro, simply by replacing the original rotation and shift macros/operations with the improved versions, 700 cycles can be saved. Instead of 835 cycles our improved version requires only 135 cycles in average. This are only 28% and because of 24 calls of the *SERPENT_LT()* macro we can save up to 16,800 CPU cycles.

Encryption

During the encryption the following important macros are used:

- *MUL_A()*,
- *MUL_G()*,
- *STEP()*,
- *FSM()*,
- *LRU()*,
- *CC1()* and
- *SRD()*.

The three macros *FSM()*, *LRU()* and *CC1()* together build the *STEP()* macro, which is called 20 times during the encryption process. Inside the *FSM()* macro, most cycles are required by a (32×32) -bit multiplication modulo 2^{32} and a left rotation by 7 bits. As we already know, we can save 121 cycles with our improved version of the rotation and also the multiplication can be speeded up.

Figure 7.14 shows a graphical illustration of a 32×32 -bit multiplication modulo 2^{32} and our approach to enhance the computation.

```

1 #define ONE32      ((u32)0xFFFFFFFF)
2 #define T32(x)     ((x) & ONE32)
3 // some lines ...
4 tt = T32(or1 * 0x54655307); \

```

The listing above shows that the the value *or1* and the static value $0x54655307$ are multiplied within the *T32()* macro. This macro cuts all digits beyond the limit of the length of 32 bits. Let us denote the first 32-bit factor for the multiplication as X and the second factor as Y. Obviously the C version of the multiplication computes the product of every byte of X multiplied with every byte of Y. Afterwards the *T32()* macro cuts the dispensable parts of the product. But there is a more efficient way to

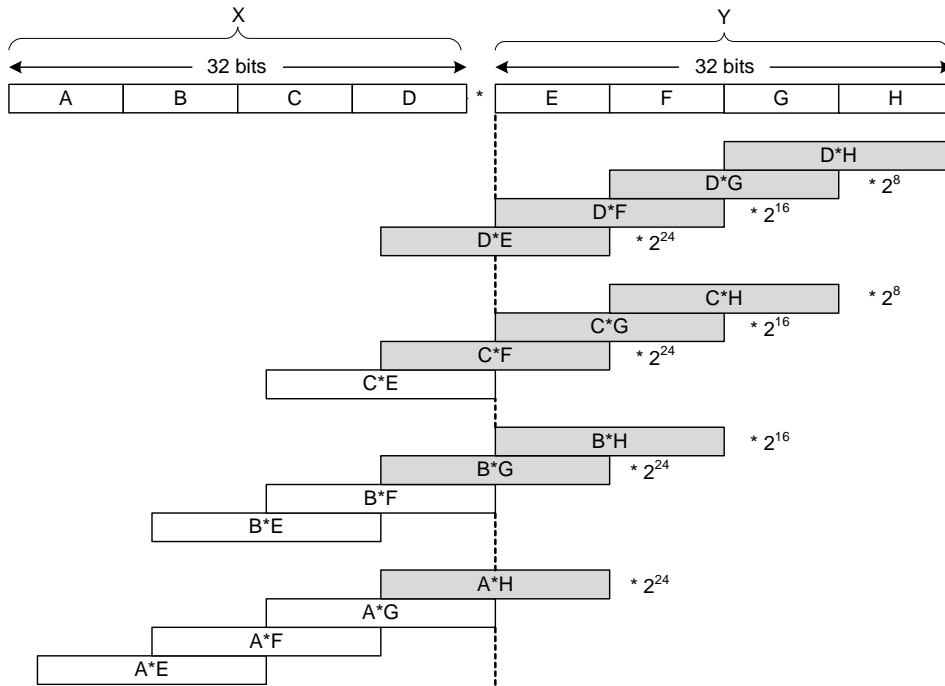


Figure 7.14: Sosemanuk: Fast implementation of a (32×32) -bit multiplication mod 2^{32}

compute the product. All intermediate products with 4 zero bytes or more (whereby the counting starts from the least significant byte) can be ignored. So the formula for the (32×32) -bit multiplication modulo 2^{32} can be transformed to a reduced form. If we denote $X = (a|b|c|d)$ and $Y = (e|f|g|h)$ then the multiplication $(X \cdot Y) \bmod 2^{32}$ can be transformed to $(a000 + b00 + c0 + d) \cdot (e000 + f00 + g0 + h) \bmod 2^{32}$, where a 0 stands for a complete byte filled with zeros. Now let us take a look at our improved formula:

$$\begin{aligned} (X \cdot Y) \bmod 2^{32} &= ((a000 \cdot e000) + (a000 \cdot f00) + (a000 \cdot g0) + (a000 \cdot h)) + \\ &((b00 \cdot e000) + (b00 \cdot f00) + (b00 \cdot g0) + (b00 \cdot h)) + \\ &((c0 \cdot e000) + (c0 \cdot f00) + (c0 \cdot g0) + (c0 \cdot h)) + \\ &((d \cdot e000) + (d \cdot f00) + (d \cdot g0) + (d \cdot h)) \bmod 2^{32} \end{aligned}$$

If we now eliminate all products with 4 or more zero-bytes we get the following formula:

$$\begin{aligned} (X \cdot Y) \bmod 2^{32} &= ((d \cdot e) + (c \cdot f) + (b \cdot g) + (a \cdot h)) \cdot 2^{24} + \\ &((d \cdot f) + (c \cdot g) + (b \cdot h)) \cdot 2^{16} + \\ &((d \cdot g) + (c \cdot h)) \cdot 2^8 + \\ &(d \cdot h) \bmod 2^{32} \end{aligned}$$

The C version multiplication requires 16 (1×1) -byte multiplications and 15 1-byte additions. If we disregard some necessary additional operations like moving of values

between registers and copy operations this multiplication approach needs 62 cycles. The improved version in Assembly language only requires 10 (1×1)-byte multiplications and 9 1-byte additions, which saves 38 cycles, a saving of 39%. If we involve the number of calls of the *FSM()* macro and include the saving of cycles because of the improved 7-bit rotation, we can save at least 2,700 CPU cycles.

```
#define MUL_A(x)      (T32((x) << 8) ^ RFF(mul_a[(x) >> 24]))
2 #define MUL_G(x)    (((x) >> 8) ^ RFF(mul_ia[(x) & 0xFF]))
```

In the *LRU()* macro, the two macros *MUL_A()* and *MUL_G()* (as shown above) are used. In the *MUL_A()* macro we can save 128 cycles because we can access the separate bytes of the parameter *x* directly. The same applies for the *MUL_G()* macro, but here only 33 cycles can be saved. This leads to an overall saving of 151 cycles per call of the *LRU()* macro and so 3,020 CPU cycles can be saved during the whole encryption process.

The *CC1()* macro only uses addition modular 2^{32} and the XOR operation, so that no great saving of cycles can be recorded.

```
static INLINE void encode32le(u8 *dst, u32 val)
2 {
4     dst[0] = val & 0xFF;
4     dst[1] = (val >> 8) & 0xFF;
6     dst[2] = (val >> 16) & 0xFF;
6     dst[3] = (val >> 24) & 0xFF;
}
```

Until now, none of the macros writes output to the SRAM. This task is handled by the *SRD()* macro, which makes extensive use of the *encode32le()* macro, which is shown above. This macro needs 48 right shifts and 4 AND operations, which we can save because of the direct access of the separate bytes. The *encode32le()* is used 4 times within the *SRD()* macro, which leads to a saving of $52 \cdot 4 = 208$ cycles with every call of the *SRD()* macro. As the *SRD()* macro is called 5 times within the encryption process, we can save 1,040 CPU cycles.

7.3 Results

7.3.1 Memory Usage

Table 7.1 shows the memory allocation in flash memory. In more detail, Table 7.1 provides (i) the size of the flash memory which is used to store the program code, (ii) the required size of flash memory for the storage of static arrays, like S-boxes for instance, (iii) the total size of program code and static arrays, and (iv) the associated AVR device, i.e. the smallest device on which the implementation of the cipher can be executed without errors. The indication of the total flash memory size in percentage is computed in relation to the maximum available size of flash memory of the corresponding device which name is given in the last column.

Table 7.1: Memory allocation in flash memory of Assembly implementations

Cipher	Program Code	Static Arrays	Total Memory		Device
	[byte]	[byte]	[byte]	[percentage]	
AES	1154	266	1420	17,33%	ATmega8
Dragon (M)	25102	2048	27150	82,86%	ATmega32
Dragon (F)	4850	2048	6898	84,20%	ATmega8
LEX	1486	5120	6606	80,64%	ATmega8
LEX V2	1332	266	1598	19,51%	ATmega8
Salsa20 (M)	2984	0	2984	36,43%	ATmega8
Salsa20 (F)	1452	0	1452	17,72%	ATmega8
Sosemanuk (M)	44648	2048	46696	71,25%	ATmega64
Sosemanuk (F)	9092	2048	11140	67,99%	ATmega16

Corresponding to Table 7.1 we can determine the existence of two groups in relation to the required size for the storage of the program code of the ciphers. The first group is made up by the AES, both versions of Salsa20, both versions of LEX and Dragon (F) which require only a small consumption of flash memory and can therefore be executed without problems on a ATmega8. In the second group we find Dragon (M) and both versions of Sosemanuk. These 3 implementations are not able to run on the small ATmega8 and the macro based version of Sosemanuk can only be processed on an ATmega64.

In contrast to Table 6.2 in Chapter 6.3, Table 7.2 exhibits only four columns. More precisely, Table 7.2 shows only the total size of the required SRAM (i) and the smallest device on which the cipher is executable. In Assembly language there is no need for separate views on the SRAM because the consumption of the SRAM is compiler-independent and handmade.

In consideration of the consumption of SRAM all ciphers would be executable on an ATmega8, which holds a maximum of 1024 bytes. But the limiting factor is not

Table 7.2: Memory allocation in SRAM of Assembly implementations

Cipher	Total SRAM		Device
	[byte]	[percentage]	
AES	224	21,88%	ATmega8
Dragon (M)	560	27,34%	ATmega32
Dragon (F)	560	54,69%	ATmega8
LEX	304	29,69%	ATmega8
LEX V2	304	29,69%	ATmega8
Salsa20 (M)	280	27,34%	ATmega8
Salsa20 (F)	280	27,34%	ATmega8
Sosemanuk (M)	712	17,38%	ATmega64
Sosemanuk (F)	712	69,53%	ATmega16

the SRAM usage, but the required amount of flash memory. Looking at Table 7.2 we observe that AES, Salsa20 and LEX need very few bytes in SRAM. Dragon and Sosemanuk require SRAM of two or three orders of magnitude greater than the rest of the ciphers.

Considering the function based versions and LEX V2, we notice that flash memory needs are low for AES, Salsa20 and LEX, moderate for Dragon and high for Sosemanuk. High amounts of program code also typically indicate a high grade of implementation complexity. This is especially true for Sosemanuk. In terms of SRAM usage, AES, Salsa20, and LEX are again most efficient, followed by Dragon and Sosemanuk.

7.3.2 Performance

Performance benchmarks are provided in Table 7.3 and Table 7.4. Table 7.3 shows the number of cycles for the initialization, key setup, IV setup and encryption for each cipher. Table 7.4 focuses on the throughput of the encryption function for each cipher while Table 7.3 gives the number of cycles for one block size.

The amount of cycles needed for initialization is composed of the required activities to initialize the microprocessor. More precisely, the key, the IV and the plaintext are stored in the SRAM, the stack pointer is initialized with the highest address of the SRAM and some definitions are made. The number of cycles needed for initialization can be reduced by storing the key as fixed value to the flash memory, but most of the given number of cycles for initialization are required to bring the microprocessor in a state to work properly. Key setup and IV setup are independent from the block size of the cipher and can be compared directly. The value given in the last column of Table

Table 7.3: Performance of initialization, key setup, IV setup, encryption of Assembly implementations (all numbers given are measured CPU cycles)

Cipher	Initialization	Key Setup	IV Setup	Encryption
AES	192	1535	57	5113
Dragon (M)	756	538	21232	16648
Dragon (F)	756	537	23680	17527
LEX	316	1484	5216	5502
LEX V2	313	1575	5595	5963
Salsa20 (M)	464	199	60	17812
Salsa20 (F)	460	199	60	18400
Sosemanuk (M)	514	14627	8559	8739
Sosemanuk (F)	519	15252	9143	9459

7.3 must be correlated with the value in Column 2 of Table 7.4, which is done in this table.

Concerning the key setup, Salsa20 is the fastest cipher with only 199 needed cycles. It is followed by Dragon with approximately 2.5 times more required cycles. The AES and LEX logically feature nearly the same cycle count for key setup (three times the number of cycles required by Dragon). Sosemanuk exhibits the largest value in Column 3 and needs almost ten times more cycles as for instance AES.

Table 7.4: Throughput of encryption of Assembly implementations

Cipher	Block Size [byte]	Ratio [cycles/byte]	Throughput [bytes/sec] @8MHz	Time Memory Tradeoff Metric [cycles/byte][byte]
AES	16	319,56	25034	453779
Dragon (M)	128	130,06	61509	3531197
Dragon (F)	128	136,93	58424	944541
LEX	40	137,55	58161	908655
LEX V2	40	149,08	53664	238222
Salsa20 (M)	64	278,31	28745	830485
Salsa20 (F)	64	287,50	27826	417450
Sosemanuk (M)	80	109,24	73235	5100954
Sosemanuk (F)	80	118,24	67660	1317166

The IV setup is dominated by the AES and Salsa20, which both require less than 100 cycles. Therefore, these two ciphers are the first choice if the cipher should be frequently reinitialized. The next ciphers in line are LEX, Sosemanuk and Dragon. Dragon requires more than 20,000 cycles for the IV which leads to the suggestion of only using Dragon when the focus is not laid on frequent reinitialization.

The ratio value given in Column 3 of Table 7.4 is the quotient of Column 2 (which shows the block size of the ciphers) and Column 5 of Table 7.3. The throughput given in Column 4 is computed by dividing the CPU clock (8 MHz) by the ratio value in Column 3. The last column of Table 7.4 introduces a time-memory tradeoff metric, i.e. the product of the ratio of cycles per keystream byte (shown in Column 3) and the total amount of flash memory (shown in Column 4 of Table 7.1). Low values of this metric indicate high efficiency in the time-memory tradeoff.

According to Table 7.4 Sosemanuk and Dragon are the fastest ciphers, followed by LEX, Salsa20 and AES. Notable is the fact that all ciphers of the eSTREAM project do the encryption faster than the AES, as specified in the call for participation. But if we examine the last column, only two ciphers exhibit a better tradeoff metric, namely LEX and Salsa20.

8 Summary and Future Work

8.1 Summary

This thesis provides the first implementation results for Dragon, HC-128, LEX, Salsa20 and Sosemanuk on 8-bit microcontrollers and therefore answers the question of how efficient modern stream ciphers can be implemented on small embedded microcontrollers that are also constrained in memory resources.

We confirm that all studied stream ciphers reach higher speeds at keystream generation than the AES. In terms of memory, Salsa20 and LEX can be implemented almost as compactly as the AES, while Dragon and Sosemanuk require noticeably more memory resources and may be sub-optimum for embedded applications with very low memory constraints (although they reach higher throughput rates). Nevertheless it is worth mentioning that all ciphers are executable on smaller devices in Assembly language, compared to our C implementations. The only cipher that is assessed to be not suitable for small embedded microcontrollers is HC-128, because of its high SRAM memory requirements.

Overall, considering the time-memory tradeoff metric, LEX and Salsa20 have turned out to yield significantly better results than AES. More detailed information is given below in four different bar graphs showing the SRAM consumption, the flash memory consumption, the throughput, and a time memory tradeoff.

NOTE: In C language we choose the original implementations to be included in the graph. HC-128 is only implemented in C language. Hence, no green bar is indicated in the HC-128 column. In Assembly language we choose the function based versions (in case of LEX we choose LEX V2) for the computation of the plots.

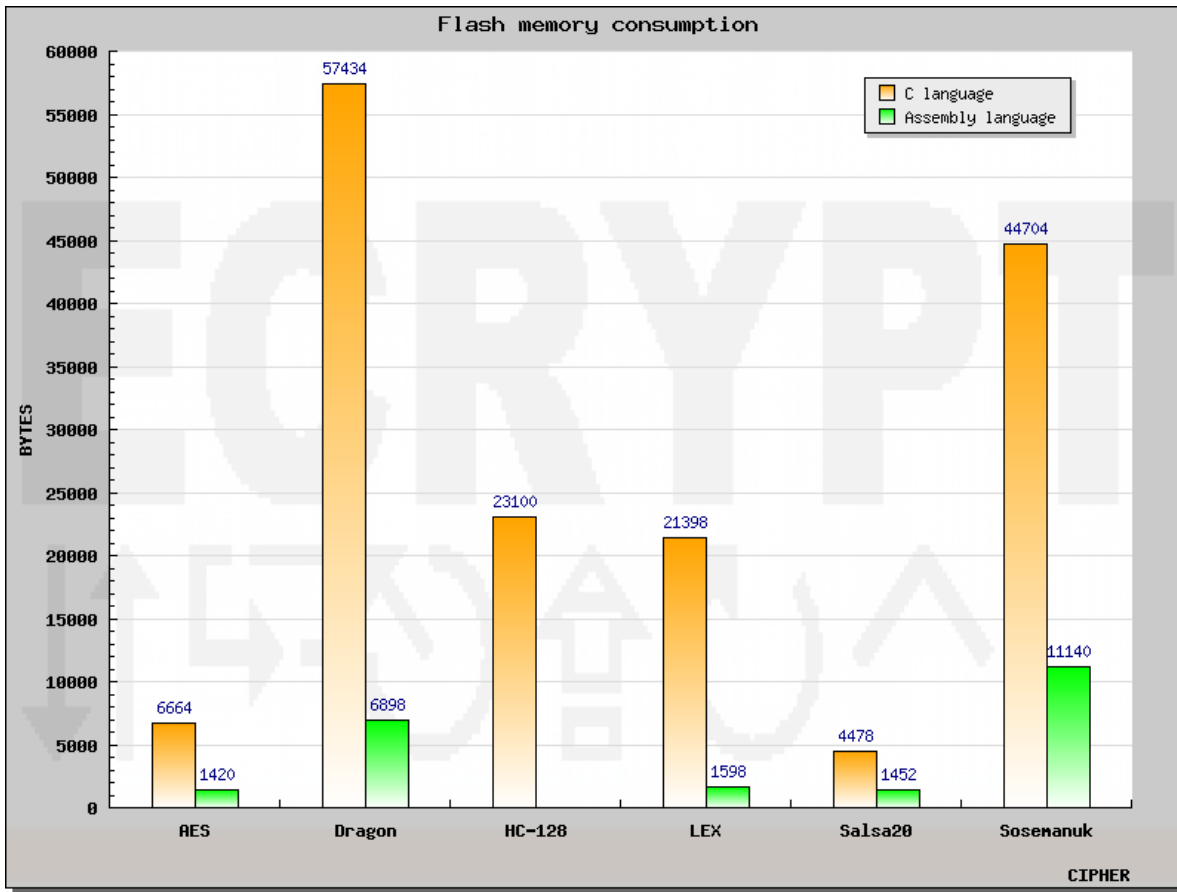


Figure 8.1: Comparison of C and Assembly language implementations: flash memory consumption

Figure 8.1 shows the flash memory consumption of our implementations, both in C language as well as in Assembly language. Here, LEX achieves the greatest amount of savings. The Assembly version needs only 7% of the flash memory requirements of the C implementation. The next best saving is accomplished by Dragon with 12%. Following these two ciphers are the AES (21%), Sosemanuk (25%), and Salsa20 (32%).

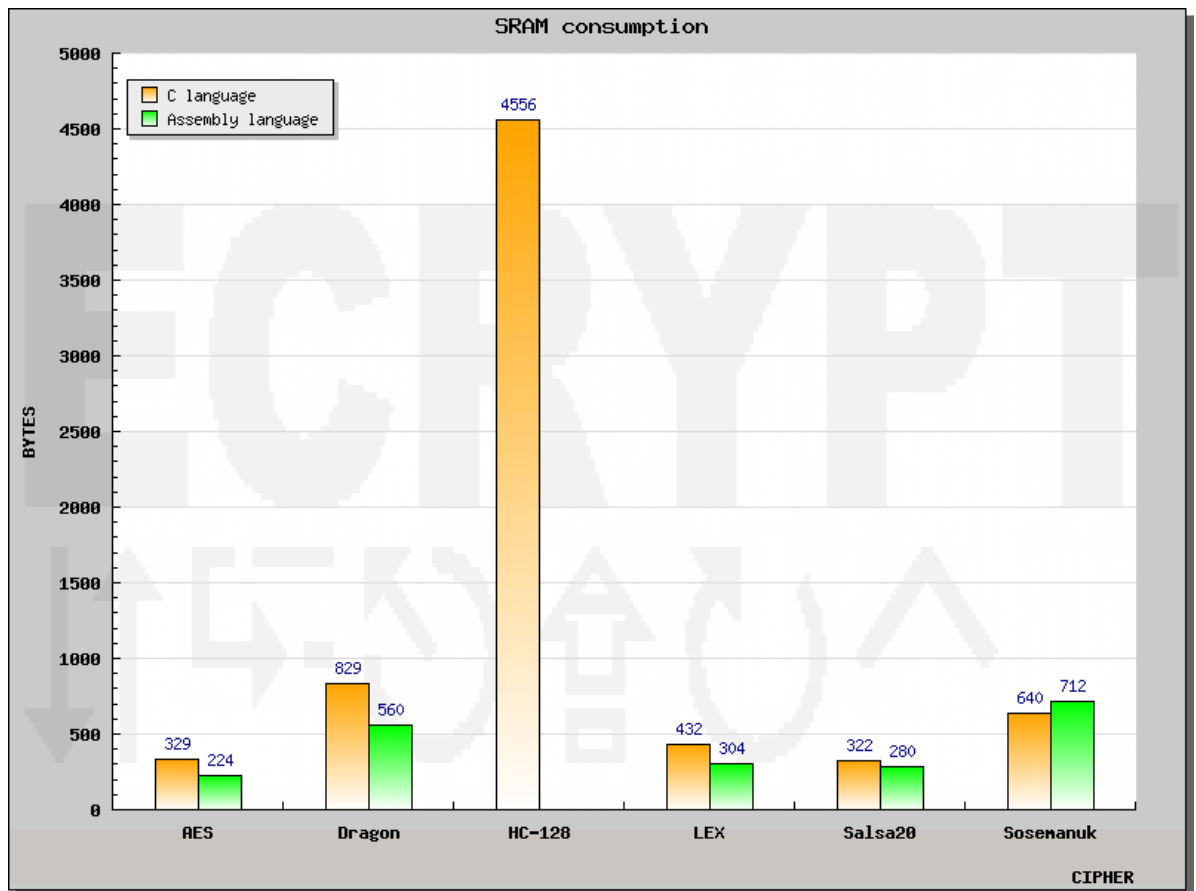


Figure 8.2: Comparison of C and Assembly language implementations: SRAM consumption

As Figure 8.2 shows, except for Sosemanuk, all Assembly language implementations require less SRAM than the C language implementations. Both, the AES and Dragon share the first place in this category. Both need only 68% of SRAM compared to the C language version. LEX (70%) and Salsa20 (86%) take the further places. Sosemanuk needs 11% more SRAM in Assembly language. Further, we see the huge usage of SRAM of the HC-128 cipher.

NOTE: The SRAM values in C language are optimized values not including the use of the stack. If we include the stack, Sosemanuk needs less SRAM in Assembly too.

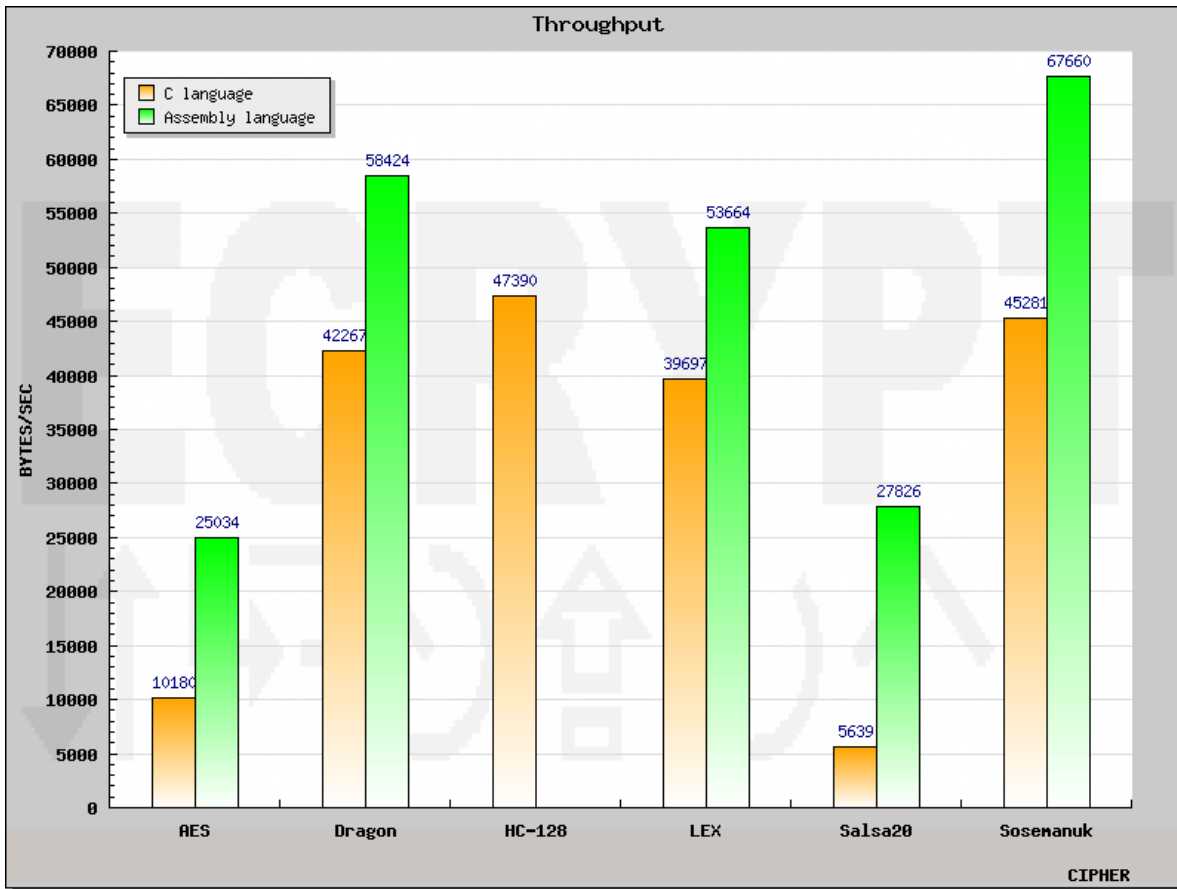


Figure 8.3: Comparison of C and Assembly language implementations: throughput

Figure 8.3 shows the throughput data of our implementations. Here, Salsa20 yields the greatest enhancement concerning throughput with a 493% higher throughput rate. Salsa20 is followed by AES (246%), Sosemanuk (149%), Dragon (138%), and LEX (135%).

NOTE: The macro-based versions of our Assembly implementations reach higher throughput rates, but as the focus lies on minimal flash memory usage, these versions are not shown in the graph.

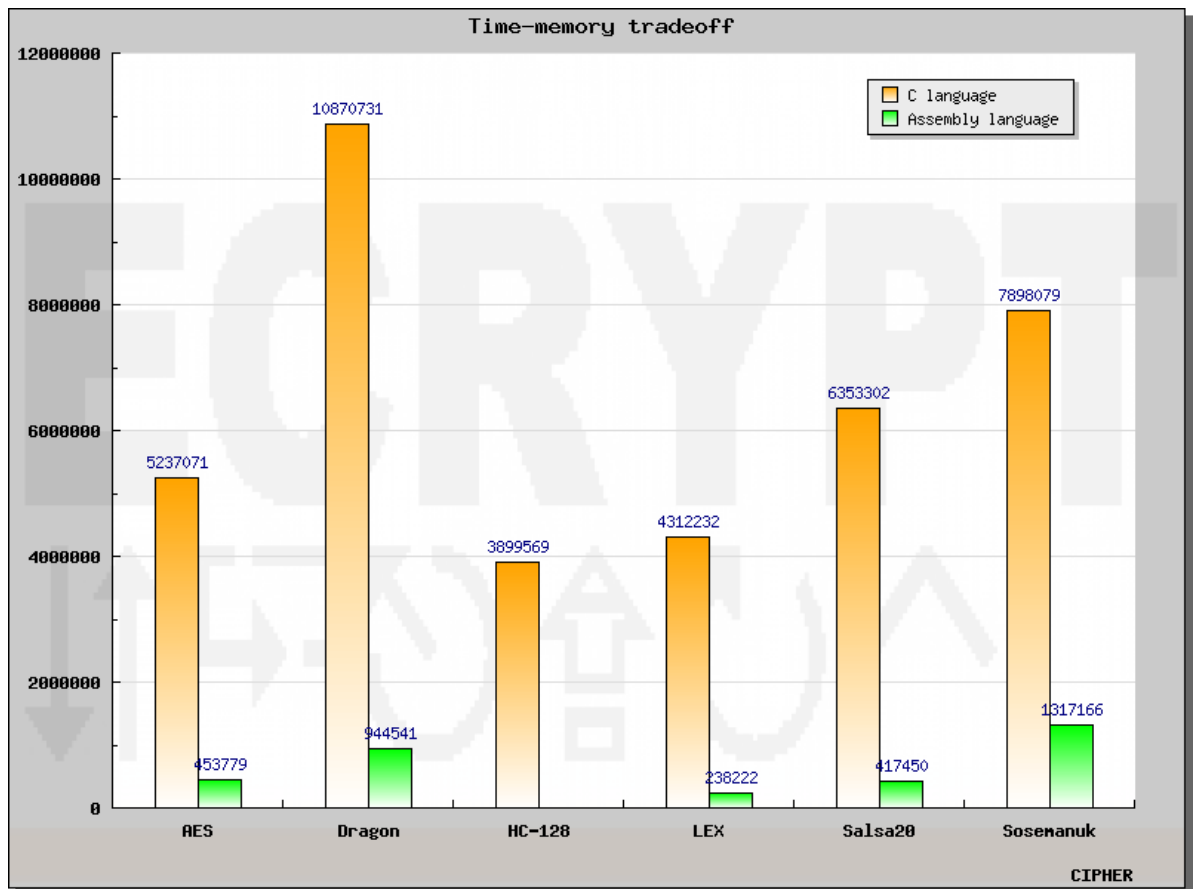


Figure 8.4: Comparison of C and Assembly language implementations: time memory tradeoff

The time memory tradeoff metric as given in Figure 8.4 is the product of the used flash memory and the encryption ratio (cycles needed for encryption multiplied by the block size of the cipher). It is a good value for benchmarking how efficiently the Assembly versions run on the microcontrollers, compared to the C language implementations. LEX achieves the greatest reduction and the time memory tradeoff value reduces to only 5.5% of the C implementation value, which corresponds to a nearly 18 times higher efficiency rate. Salsa20 is 15 times more efficient (6.6%), AES and Dragon are 12 times more efficient (8.7%), and Sosemanuk is at least 6 times more efficient (16.7%) implemented in Assembly language.

8.2 Future Work

The next obvious step is the implementation of the eSTREAM Phase 3 candidates of Profile I. Some ciphers have been advanced from the non-focused group to the focused group of Phase 3 (i.e. CryptMT, NLS, and Rabbit). These ciphers should also be considered for Assembly language implementations.

The implementation in Assembly language offers great potential for optimizing the ciphers and the chance to fit them perfectly to a particular device. This probably helps to determine the best stream cipher from the perspective of the the industry and of course, the eSTREAM project as well.

Furthermore it makes sense to implement some of the stream ciphers belonging to Profile II of the eSTREAM project. These ciphers are indeed optimized for the implementation on hardware but nevertheless it may be reasonable to implement them, too. Certain ciphers of Profile II (like Trivium for instance) are designed for eventual use in constrained environments. If it is possible to create, for instance, an 8-bit output version of Trivium, a software implementation can also achieve good results in terms of improved speed and memory consumption.

A Appendix

In this chapter we list the source files of our C and Assembly language implementations. All listed files are included on the attached compact disc.

A.1 C language implementations

- AES
 - aes-avr.c (26,204 Bytes)
 - Makefile (16,926 Bytes)
- Dragon
 - dragon-avr.c (26,434 Bytes)
 - Makefile (16,933 Bytes)
- HC-128
 - hc128-avr.c (16,517 Bytes)
 - Makefile (16,937 Bytes)
- LEX
 - lex-avr.c (39,080 Bytes)
 - Makefile (16,929 Bytes)
- Salsa20
 - salsa20-avr.c (7,988 Bytes)
 - Makefile (16,936 Bytes)
 - salsa20-avro.c (9,855 Bytes)
 - Makefile (16,936 Bytes)
- Sosemanuk
 - sosemanuk-avrm.c (25,805 Bytes)
 - Makefile (16,937 Bytes)
 - sosemanuk-avrf.c (31,506 Bytes)
 - Makefile (16,937 Bytes)

A.2 Assembly language implementations

- AES
 - aes.asm (14,498 Bytes)
- Dragon
 - dragonf.asm (77,400 Bytes)
 - dragonm.asm (86,591 Bytes)
- LEX
 - lex-aes.asm (17,665 Bytes)
 - lex-lex.asm (42,064 Bytes)
- Salsa20
 - salsa20f.asm (16,180 Bytes)
 - salsa20m.asm (16,258 Bytes)
- Sosemanuk
 - sosemanukf.asm (131,259 Bytes)
 - sosemanukm.asm (129,805 Bytes)

B Bibliography

- [1] ATMEL Corporation. Available from: <http://www.atmel.com>.
- [2] AVR-sizeX. Available from: <http://alt.kreatives-chaos.com/download/avr-sizeX.exe>.
- [3] AVR Studio 4.12, Atmel Corporation, Build 460, November 2005. Available from: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725.
- [4] Mfile - A Makefile generator for AVR-GCC Motivation for Mfile, 4th November 2003. Available from: <http://www.sax.de/~joerg/mfile/>.
- [5] WinAVR : avr-gcc for Windows, Release 20060421, 21th April 2006. Available from: <http://winavr.sourceforge.net/>.
- [6] Call for Stream Cipher Primitives, Version 1.3, 12th April 2005. eSTREAM, ECRYPT Stream Cipher Project, 2005. Available from: <http://www.ecrypt.eu.org/stream/call/>.
- [7] eSTREAM – Update 1, September 2, 2005. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/057, 2005. Available from: <http://www.ecrypt.eu.org/stream/papersdir/057.pdf>.
- [8] FIPS 197. Announcing the AES, November 2001. Available from: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
- [9] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert. SOSEMANUK, a fast software-oriented stream cipher. Available from: http://www.ecrypt.eu.org/stream/p2ciphers/sosemanuk/sosemanuk_p2.pdf.
- [10] Elwyn Berlekamp and James Massey. Berlekamp-massey algorithm for finding the shortest linear feedback shift register (lfsr) for a given output sequence, 1968. Available from: <http://planetmath.org/encyclopedia/BerlekampMasseyAlgorithm.html>.
- [11] D. J. Bernstein. Notes on the ECRYPT Stream Cipher project (eSTREAM). Available from: <http://cr.yp.to/streamciphers.html>.
- [12] Alex Biryukov. A new 128-bit key stream cipher LEX. Available from: http://www.ecrypt.eu.org/stream/p2ciphers/lex/lex_p2.zip.

-
- [13] Christophe De Cannière. eSTREAM testing framework. eSTREAM, ECRYPT Stream Cipher Project. Available from: <http://www.ecrypt.eu.org/stream/perf/>.
- [14] K. Chen, M. Henricksen, W. Millan, J. Fuller, L. Simpson, E. Dawson and H. Lee, and S. Moon. Dragon: A fast word based stream cipher. Available from: http://www.ecrypt.eu.org/stream/p2ciphers/dragon/dragon_p2.pdf.
- [15] Brian Gladman. Brian Gladman's Home Page. Available from: <http://fp.gladman.plus.com/AES/index.htm>.
- [16] Simon Guillem-Lessard. Chiffrements par blocs rijndael. Available from: http://www.uqtr.ca/~delisle/Crypto/prives/blocs_rijndael.php.
- [17] Thomas Johansson Patrik Ekdahl. A new version of the stream cipher snow. Available from: <http://www.it.lth.se/cryptology/snow/snow20.pdf>.
- [18] Christian Roepke. Implementation of the aes encryption on an 8-bit avr microcontroller. Available from: http://www.christianroepke.de/studium/praktikum_b/aes_encryption.asm.
- [19] Lars Knudsen Ross Anderson, Eli Biham. A candidate block cipher for the advanced encryption standard. Available from: <http://www.cl.cam.ac.uk/~simrja14/Papers/serpent.pdf>.
- [20] Wikipedia. Common object file format (coff). Available from: <http://en.wikipedia.org/wiki/COFF>.
- [21] Wikipedia. Executable and linking format (elf, formerly called extensible linking format). Available from: http://en.wikipedia.org/wiki/Executable_and_Linkable_Format.
- [22] Wikipedia. Specification of the aes rcon table. Available from: http://en.wikipedia.org/wiki/Rijndael_key_schedule#Rcon.
- [23] Hongjun Wu. The stream cipher HC-128. Available from: http://www.ecrypt.eu.org/stream/p2ciphers/hc256/hc128_p2.pdf.
- [24] Hongjun Wu and Bart Preneel. Differential-Linear Attacks against the Stream Cipher Phelix. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/056. Available from: <http://www.ecrypt.eu.org/stream/papersdir/2006/056.pdf>.
- [25] Hongjun Wu and Bart Preneel. Key Recovery Attack on Py and Pypy with Chosen IVs. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/052. Available from: <http://www.ecrypt.eu.org/stream/papersdir/2006/052.pdf>.