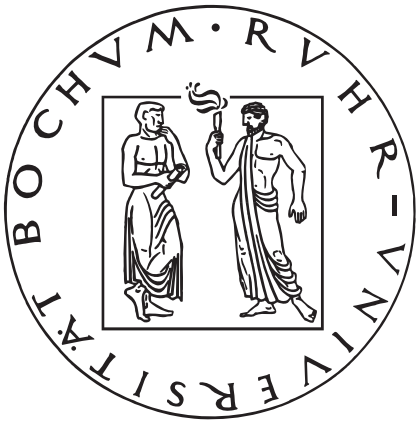
Performance Analysis of Contemporary Light-weight Cryptographic Algorithms on a Smart Card Microcontroller

Schriftliche Prüfungsarbeit
für die Bachelor-Prüfung des Studiengangs
Angewandte Informatik
an der Ruhr-Universität Bochum

vorgelegt von:

Sören Rinne



23. April 2007

Themensteller: Prof. Dr.-Ing. Christof Paar
2. Prüfer: Prof. Dr. rer. nat. Jörg Schwenk

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für ein anderes Examen eingereichten Arbeit.

Ich erkläre weiterhin, dass ich die Arbeit nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen und bildliche Darstellungen und dergleichen.

Datum	Unterschrift

Abstract

In this work we present a performance analysis of software implementations of newly-proposed block ciphers. These so-called light-weight algorithms are especially designed for the domain of ubiquitous computing, with the goal of having low software or hardware implementation costs. As the analysis focuses on the special properties of embedded devices, we measured code size and clock cycles due to the restricted properties in cost (given by memory consumption) and energy needs (given by execution time) of ubiquitous devices. The analyzed light-weight ciphers in this work are DESL, HIGHT, SEA, TEA, and XTEA. We implemented all ciphers in assembly language and compared them with reference assembly implementations of the AES, DES, DESX, and IDEA. As programming platform we used a Smart Card with an 8-bit microcontroller.

Our implementation results show that newly-proposed light-weight cryptography algorithms can only keep up with standardized and optimized algorithms in memory consumption. In terms of throughput the ciphers failed to outperform most of the members of the reference group.

Zusammenfassung

Die vorliegende Arbeit präsentiert eine Performance Analyse von Software-Implementierungen neu vorgestellter Blockchiffren. Sogenannte light-weight Algorithmen mit dem Ziel geringer Software- bzw. Hardware-Implementierungskosten sind speziell für Ubiquitous Computing entworfen worden. Da die Analyse sich auf die speziellen Eigenschaften von eingebetteten Systemen konzentriert, wurden Codegröße und Taktzyklen gemessen. Die gemessenen Größen bestimmen die Kosten (gegeben durch den Speicherbedarf) und den Energiebedarf (gegeben durch die Ausführungszeit) solcher Geräte. Die analysierten Chiffren in dieser Arbeit sind DESL, HIGHT, SEA, TEA und XTEA. Alle Algorithmen wurden in Assembler implementiert und mit Referenzimplementierungen in Assembler von AES, DES, DESX und IDEA verglichen. Als Programmierplattform wurde eine Smart Card mit 8-bit Mikrocontroller benutzt.

Die Ergebnisse zeigen, dass light-weight Kryptographiealgorithmen nur im Speicherbedarf mit standardisierten und optimierten Algorithmen mithalten können. Hinsichtlich Durchsatz schafften die meisten light-weight Chiffren es nicht, alle Vertreter der Referenzgruppe zu übertreffen.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter we introduce our motivation on comparing newly proposed ciphers, which are especially designed for the domain of ubiquitous computing, to existing wide spreaded and deeply analyzed ciphers like the AES. Afterwards we give an overview of the organization of the thesis.

1.1 Motivation

Computer aided systems have attended us for decades. From the slide rule to the contemporary personal computer people have developed devices to support us in every day live. These systems helped us to automatize procedures, to accelerate processes, to ease technical issues. Whereas in former times computers filled almost whole rooms like Konrad Zuse's Z series todays devices even fit in a price tag like RFID [12] (Radio Frequency Identification) devices.

The use of computing devices in the last century shows some interesting developments. Whereas in the 1950s people get used to mainframes (one computer used by many people) in the 1970s the personal computer (one computer used by one person) enters offices and domestic areas. At the same time the use of mainframes decreased. During the turn of the century so called ubiquitous computing (many computers used by one person) became more and more relevant. Ubiquitous devices integrate computing into every day life, while staying perceptually invisible to the users who would be able to interact with these objects naturally. Many things we use during a day are equipped with a microprocessor. From the coffee machine in the morning, the car on our way to work, the chip card to enter the building, and the price tags in the shop to the lamps that automatically switch on in the evening; each of them includes a microcontroller.

Due to the growth of ubiquitous devices the question of security also rises. Should every communication between these devices be readable for every one? Consider following situation: doctor Arthur equips Bob with a digital long-time pulse meter. The device stores Bobs pulse during 24 hours. The measured values can easily be read by doctor Arthur via radio frequency. If this communication is not encrypted the official discretion makes no sense if the device tells Bobs values to every one standing with a reader close to him. Analog scenarios can easily be found concerning every

ubiquitous device that shares confidential information. Unencrypted communication can always be eavesdropped or manipulated. Even Caesar encrypted his messages for his commanders at the front.

Since ubiquitous devices should be perceptually invisible they have to be very small, should not be very power consumptive and not too expensive. Microcontroller perfectly fit in this pattern. They are a type of microprocessors with low power consumption emphasizing self-sufficiency and cost-effectiveness. By having a small size they are constrained in some respects like power consumption or storage space. In contrast to a general-purpose microprocessor as they are used in personal computers microcontroller work on smaller blocks like 8 bit compared to 32-bit processors. They have also a reduced instruction set (RISC) which leads to a lack of instructions compared to general-purpose microprocessors. Anyhow its size and cost-effectiveness is unbeatable.

By using constrained devices for encrypting messages there is an approach to use a special sort of algorithm: Light-weight cryptography algorithms. These algorithms are especially designed for light-weight environment like RFID tags. Thus far, there have been two approaches for providing cryptographic primitives for such situations. On the one hand optimized low-cost implementations for standardized and trusted algorithms and on the other hand new designed ciphers with the goal of having low software implementation costs.

In our work we implemented representatives for both two groups. Our AES, DES, DESX, and IDEA implementations represent the first group, DESL, HIGHT, SEA, TEA, and XTEA the second one. We implemented the ciphers on a constrained device, a Smart Card. Our used FunCard is equipped with an AVR ATmega163 which can be compared to an AVR ATmega128 embedded in wide spread pervasive devices like ubiquitous sensor networks (Mica Motes i.e.). With our work we intended to analyze the performance of contemporary light-weight cryptography algorithms in comparison to standardized algorithms.

1.2 Organization of the Thesis

The thesis is basically organized in three sections: the hardware, the software, and the result section. In the hardware section we will give an overview of the Smart Card history, see what specifies Funcards and the ATmega family and end on how to communicate with the Smart Card. We will go on with an introduction to our focused ciphers presenting design parameters and basic data. This leads us to the software section discussing first the assembly programming language. Further on we will take a look at the tool chain we used and how we finally communicate with our Smart Card concerning given software on both Smart Card and terminal side.

Afterwards we will discuss optimization goals and how we implemented the several ciphers. Finally we introduce our results regarding memory usage and performance. We conclude with a summary and propose further work.

2 An Introduction to Smart Cards

In this chapter we will first take an insight in the history of Smart Cards from plastic cards to contemporary Java Smart Cards with embedded microcontrollers. After taking a closer look at the ATmega family of Atmel, especially the ATmega163 mounted in our Smart Card, we move on to a passage about the communication with the chip card.

2.1 History of Smart Cards

The spreading of plastic cards started in the USA in the early 1950s. Due to its low-priced and long-lasting material PVC the plastic card outrivaled the existing paperboard cards. The Diners-Club was the first company that handed out cards fully made of plastic which enabled the owner to pay for something without having any cash. As a result of impersonation and its impossibility of being read by a machine a magnetic stripe was added to the plastic card.

The microelectronics achieved considerable progress in the 1970s which enabled them to integrate data memory and an arithmetic-logic unit on a silicon die with a size of only a few square millimeters. The idea of implementing such an integrated circuit on a plastic card has already been applied for a patent in 1968 in Germany by Jürgen Dethloff and Helmut Grötrupp. In 1970 Kunitaka Arimura filed a similar patent in Japan. But in 1974 as Roland Moreno filed a patent in France the semiconductor industry was able to produce the integrated circuits for an acceptable price in an acceptable quantity.

The French PTT first applied chip cards as phonecards in 1984. One year later the german phone company started a test run of phonecards with different technologies. The chipcard came off as winner and in the late 1990s over 200 millions of phonecards were circulating in Germany.

Microcontroller Chipcards The first appliance of Microcontroller chipcards took place in the financial sector in France. The possibility of storing secret keys in a secure way and to run cryptographic algorithms on a chipcard realized an offline-paysystem with a high security level. Due to its reprogrammability the functionality

of a Smart Card was only bounded by its memory capacity and the computing power of its arithmetic-logic unit.

Today microprocessor chipcards are used in wide areas e.g. health cards in the german health care system or decoder cards for decoding paytv channels.

Contactless Smart Cards Another type of processor card that became more and more important in the 21st century is the contactless Smart Card. The chip inside is communicating with the card reader through RFID (radio-frequency identification) induction technology. These cards require only close proximity to an antenna to interact. They are often used when transactions must be processed quickly or hands-free, such as on mass transit systems like ski-lifts, where Smart Cards can be used without even removing them from ones pocket or wallet. Contactless Smart Cards are widespread in big cities for public transportation on nearly every continent nowadays.

Cryptographic Smart Cards There are also Smart Cards available equipped with specialized cryptographic hardware. With these cards one can use crypto algorithms such as RSA or DSA which are implemented in hardware on board instead of in software. Such Smart Cards are mainly used for digital signature and secure identification. The most widely used cryptographics in the cards are DES or 3DES and RSA. The key set is usually loaded (DES) or generated (RSA) on a personalisation stage. A DES key set is typically used for signing or encrypting data on a host system. RSA is used in banking or passport cards for signing transaction data.

Java Chipcards The so called Java Card is a variant of the programming language Java, which allows Java Card Applets to be executed on chipcards [34]. Java Card Applets are Java Applets based on a reduced Java-Standard. A Java Chipcard can store several Applets which does not affect each other. Java Card Applets can be installed supplementary and are independent from the hardware of the chip card, such as Java programs. These Applets communicate exclusively over APDUs (Application Protocol Data Units; cp. section 2.4) with a connected card reader.

2.2 Funcards and the ATmega Family

So called "Funcards" are low-priced Smart Cards used for time recording or access control. They have also been used for Smart Card emulation in the pay tv sector. Funcards are freely available for about 8 Euro apiece and can be used without subscribing a non-disclosure agreement. They are usually based on an Atmel AVR microcontroller like the AT90S8515 or the ATmega163 combined with a program

memory (Flash ROM), an EEPROM like the 24C256, and RAM. We implemented our ciphers on a FunCard, whose microcontroller, the ATmega163, can easily be compared to microcontrollers used in wireless sensor networks, for example the Mica Mote 2 which is equipped with an ATmega128(L).

The Atmel AVR family distributed by the Atmel Corporation are 8-bit RISC microcontrollers which can be classified into three broad groups:

- tinyAVRs
  - 1-8kB program memory
  - 8-20-pin package
  - Limited peripheral set
- megaAVRs
  - 4-256kB program memory
  - 28-100-pin package
  - Extended instruction set (Multiply instructions and instructions for handling larger program memories)
  - Extensive peripheral set
- Application specific AVRs
  - megaAVRs with special features not found on the other members of the AVR family, such as LCD controller, USB controller, advanced PWM etc.

The AVR is a Harvard Architecture machine with programs and data stored separately. Typical Harvard type machines store programs in permanent or semi-permanent memory and data in volatile memory.

Figure 2.1 shows the block diagram of a processor-chipcard.

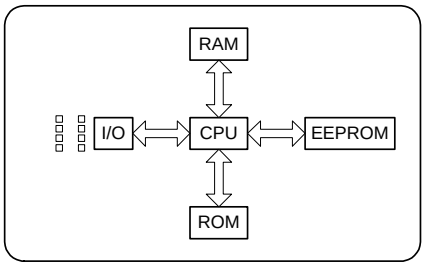


Figure 2.1: Block diagram of a processor-chipcard

2.3 The ATmega163

We implemented our algorithms on a Funcard with an ATmega163 microcontroller (see Figure 2.2). Since the ATmega163 belongs to the AVR family it has an 8-bit RISC processor, 130 most single clock cycle execution instructions, 32 8-bit working registers R0-R31, 512 bytes EEPROM, and 1024 bytes internal SRAM. The last six working registers R26-R31 can be used as three 16-bit pointers X, Y and Z.

The advantage of the RISC architecture is sparse use of clock cycles. Most instructions are executed within one clock cycle which leads to a command throughput of nearly one instruction per clock cycle. Hence an AVR microcontroller clocked with 8 MHz achieves nearly 8 MIPS. Instructions which access the SRAM or EEPROM typically need 2-3 clock cycles. For a better throughput memory-intensive applications should be avoided.

Figure 2.2: Sample picture of a Funcard with an AVR microcontroller

2.4 Communication with the Smart Card

Interaction with the Smart Card requires the possibility of communication between the chip card and a terminal. To do so we have a one-wire connection only. Due to this one-way layout, chip card and terminal can only send data alternating. In each case the other party has to receive at that time. Figure 2.3 shows the contacts of the chip card with its specific function. The five used contacts are supply voltage (Vcc), input for reset (RST), input for clock (CLK), ground (GND), and input/output for serial communication (I/O). The programming voltage (Vpp) contact and the RFU (reserved for future use) contacts are not used for communication with the Smart Card.

The initiator of the communication is always the terminal. The chip card only responds to the commands sent by the terminal. It never sends data by itself. Therefore we have a master-slave behaviour in which the terminal acts as master and the chip card as slave.

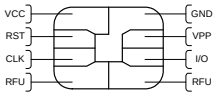


Figure 2.3: Smart Card contacts

After putting the chip card in a terminal the five used contacts (VCC, RST, CLK, GND, I/O) get activated in correct order. Thereupon the card does a Power-On-Reset and sends an Answer to Reset (ATR) back to the terminal. The ATR containing miscellaneous card parameters will be analyzed and the first command will be send to the card. The chip card executes it, generates an answer and sends it back to the terminal. This command-answer procedure will be continued until the deactivation of the chip card (cp. Figure 2.4).

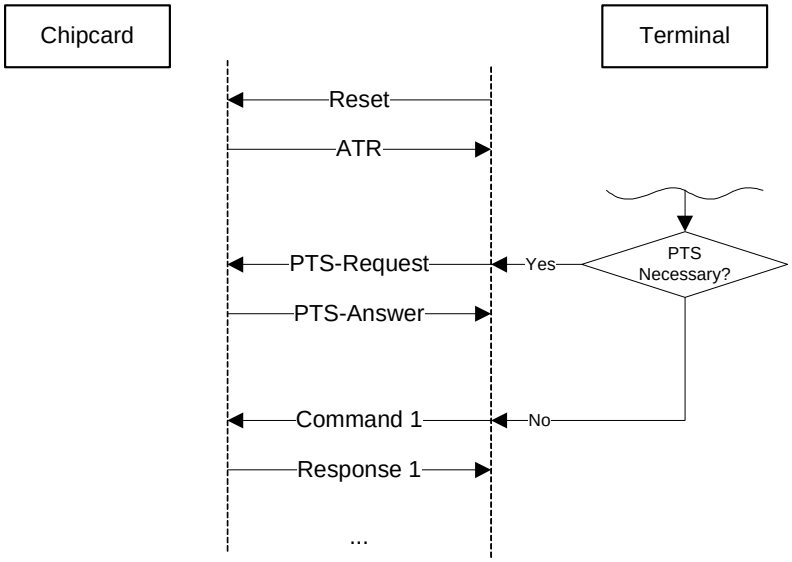


Figure 2.4: Answer to Reset (ATR), Protocol Type Select (PTS) and the first command-response-pair

There are different possibilities to establish a connection with the chip card or how to synchronize both partys in case of a transmission breakdown. The special commands, the responding answers and the behaviour on disruption can be found in the different transmission protocols. Table 2.1 gives an overview [29].

Two protocols became international accepted: the T=0 protocol were standardized in 1989 (ISO/IEC 7816-3), the T=1 protocol in 1992 as addendum to ISO/IEC 7816-3.

Transmission protocol	Description
T=0	asynchronous, half-duplex, byte-orientated, specified in ISO/IEC 7816-3
T=1	asynchronous, half-duplex, block-orientated, specified in ISO/IEC 7816-3 Amd. 1
T=2	asynchronous, full-duplex, block-orientated, specified in ISO/IEC 10536-4
T=3	full-duplex, not specified
T=4	asynchronous, half-duplex, byte-orientated, extension of T=0, not specified
T=5 ... T=13	reserved for future use, not specified
T=14	for national appliance, not specified in ISO
T=15	reserved for future use, not specified

Table 2.1: Transmission protocols

For communication with our Funcard we used the T=1 protocol. T=1 is a block-orientated protocol and starts after the chip card sent the ATR or a PTS has been completed successfully. The first block is sent by the terminal, the next one comes from the chip card. The communication carries on with alternating broadcasting.

One transmission block consists of a leading prologue field, the information field and a finishing epilogue field. Prologue and epilogue field are mandatory. The information field is optional and contains data for the application layer. That is a command APDU for, or a response APDU from the chip card. Figure 2.5 shows a common T=1 block.

Prologue field			Information field	Epilogue field
Node Adress NAD	Protocol Control Byte PCB	Length LEN	APDU	EDC
1 Byte	1 Byte	1 Byte	0 ... 254 Byte	1 ... 2 Byte

Figure 2.5: Common structure of a T=1 block

Figure 2.6 gives an overview of the communication chain between the Smart Card and the terminal. The Smart Card Operating System written in C language will be compiled and linked together with the cipher, which is written in assembly language. The linked file will be burned to the Flash ROM of the Smart Card. The T=1 protocol defines the communication between the Smart Card micro controller and the Card Reader. The Card Reader is connected via USB to the computer with the terminal application running. This application communicates through the PC/SC driver. The Smart Card Operating System itself provides the basic operations for communication with the terminal like the T=1 protocol implementation or the APDU handling. It basically consists of an infinite loop in which the OS waits for an incoming APDU. This reflects the master-slave behaviour with the chip card as

slave waiting for commands from the master device.

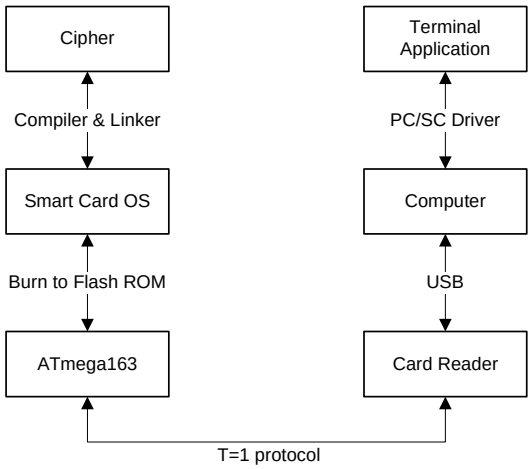


Figure 2.6: Smart Card communication chain

3 Focused Ciphers

This section provides a short description of each cipher. An overview of the ciphers' parameters is given in Table 3.1, ordered alphabetically. The values of SEA can be chosen, so the values that fit our implementation are given in this table.

Cipher	AES	DES	DESL	DESX	HIGHT	IDEA	SEA	TEA	XTEA
Block length	128	64	64	64	64	64	96	64	64
Key length	128	56	56	56	128	128	96	128	128
Rounds	10	16	16	16	32	8	141	32	64

Table 3.1: Characteristic sizes of the focused ciphers

We spend roughly the same time on implementing each cipher to achieve the same preconditions for every one to receive a fair benchmarking process. In our performance analysis we will use the AES and IDEA as reference implementations for the other ciphers. These two ciphers were implementations of the chair for Communication Security at the Ruhr University of Bochum. As SEA implementation we used an existing one of the author [9].

We implemented a quite huge range of ciphers, starting with the DES family providing only a medium security level with a 56-bit key. Others cipher like HIGHT use a 128-bit key providing a high security level but work on smaller block sizes than the AES to fulfill the needs of a restricted environment. The cipher SEA is kept flexible in all parameters depending on the one hand on the processor word size of the processor used and on the other hand the security goal and performance needed so that each user may configure it as required.

3.1 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [7], also known as Rijndael[1], is the successor of the Data Encryption Standard (DES). It was announced by the National Institute of Standards and Technology (NIST) as a U.S. FIPS in 2001. The cipher

[1] Rijndael supports a larger range of block and key sizes; AES has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits

developed by J. Daemen and V. Rijmen was the winner of a 5-year standardization process.

The AES is designed to be fast in both software and hardware. This is, amongst others, the reason why Rijndael was chosen to be the AES. The cipher is relative easy to implement and requires little memory.

AES is a block cipher using an 128 bit block with an 128, 192 or 256 bit key as input. It operates on a 4×4 array of bytes, termed the state. Each round of AES consists of four stages, namely `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`. `AddRoundKey` combines each byte of a state with the round key which is derived from the cipher key using a key schedule. `SubBytes` is a non-linear substitution step using a lookup table to replace each byte with another. `ShiftRows` shifts each row of the state cyclically a certain number of steps. `MixColumns` operates on the columns of the state, combining the bytes in each column using a linear transformation. The AES is known to be quite efficient on 8-bit architectures due to its byte-oriented design. Our assembler implementation of the AES is inspired by the AES implementation of Brian Gladman [13].

Known attacks In April 2005, D. J. Bernstein announced a cache timing attack [3] requiring over 200 million chosen plaintexts. In October 2005, D. A. Osvik together with A. Shamir and E. Tromer presented a paper with several cache timing attacks [27] against AES. The attack was able to obtain an entire key after 800 writes in 65 milliseconds.

3.2 International Data Encryption Algorithm (IDEA)

The International Data Encryption Algorithm (IDEA) [23] is another block cipher which works on 64-bit blocks using a 128-bit key. It is based on a novel generalization of the Feistel structure with 8 identical rounds followed by an output transformation. Each round consists of bit-operations, namely exclusive-or, addition mod 2^{16}, and a 16-bit multiplication mod 2^{16}. Figure 3.1 shows one round of the IDEA Feistel network.

IDEA is designed for being implemented in software, mirroring in the small footprint of our implementation. Slow table look ups also do not apply to the cipher, as it does not have any substitution boxes.

Known attacks Existing attacks are attacks on weak keys or exhaustive key searches on the 128-bit key space.

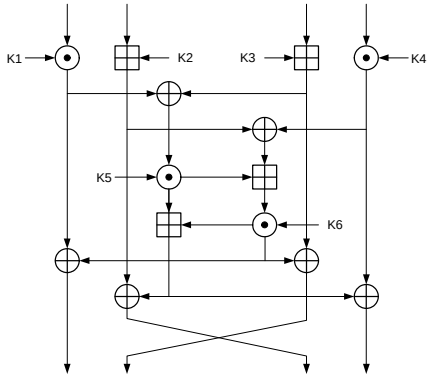


Figure 3.1: One encryption round of IDEA

3.3 Scalable Encryption Algorithm (SEA)

The Scalable Encryption Algorithm ($SEA_{n,b}$) [32] is designed to be parametric in plaintext/key and processor size. In dependence on given hardware parameters like processor word size the SEA parameters will be chosen. The main advantages of SEA are efficient combination of encryption/decryption and "on-the-fly" key derivation. It was invented to be an algorithm for small embedded applications like RFID or Smart Cards. SEA is designed for being implemented in software.

$SEA_{n,b}$ parameters in our case are plaintext/key size $n = 96$, processor word size $b = 8$, and number of words per Feistel branch $n_b = \frac{n}{2b} = 6$. Therefore we have a suggested number of cipher rounds of $n_r = \frac{3n}{4} + 2 \cdot (n_b + \lfloor b/2 \rfloor) = 93$.

The cipher is targeted for processors with a limited instruction set and therefore uses only bit operations such as exclusive-or, word rotation, bit rotation, addition mod 2^b, and a substitution box. Figure 3.2 shows one encryption and key round, where R denotes the word rotation, r the bit rotation, and S the substitution box.

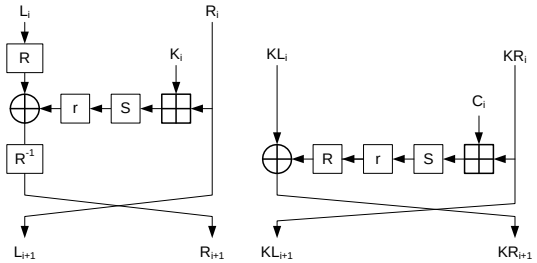


Figure 3.2: Encrypt/Decrypt round and key round of SEA

3.4 Data Encryption Standard (DES)

The Data Encryption Standard (DES) [11] is a cipher selected as an official Federal Information Processing Standard (FIPS) for the United States in 1976. As a block cipher DES operates on blocks with a size of 64 bits. The key also consists of 64 bits; only 56 of these are actually used by the algorithm, the other ones are parity check bits.

Like other block ciphers, DES must be used in a certain mode of operation (e.g. ECB, CBC, CFB) if applied to a message longer than the operational block size of 64 bits.

The overall structure consists of a so called Feistel network with 16 identical base rounds with 8 substitution boxes (S-Boxes), an initial permutation, a final permutation, and a separate key schedule. The whole cipher consists only of bit operations, namely shifts, bit-permutations and exclusive-or operations. Figure 3.3 shows the enciphering computation.

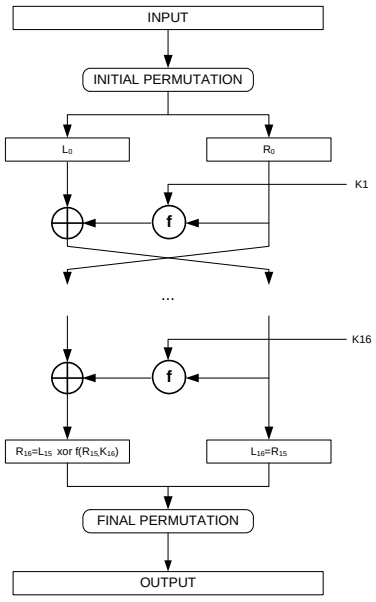


Figure 3.3: Encryption structure of DES

Due to its bit-permutations DES is designed to be implemented in hardware. Permutations, especially bit-permutations, are almost free in hardware but very expensive in software. This fact is clearly reflected in the footprint of the DES family in our software implementation.

Known attacks DES is deemed to be broken. The only practical attack is a brute force attack. There are several confirmed DES cracker such as the EFF DES Cracker [10] or the COPACOBANA [22]. Theoretical attacks are differential cryptanalysis, linear cryptanalysis, and Davies' attack [5].

3.4.1 Extension to DES (DESX)

The block cipher DESX (or DES-X) [20] is an extension to DES to improve some weaknesses of its predecessor. It is defined by $DESX_{K,K_1,K_2}(M) = K_2 \oplus DES_K(M \oplus K_1)$. It was originally suggested by Ron Rivest in 1984 to protect the cipher DES against exhaustive key-search attacks. DESX is said to be substantially more resistant than DES.

3.4.2 DES Lightweight Extension (DESL)

Like the above mentioned DESX the DES Lightweight Extension [28] (DESL) is an extension to DES to comply with the requirements of small computational devices like RFID devices or Smart Cards. It was suggested by A. Poschmann et al. in 2006 as a new alternative for ultra-low-cost encryption. To decrease chip size requirements it uses only one S-Box repeated eight times. It therefore requires 38% less transistors than the smallest DES implementation published.

3.5 HIGHT

HIGHT is another block cipher proposed by Deukjo Hong et al. [16] which is working on a 64-bit block length and a 128-bit key length. It was proposed to be used at ubiquitous computing devices such as a sensor in USN or a RFID tag at CHES '06. It is predestinated for its application area due to its low-resource hardware implementation. The specialty of HIGHT is its use of simple operations such as exclusive-or, addition mod 2^8, and bitwise rotation. HIGHT is designed for being implemented in hardware to have a low cost of gates.

The cipher is a variant of generalized Feistel network. It consists of an initial transformation, 32 rounds using 4 subkeys at a time, a final transformation and a key schedule producing 128 subkeys. HIGHTs key schedule algorithm is designed to keep the original value of the master key after generating all whitening keys and all subkeys. Therefore the subkeys are generated on the fly in encryption and decryption. Figure 3.4 shows the encryption process of HIGHT.

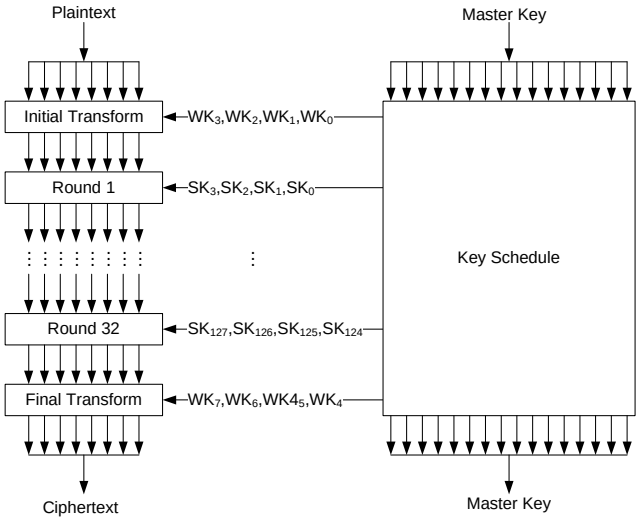


Figure 3.4: Encryption process of HIGHT

3.6 Tiny Encryption Algorithm (TEA)

The specialty of the Tiny Encryption Algorithm (TEA) [35] is its simplicity of description and implementation. It was first presented at the Fast Software Encryption workshop in 1994 by David Wheeler and Roger Needham.

TEA is a block cipher operating on 64-bit blocks with a 128-bit key. The Feistel structure is dominated by suggested 64 identical rounds consisting of bit-operations like shifts, addition/subtraction mod 2^8 and exclusive-or operations. Figure 3.5 shows two rounds of encryption.

TEA is designed to be implemented in software. The cipher only consists of few less than 10 lines of C source code. Instead of a complicated program it uses a large number of iterations. It is based on a weak non linear iteration which will be rerun enough rounds to make it secure. The cipher uses little set up time and has no preset tables.

A number of revisions of TEA has been designed in order to obliterate some weaknesses of the original version. The revisions of the cipher, Block TEA (often referred to as XTEA) and XXTEA (published in 1998), were needed to secure the cipher.

Known attacks TEA suffers from equivalent keys, which means that each key is equivalent to three others. Therefore the effective key size is 126 bits instead of

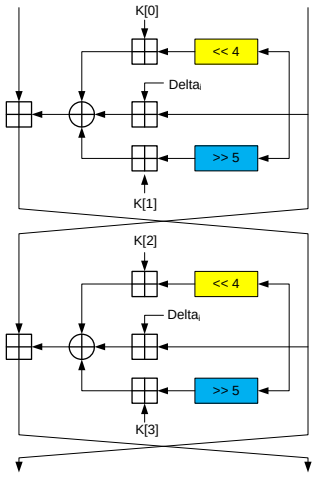


Figure 3.5: Two Feistel rounds of TEA

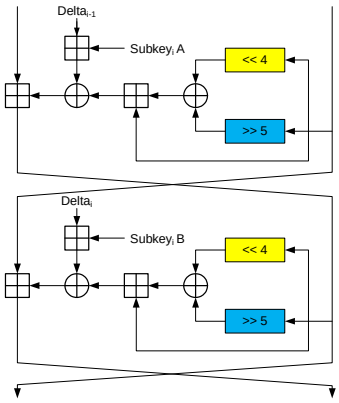


Figure 3.6: Two Feistel rounds of XTEA

128 bits. Due to this weakness a method for hacking the Microsoft's Xbox game console, where TEA was used as a hash function, was developed [30]. The cipher is also vulnerable to a related-key attack which requires 2^{23} chosen plaintexts under a related-key pair, with 2^{32} time complexity [19].

Table 3.2 gives a summary of attacks [18][19][15][14][25][17][21][31].

Cipher	Type of Attack	Rounds	Plaintexts	Time
TEA	Equivalent Keys	Any	1	2^{126}
TEA	Related-Key	64	2^{23}	2^{32}
TEA	SAC Distinguisher	10	2^{25}	-
TEA	Partitioning	2-8	-	-
TEA	Impossible Differential	11	$2^{52.5}$	2^{84}
XTEA	Impossible Differential	14	$2^{62.5}$	2^{85}
XTEA	Differential	15	2^{59}	2^{120}
XTEA	Truncated Differential	23	$2^{20.55}$	$2^{120.65}$
XTEA	Related-Key	27	$2^{20.5}$	$2^{115.15}$
BlockTEA	Differential	-	2^{34}	-
XXTEA	Distinguisher	6	2^{80}	-

Table 3.2: Summary of Attacks on TEA and Variants

3.7 Extension to TEA (XTEA)

As mentioned before the effective key length of TEA is 126 bits not 128. So in 1996 Needham and Wheeler made two adjustments [26]. The first was to adjust the key schedule and the second was to introduce the key material more slowly. With these adjustments the weaknesses should be repaired and the simplicity is almost retained. Figure 3.6 shows two rounds of encryption.

As Table 3.2 shows there are still attacks on the new variants, mostly based on a small number of rounds.

4 Framework Set-Up and Tool Chain

In this section we will describe how we implemented the ciphers on an AVR Microcontroller. A short introduction to Assembler, the software development tools, and how to measure clock cycles and throughput follow.

4.1 Assembler vs. High Level Programming Languages

Due to the memory constraints of a Smart Card we needed to downsize the source code of the ciphers. A standard implementation in C language would be too large and too slow. As mentioned before we have strict restrictions on memory in constrained devices. Furthermore most transactions on these devices must be accomplished in a limited time slot. To benefit from the increased speed of processing assembly instructions we wrote our ciphers in assembly language. Assembly language is a low-level language which will be translated into the target computer's machine code by a utility program called assembler. This sort of language is close to a one to one correspondence between symbolic instructions and executable machine codes. Assembly languages also include directives to the assembler, directives to the linker, directives for organizing data space, and macros. Macros can be used to combine several assembly language instructions into a recurrent code block with changing parameters (as well as other purposes). Symbolic assembly language instructions correspond to individual executable machine instructions.

Unlike assembly languages High-level languages are abstract. Typically a single high level instruction is translated into several executable machine language instructions and therefore produces a large amount of source code. High level programming languages are much easier for less skilled programmers to work in and allow faster development times than working in assembly language. Assembly language is much harder to be programmed. The programmer must pay attention to far more detail and must have intimate knowledge of the processor in use. Assembler gives us the chance to control our target machine on a low level. We have the possibility to manipulate every single working register in the way we need it. In return we have to care about every single register so that it won't be overwritten for example. In a high-level programming language like C or C++ there is no need to attend registers. But high quality hand crafted assembly language programs can run much faster and

use much less memory and other resources than a similar program written in a high level language. Speed increases of two to 20 times faster are fairly common.

So Assembler fits best to achieve fast cipher implementations with low memory costs.

4.2 AVR-GCC Assembler

As our micro controller is a member of the AVR family we used the GNU AVR-GCC compiler. The AVR-GCC is a freeware C compiler and assembler that is made available through the GNU project[1].

4.3 Implementing on an AVR Microcontroller

Every cipher normally comes with a standard implementation. We had many different programming languages, e.g. C, AVR-Assembler, or Java. As mentioned above, in order to reduce the size of the code and to make it faster we re-implemented them in AVR-gcc-Assembly language. To achieve a fair benchmarking process we used the existing implementations as a starting point for the assembly implementations.

The ciphers were neither solely optimized for performance only nor for extremely small code size. Instead we tried to yield a good trade-off between both and keep the source code clear. In chapter 5 we will take a closer look at cipher-specific implementations.

4.4 Source-code Testing Environment

As we expected to run the ciphers on an 8-Bit Smart-Card we used a small part of a Smart Card Operating System (OS) to wrap it around our Assembler-Code. It is written in C-Language and enables us to see if the variable handoff on the Smart Card is working correctly. As variables we had to transfer the plaintext/cipher and the key to the Assembler functions. The testing environment runs in a simulator which will be described later on.

By using this environment there was no need to burn the complete Smart Card OS with the current cipher on the Smart Card every time we changed the code to debug our implementation.

[1] `http://gcc.gnu.org/`

4.5 Smart Card OS

We considered porting an existing OS to our Smart Card. The OS is mainly used in wireless sensor networks. Since the embedded microcontrollers in both devices, our FunCard and wireless sensor network devices like Mica Motes, are nearly the same (ATmega128 vs. ATmega163) we aimed for porting the TinyOS [6]. TinyOS is an open-source operating system developed for wireless embedded sensor networks. Due to the must of low power consumption of wireless devices to achieve a long lifetime, the device is basically kept in a stand-by mode in TinyOS. The OS is event driven, so that only an event, for example triggered by the temperature sensor, wakes up the device. This spots the main difference between the OS on our Smart Card and the OS on Motes. The SCOS runs the whole time in a loop waiting for commands from the terminal whereas the TinyOS only interacts when an event is triggered. Our approach in porting TinyOS was to use interrupt service routines on our Smart Card to keep the events, on which the whole OS is based. Unfortunately it is not possible to use interrupt service routines because of the wiring from the controller to the outside world, namely the contacts of the Smart Card. In the SCOS we have a Master Input/Slave Output (MISO) direction, whereas we would need Master Output/Slave Input (MOSI) direction to realize interrupts on a Smart Card. The only interrupt is reasonamle to use would be the SPI (Serial Peripheral Interface) interrupt which will be called when a serial transfer is completed. This transfer has to be occur on one of the ports of the microcontroller which are not connected to one of the contacts of the Smart Card. Therefore it makes no sense to port an event based OS to a device based on an infinite loop.

Another alternative was the free operating system SOSSE (Simple Operating System for Smartcard Education)[2]. As it is no longer maintained we made use of a part of another slight Smart Card OS (SCOS) for our testing environment instead. With kindly permission we used an existing operating system from the IAIK of the Graz University of Technology[3].

The OS itself first needed some changes to get it running with our used compiler. It has been developed with the AVR development tool "CrossStudio" from Rowley Associates Limited[4]. Using the CrossStudio compiler a 2 byte address of a variable is handed off from a C function to an Assembler function in registers 26 and 27, whereas the AVR-GCC compiler we made use of would place the pointer in registers 24 and 25. Thus we had to change most of the core functions so that our variables won't be overwritten.

[2] `http://www.mbsks.franken.de/sosse/`
[3] `http://www.iaik.tugraz.at/`
[4] `http://www.rowley.co.uk/avr/index.htm`

The SCOS itself runs on the Smart Card providing the basic operations for communication with the terminal, e.g. T=1 protocol implementation or APDU handling. It basically consists of an infinite loop in which the SCOS waits for an incoming command APDU. After receiving a command APDU it is passed to the command handler, an if-clause for executing the tasks asked for (like encoding or decoding procedures). After execution a response APDU will be created. This will be send back to the terminal and the infinite loop starts again. Before starting the main loop the ATR will be sent to the terminal.

Figure 4.1 gives an overview of the core routines of the Smart Card OS.

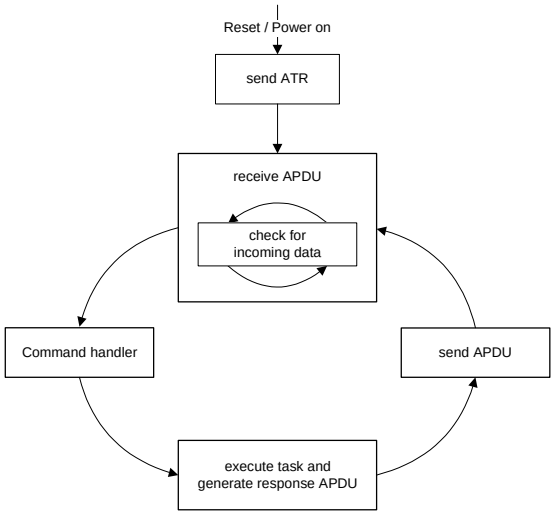


Figure 4.1: Block diagram of the Smart Card OS

As terminal application we used a program written in C language which uses the WinSCard-API [24] library to communicate with the Smart Card. Figure 4.2 shows the UI for using and testing the encryption/decryption routines burned on the Card.

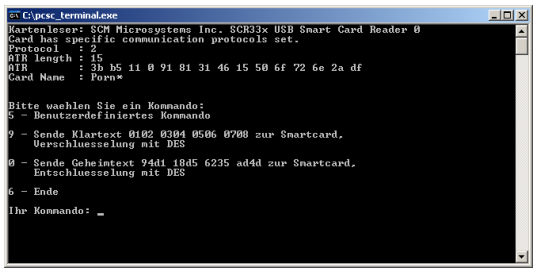


Figure 4.2: User Interface on Windows platform

4.6 Development Tools

For the software development we used the tool "Programmer's Notepad 2" [33]. This is an open source text editor with special features for coders hosted on the Windows platform. Programmer's Notepad 2 contains an automatic makefile execution. It compiles the C-Code of the testing environment, assembles the cipher's Assembler-Code, links it to an $.elf$-file, and then converts it to a $.cof$-file. After this procedure has run without errors we used the output file from Programmer's Notepad 2 to execute and debug the code in "AVR Studio 4" [1] and simulate it on an ATmega163 device. AVR Studio 4 is an Integrated Development Environment (IDE) for writing and debugging AVR applications on the Windows platform. With this program we can use common debugging tools such as watching registers and obtain the CPU cycle counts. The latter enables us to measure clock cycles for benchmarking throughput.

To get the whole Smart Card OS running on the Smart Card itself instead of using a simulator we converted the $.elf$-file into an intel hex file. This hex file can be burned on the Smart Card's Flash ROM by using the "CAS Interface Studio" [8].

Code size is measured in Programmer's Notepad 2. After compiling, assembling, linking, and converting it shows a summary of the used Flash ROM size. To get only the Assembler-Code size we did the compiling procedure two times. First with the C environment together with the Assembler-functions and then the C-Code only. By computing the difference between the first and the second result we received the code size used by the cipher.

5 Implementation and Optimization

In this section we will discuss the implementation goals in general and take a closer look at the specific cipher implementations.

5.1 General Implementation Ideas

There is always the question of how to implement a cipher. On the one hand it can be optimized for code size or on the other hand optimized for speed. Here we have a conflict of goals. It is not possible to optimize the source code for both aims, code size and speed. We have to consider which one of these goals is adequate in our special case and will fit our needs.

To optimize source code for speed we can use word instructions in Assembler or work with macros instead of functions. Word instructions like `MOVW` copies a register pair instead of just one register but needs the same clock cycles as `MOV`. Macros for example are code snippets which will be pasted in the source code by the compiler every time it is retrieved. Compared to functions we can save the time to push the return address on the stack. Needless to say that a macro repeated five times needs five times more code size than an equal function called as often.

5.1.1 Loop Unrolling

Loops appear in nearly every program and are therefore a rewarding aim for optimization. To break open loops we need to know how often we have to run through the loop. By applying loop enrollment to functions we save n jumps, n similies, and $n-1$ decrements or increments on n loop passes. In assembly language we can save one clock cycle per `DEC` plus two clock cycles per `BRNE` for every n. If there is a call of a subroutine in a loop we can save three clock cycles per `RCALL` and another four clock cycles per `RET` in the subroutine. This sums up to $10 \cdot n$ saved clock cycles in the worst case. For further details on AVR instructions see [2], p. 174ff.

5.1.2 Macros vs. Functions

To unroll loops we only need to write the code inside the loop n times. Fortunately it can be easier done by using macros as mentioned above. The great advantage of a macro is its parameter handoff. Every time we call a macro we can change the parameters handed over:

```
.MACRO myMacro parameter1, parameter2
  ADD \parameter1, \parameter2
.ENDM

...

myMacro R17, R18
myMacro R19, R20
```

By contrast a function is static. It will work for example every time it is called on the same registers:

```
myFunction:
  ADD R17, R18
  RET

...

RCALL myFunction
```

5.1.3 Bit-Slicing

To optimize DES Eli Biham presented his paper *A Fast New DES Implementation in Software* at Fast Software Encryption 4 in 1997 [4]. There he described a non-standard implementation of DES whereby the algorithm is broken down to AND, OR, NOT, and XOR gates. These gates are implemented as machine instructions. This means, that for example an 64-bit machine will execute DES 64 times in parallel. Depending on the word size of a specific architecture this can be significantly faster than traditional DES implementations.

The method of bit-slicing could be advantageous to other cipher implementations as well. This would be an interesting topic to be discussed in future work.

5.2 Implementation of the Ciphers

In this section we will take a closer look at the ciphers implementations. By viewing the pseudo code of some ciphers we describe how we realized them.

5.2.1 DES Family

In our implementation we mostly did not make use of macros. Every part of the DES feistel network such as the final permutation FP or the permuted choice PC-1 were implemented as functions. Thus our source code mainly consists of the functions `initialperm`, the initial permutation, `PC1` which computes the permuted choice PC-1, `leftshift` which does the left shifts, `PC2` that computes the permuted choice PC-2, `Expansion` which represents the expansion function E, `XOR48` which computes the 48-bit exclusice-or, `sboxen` which handles the S-Boxes, `Permutation_und_XOR32` which does the permutation P and computes the 32-bit exclusive-or, `swappen` which swaps the two halfs L and R, and `finalperm`, the final permutation. So the pseudo code of our implementation is as follows:

```
Encryption {
  initialperm;
  PC_1;
  for n=1 to 16 {
    leftshift;
    PC_2;
    Expansion;
    XOR48;
    sboxen;
    Permutation_und_XOR32;
    swappen;
  }
  swappen;
  finalperm;
}
```

To speed up the table lookup of the S-Boxes we reordered the values of each table. Normally a DES S-Box is represented as a table with four rows and 16 columns. The 6-bit input block is splitted in two parts to get row and column of the 4-bit output value as shown in Figure 5.1.

Expecting 111010_2 as input we have to look at the 2nd row and 13th column of the respective S-Box to get the output. By putting the values from 0 to 63 one after

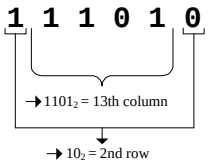


Figure 5.1: Dividing the 6-bit input into row and column of S-Box

another in the S-Box we see that the values are ordered in zigzag in each pair of two rows (Figure 5.2).

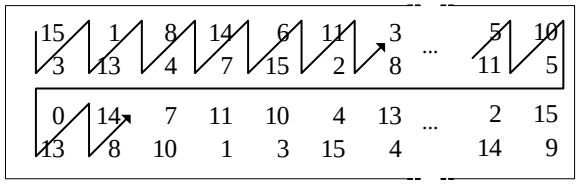


Figure 5.2: S-Box 2, normal order

We reordered the tables so that we got an one-dimensional array of values (Figure 5.3). Now the output of the S-Box is computed by adding the input value to the start address of the table. The value located at the offset address is the output.

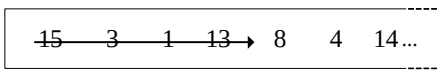


Figure 5.3: S-Box 2, reordered

We computed the new array using the algorithm which can be found in the addendum (see Listing A.1). It reorders the values and produces the right AVR Assembler code at the same time.

As DESX and DESL are only small changes to DES we took the DES implementation and added the additional features like the use of three keys in DESX and the substitution of the S-Boxes with one single S-Box in DESL.

5.2.2 AES

As an AES implementation we used a highly optimized version programmed by Kerstin Lemke-Rust et al. from the chair of Communication Security at the Ruhr University Bochum including small changes with respect to our compiler.

5.2.3 IDEA

For IDEA we used an existing speed optimized implementation of the chair for Communication Security at the Ruhr University Bochum including small changes with respect to our compiler.

5.2.4 TEA and XTEA

TEA in pseudo code is as follows[1]:

```
void TeaEncryption(long* v, long* k)
{
  unsigned long y=v[0], z=v[1], sum=0,
                delta=0x9E3779B9, n=32;

  while(n-->0) {
    sum += delta;
    y += ((z << 4)+k[0]) ^ (z+sum) ^ ((z >> 5)+k[1]);
    z += ((y << 4)+k[2]) ^ (y+sum) ^ ((y >> 5)+k[3]);
  }

  v[0]=y; v[1]=z;
}
```

The core arithmetic functions of the cipher are addition, xor, and shifts. To keep the code clear we implemented macros for them. So our code mainly consists of the macros `left_shift_registers` which computes the left shift, `add_register` which computes the sum of two registers, `xor_register` to compute the addition modulo 2, and `right_shift_registers` which computes the right shifts. With these macros and some auxiliary macros like `copy_register`, which copies 4 registers, we implemented the two basic equations straight forward in a main loop which runs 32 times. As the authors of TEA mentioned, the cipher is facile to implement in software.

The only macros that make sense to replace in order to reduce the code size are the macros that do the left and right shifts. By trying to replace the macros with loops as we did with HIGHT we had a code size advantage of 5.5% but a speed loss of 13.79%. The advantage is very small and further more out of all proportion to the speed loss so we decided to keep the macros. This applies to XTEA as well.

[1] with key in k[0] - k[3], data in v[0] and v[1]

5.2.5 SEA

We used a part of the assembler standard implementation package provided by the author. The package comes with a speed optimized encryption, a decryption, and a version providing both encryption and decryption (depending on T flag). We used the speed optimized version including small changes with respect to our compiler.

5.2.6 HIGHT

We implemented the cipher HIGHT in two different versions. One is optimized for speed (referred to as HIGHT), the other one for code size (referred to as HIGHT-2). In the speed optimized version we made highly use of macros whereas we replaced some macros with a function in the code size optimized version.

HIGHT in pseudo code is as follows[2]:

```
HightEncryption(P, MK) {
  KeySchedule(MK, WK, SK);
  HightEncryption(P, WK, SK) {
    InitialTransformation(P, X_0, WK_3, WK_2, WK_1, WK_0);
    For i = 0 to 31 {
      RoundFunction(X_i, X_{i+1}, SK_{4i+3}, SK_{4i+2}, SK_{4i+1}, SK_{4i});
    }
    FinalTransformation(X_32, C, WK_7, WK_6, WK_5, WK_4);
  }
}
```

As one can see the round function is called every time with different parameters. The easiest way to realize it is using macros. Since the SK_i only change in the round function we can compute these in macros and compute the rest of the round (which does not have any changing parameters) in a function. Maybe it would speed up the cipher if the SK_i would be pre-computed. So our implementation mainly consists of 32 calculations of the current SK_i with the macro `SK` which represents the key schedule, the call of the round function `round_with_rotate` which computes one round and rotates the plaintext for one byte, and a `transform` macro to compute the initial and final permutation with corresponding parameters.

The difference between HIGHT and HIGHT-2 is that we replaced the round function with macros. That gives us a speed advantage of 21% but also a code size disadvantage of 56% compared with each other (see Table 6.1 & 6.2).

[2] where P denotes the plaintext, MK the master key, WK the whitening key, SK the subkey, X_i the intermediate values, and C the ciphertext

6 Results

This section provides the results of our implementations. The results are compared to our reference implementations AES and IDEA. The AES implementation is optimized for the 8-bit AVR microcontroller environment as well as the lightweight-algorithms DESL, HIGHT, SEA, TEA, and XTEA. The comparison focuses on code size, because memory is important for size and price of a constrained device, and on execution time, e.g. throughput, as execution time responds to the power consumption of a device.

6.1 Memory Usage

As we implemented the ciphers on a Smart Card we have restrictions in the size of available Flash memory and SRAM. The Flash (program) memory of the device is used to store program code or look-up tables, if applicable. The smaller SRAM is used for dynamic access during program execution. Our Funcard equipped with an AVR ATmega163 has 16K Bytes of Flash memory.

Table 6.1 shows the memory allocation in flash memory of every cipher in ascending order. Figure 6.1 visualizes the values ordered by size. Rows and columns highlighted in gray indicate our reference implementations from now on.

Cipher	IDEA	TEA	XTEA	SEA	DESL	AES	DES	DESX	HIGHT-2	HIGHT
Code size	596	1140	1160	2132	3098	3410	4314	4406	5672	8836

Table 6.1: Memory allocation in flash memory in byte

As expected the IDEA cipher is the smallest. This results from its high optimization for code size and its lack of a substitution box. The next one is the cryptanalytically weak TEA followed by its successor XTEA and the SEA implementation. The miscellaneous DES implementations together with the AES are following. The two implementations of HIGHT use the most flash memory of all implementations. We can see that the ciphers written for hardware implementation like the DES family and HIGHT allocate the most memory. This is due to the design objectives of the ciphers. The DES family for example, including DESL, relies on bit permutations

which are almost for free in hardware but very expensive in software. This is even true on an 8-bit microcontroller. The ciphers TEA and XTEA are designed for being implemented in software and rely on weak non linear iterations with enough rounds to make it secure. Since DESL uses only one substitution box it has a slightly smaller footprint than the other family members.

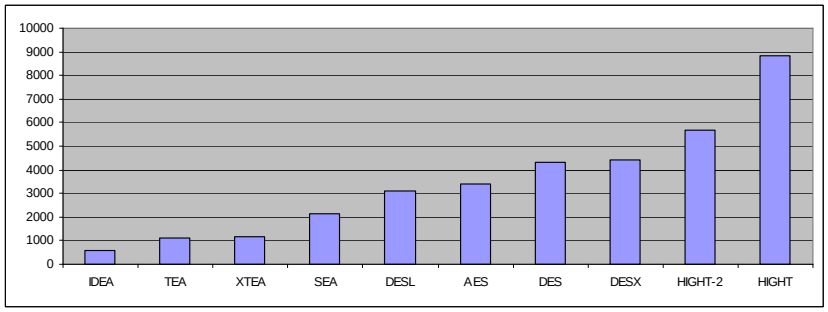


Figure 6.1: Code size of ciphers

6.2 Performance

In the following performance benchmark input and output arrays are of the size of the block size of each cipher. That is to say that we encrypt or decrypt one block with each cipher.

Table 6.2 shows the number of cycles for encryption and decryption for every cipher with ascending order of encryption.

Cipher	HIGHT	IDEA	HIGHT-2	AES	TEA	XTEA	DESL	DES	DESX	SEA
Encryption	2449	2700	2964	3766	6271	6718	8365	8633	8699	9654
Decryption	2449	15393	2964	4558	6299	6718	7885	8154	8220	9654

Table 6.2: Performance of encryption and decryption in measured CPU cycles

As seen in Table 6.1 the first implementation of HIGHT is the one with the maximum use of flash memory. Nevertheless in this benchmark it achieves the lowest number of cycles for encryption and decryption of one block of data. This can be traced back to our highly use of macros which use a lot of space but are fast instead. HIGHT outperforms IDEA, HIGHT-2 beats the AES, TEA and XTEA follow. As we reduced the macros in HIGHT-2 it performs slightly slower than the first implementation. Again the DES family together with SEA come out as last ones.

Table 6.3 and Table 6.4 focus on the throughput of encryption and decryption of each cipher. Column 2 in Table 6.3 and Table 6.4 shows the block size in bit, column 3 recapitulates the count of cycles from Table 6.2. Column 4 is the quotient of column 3 and 2 and column 5 shows the throughput of encryption/decryption in bit/sec. The throughput is computed by dividing the CPU clock (assuming 4 MHz) by the value in column 4.

Cipher	block size [bit]	Encryption [cycles]	Encryption [cycles/bit]	Throughput [bit/sec]
AES	128	3766	29,42	135953
HIGHT	64	2449	38,27	104532
IDEA	64	2700	42,19	94815
HIGHT-2	64	3188	49,81	80301
TEA	64	6271	97,98	40823
$SEA_{96,8}$	96	9654	100,56	39776
XTEA	64	6718	104,97	38107
DESL	64	8365	130,70	30604
DES	64	8633	134,89	29654
DESX	64	8699	135,92	29429

Table 6.3: Throughput of Encryption in bit/sec

Cipher	block size [bit]	Decryption [cycles]	Decryption [cycles/bit]	Throughput [bit/sec]
AES	128	4558	35,61	112330
HIGHT	64	2449	38,27	104532
HIGHT-2	64	3188	49,81	80301
TEA	64	6299	98,42	40641
$SEA_{96,8}$	96	9654	100,56	39776
XTEA	64	6718	104,97	38107
DESL	64	7886	123,22	32463
DES	64	8154	127,41	31396
DESX	64	8220	128,44	31144
IDEA	64	15393	240,52	16631

Table 6.4: Throughput of Decryption in bit/sec

Figures 6.2 and 6.3 reprints the values of Tables 6.2, 6.3 and 6.4 ordered by cycles and respectively by throughput of encryption.

Figure 6.3 shows that the reference AES implementation outperforms all of the newly proposed ciphers. The AES has a significant higher throughput compared to the others. The two HIGHTs do well because of their high speed but cannot outperform the AES due to their block size of 64 bit. IDEA only outperforms most of the ciphers in encryption. This can be traced back to the computation of the inverse of the key in the decryption process. In our implementation the calculation is done by using Fermat's little theorem. The computation uses existing multiplication routines from the encryption, which results in the one hand to a higher amount of clock cycles

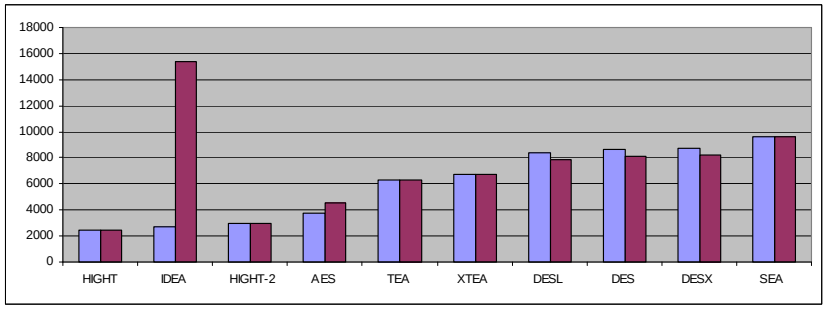


Figure 6.2: Cycle count of Encryption and Decryption

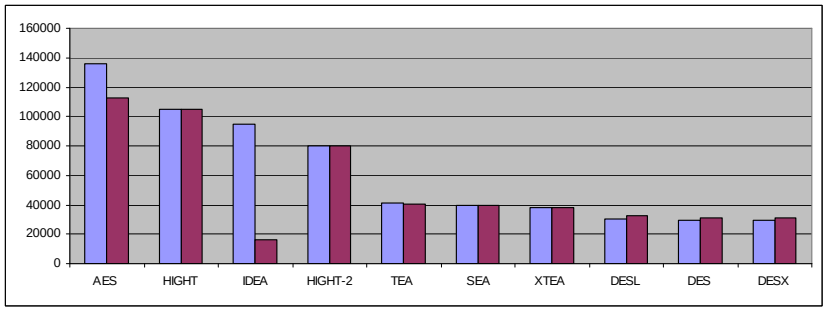


Figure 6.3: Throughput of Encryption and Decryption

during decryption but on the other hand to a small footprint (cp. Table 6.1). The TEA family and SEA follow with a slight spacing but still do better than the DES family including DESL. Even though SEA is slow in encrypting one block, it can keep up with the TEA family because of its bigger block size, namely 96 bit. DESL again performs better than the other members of the DES family due to its single S-Box. DESX is slightly slower than the DES because of the additional XORing part of the keys at the beginning and the end of encryption and decryption. The bad performance of the DES family can be explained by the hardware orientation of the cipher with its bit permutations. As mentioned before, permutations in hardware are nearly for free, whereas they are leading to a poor performance in software.

As one can see we have a slight difference of the leading ciphers in both code size and throughput. Since we tried to get a good trade off between code size and throughput in our implementations we have to reduce this indifference to a common denominator. We now introduce an additional metric to give consideration to both code size and throughput. The ratio of both was computed to visualize the combined metric. This metric is given in Figure 6.4. Please notice the logarithmic scale on the

y-axis.

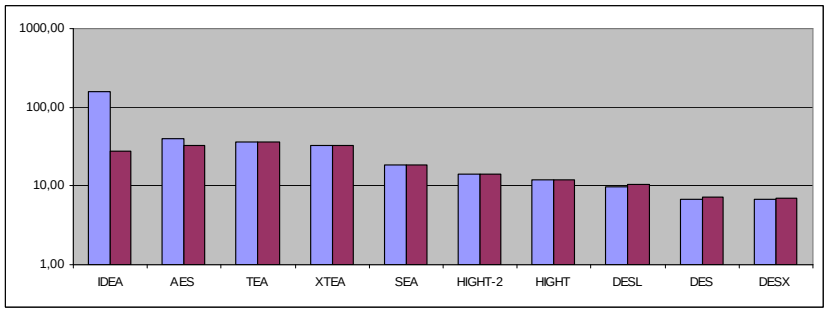


Figure 6.4: Throughput-Code size ratio of Encryption and Decryption

Again our highly optimized IDEA leads the field, AES together with the TEA family follow. IDEA only wins with its fast encryption. The TEA family is just slightly behind our reference group followed by SEA and the HIGHTs. The DES family brings up the rear, again the lightweight version DESL outperforms the other two members of the family. In our new metric the TEA family is at least able to outperform IDEA and AES in decryption. It can be seen that the ciphers designed for 8-bit software platforms, namely TEA/XTEA and SEA (and AES & IDEA, of course) outperform the hardware-oriented ciphers HIGHT and the DES family, as expected.

7 Conclusions and Further Work

We presented a performance analysis of newly proposed light-weight encryption algorithms on an 8-bit Smart Card microcontroller. The same microcontroller and setup can be found in many embedded devices and many applications of ubiquitous computing like wireless sensor networks. Our reference implementations of AES and IDEA won most of the rankings. Only in a few sections our implemented crypto algorithms designed for light-weight applications came off well. Some of them did very well in the code size or cycle count ranking: the implementation of HIGHT outperforms IDEA in both performance of encryption and decryption as well as in memory allocation. Though DESL is slightly smaller than AES in code size, it has a worse performance and does not provide comparable security. The drawback is that they still have to show that they can withstand cryptanalysis over a long time. Most of them are too young to be considered secure. By contrast older ciphers like DES or the TEA variants have been deeply analyzed in the meantime. Our analysis shows, that ciphers especially designed for light-weight applications cannot really keep up with standardized algorithms like the AES. Even if they are optmized for software implementations like the TEA family, at least they failed to outperform the AES in throughput.

We propose to optimize the light-weight cryptography algorithms XTEA, SEA, and HIGHT to reveal their full potential. Maybe bit-slicing could be applied to them as well. With our work we have shown that cryptography algorithms specially designed for light-weight purpose can only keep up with standardized and trusted algorithms in code size. The AES and IDEA outperformed nearly all of the ciphers in terms of throughput at least in encryption. One should consider well before using one of these ciphers if they first need to be highly optimized to beat the AES.

As an overall summary one can say that these light-weight block ciphers are not the best choice for using them on an 8-bit microcontroller. Only if memory is highly critical, some of the ciphers might be an alternative to standardized and widely spread algorithms.

A Source Codes

Listing A.1: S-Box Reorder Algorithm

```
int main(int argc, char* argv[])
{
        int i;
        int j;
    for(j=0;j<8;j++)
    {
    printf("\n");
    printf("sbox%i:\n.byte ",j+1);
    for(i=0;i<8;i++)
    {
        if(i == 7) printf("0x0%x, 0x0%x", S[j][i], S[j][i+16]);
        else printf("0x0%x, 0x0%x, ", S[j][i], S[j][i+16]);
    }
    printf("\n.byte ");
    for(i=8;i<16;i++)
        {
        if(i == 15) printf("0x0%x, 0x0%x", S[j][i], S[j][i+16]);
        else printf("0x0%x, 0x0%x, ", S[j][i], S[j][i+16]);
    }
    printf("\n.byte ");
    for(i=32;i<40;i++)
        {
        if(i == 39) printf("0x0%x, 0x0%x", S[j][i], S[j][i+16]);
        else printf("0x0%x, 0x0%x, ", S[j][i], S[j][i+16]);
    }
    printf("\n.byte ");
    for(i=40;i<48;i++)
        {
        if(i == 47) printf("0x0%x, 0x0%x", S[j][i], S[j][i+16]);
        else printf("0x0%x, 0x0%x, ", S[j][i], S[j][i+16]);
    }
    printf("\n");
}
return 0;
}
```

B Bibliography

[1] Atmel Corporation. Avr studio 4.12, build 498. Available from: `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725`.

[2] Atmel Corporation. *8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash*, revision 1142e-avr-02/03 edition, 2003. Available from: `http://www.atmel.com/`.

[3] D. J. Bernstein. Cache-timing attacks on aes. published on website, 2005. Available from: `http://cr.yp.to/papers.html`.

[4] E. Biham. A fast new des implementation in software. In *Fast Software Encryption, 4th International Workshop*. Springer Verlag, 1997.

[5] E. Biham and A. Biryukov. An improvement of davies' attack on des. In *Proceedings of EUROCRYPT '94*, pages 461–467. EUROCRYPT '94, 1994.

[6] D. Culler et al. Tiny os - a component-based os for the networked sensor regime. 2003. Available from: `http://webs.cs.berkeley.edu/tos`.

[7] J. Daemen and V. Rijmen. *The design of Rijndael, the Advanced Encryption Standard*. Springer-Verlag, 2003.

[8] Duolabs. Cas interface studio, version 7.9b. Available from: `http://www.duolabs.com/`.

[9] Efton Homepage. Implementing sea on x51 and avr. Available from: `http://www.efton.sk/crypt/sea.htm`.

[10] Electronic Frontier Foundation. *Cracking DES*. O'Reilly & Associates, 1998.

[11] Federal Information Processing Standards Publication 46-3. Data encryption standard (des). Technical report, FIPS, 1999.

[12] K. Finkenzeller and R. Waddington. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification*. John Wiley and Sons, 2003.

[13] Brian Gladman. Byte oriented aes implementation. Available from: `http://fp.gladman.plus.com/AES/`.

[14] J.C. Hernandez and P. Isasi. Finding efficient distinguishers for cryptographic mappings, with an application to the block cipher tea. In *Proceedings of the 2003 Congress on Evolutionary Computation*, 2003.

[15] J.C. Hernández, J.M. Sierra, A. Ribagorda, B. Ramos, and JC Mex-Perera. Distinguishing tea from a random permutation: Reduced round versions of tea do not have the sac or do not generate random numbers. In *Proceedings of the IMA Int. Conf. on Cryptography and Coding 2001*, pages 374–377, 2001.

[16] D. Hong et al. Hight: A new block cipher suitable for low-resource device. In *Proceedings of CHES 2006*, 2006.

[17] S. Hong, D. Hong, Y. Ko, D. Chang, W. Lee, and S. Lee. Differential cryptanalysis of tea and xtea. In *Proceedings of ICISC 2003*, 2003.

[18] J. Kelsey et al. Key-schedule cryptanalysis of idea, g-des, gost, safer, and triple-des. In *Advances in Cryptology*, pages 233–246, 1996.

[19] J. Kelsey et al. Related-key cryptanalysis of 3-way, biham-des, cast, des-x new des, rc2, and tea. In *First International Conference on Information and Communication Security*, pages 233–246, 1997.

[20] J. Kilian and P. Rogaway. How to protect des against exhaustive key search (an analysis of desx). *Journal of Cryptology*, Volume 14:17–35, 2001.

[21] Y. Ko et al. Related key differential attacks on 26 rounds of xtea and full rounds of gost. In *Proceedings of FSE '04*. Springer-Verlag, Berlin, Germany/Heidelberg, Germany/London, UK, 2004.

[22] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with copacobana - a cost-optimized parallel code breaker. In *Conference on Special-purpose Hardware for Attacking Cryptographic Systems*. Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany Institute of Computer Science and Applied Mathematics, Faculty of Engineering, Christian-Albrechts-University of Kiel, Germany, 2006.

[23] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[24] Microsoft Corporation. Winscard-api. Available from: `http://msdn2.microsoft.com/en-us/library/aa379479.aspx`.

[25] D. Moon et al. Impossible differential cryptanalysis of reduced round xtea and tea. In *Fast Software Encryption: 9th International Workshop*, page 49, 2002.

[26] R.M. Needham and D.J. Wheeler. Tea extensions. Computer Laboratory, Cambridge, 1997.

[27] D.A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Proceedings of RSA Conference*, 2006.

[28] A. Poschmann, G. Leander, K. Schramm, and C. Paar. New light-weight des variants suited for rfid applications. In *Proceedings of FSE 2007*. FSE 2007, 2007.

[29] W. Rankl and W. Effing. *Handbuch der Chipkarten*, volume 2. Carl Hanser Verlag München Wien, 1996.

[30] Matthew D. Russell. Tinyness: An overview of tea and related ciphers. Draft v0.3, February 2004.

[31] M.-J. Saarinen. Cryptanalysis of block tea. Unpublished manuscript, October 1998. Available from: `http://www.cc.jyu.fi/~mjos/block_tea.ps`.

[32] F.X. Standaert, G. Piret, N. Gershenfeld, and J.J. Quisquater. Sea: A scalable encryption algorithm for small embedded applications. Workshop on RFIP and Lightweight Crypto, Graz, Austria, 2005.

[33] Simon Steele. Programmer's notepad 2, version v2.0.6.1-ella. Available from: `http://www.pnotepad.org/`.

[34] Sun Microsystems, Inc. *Java Card Technical Articles*, 2007. Available from: `http://developers.sun.com/techtopics/mobility/javacard/articles/`.

[35] D. Wheeler and R. Needham. Tea, a tiny encryption algorithm. In *Lecture Notes in Computer Science*, 1994.