7-17-2003

Embedded Cryptography: An Analysis and Evaluation of Performance and Code Optimization Techniques for Encryption and Decryption in Embedded Systems

Jayavardhan R. Kandi
University of South Florida

Embedded Cryptography:

An Analysis and Evaluation of Performance and Code Optimization Techniques for Encryption and Decryption in Embedded Systems

by

Jayavardhan R. Kandi

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering
Department of Electrical Engineering
College of Engineering
University of South Florida

Major Professor: Dr. Wilfrido Moreno, Ph.D.
Dr. James Leffew, Ph.D.
Dr. Kenneth Buckle, Ph.D.

Date of Approval:
July 17th, 2003

Keywords: AES, Rijndael, DSP, Co-synthesis, StarCore

DEDICATION

To

Dr. Wilfrido Moreno

ACKNOWLEDGMENTS

It gives me a great pleasure in acknowledging the persons who have helped me in this endeavor. I would like to thank and express my sincere gratitude to my Major Professor and advisor Dr. Wilfrido Moreno for all his support and the freedom he provided me in my Masters career. I would also like to thank Dr. James Leffew and Dr. Kenneth Buckle for their consent to be my committee members.

I thank my colleagues Mr. Eduardo Zurek, Mr. Luis Navarrete and Mr. Jorge Galvis for teaching me a lot about how things go about in this world. I thank my friends Mr. Lolla, Mr. Barri and Mr. White Murthy for their moral support. I also thank my roommates and the whole bunch of the Miguel Ct. Volley Ball team for their everlasting trust and belief in my work. I thank my family, for whom my life and all my achievements are dedicated.

My final thanks to the inspiration I have been receiving from beyond this physical realm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE SAMPLES

EMBEDDED CRYPTOGRAPHY:

AN ANALYSIS AND EVALUATION OF PERFORMANCE AND CODE OPTIMIZATION TECHNIQUES FOR ENCRYPTION AND DECRYPTION IN EMBEDDED SYSTEMS

Jayavardhan R Kandi

ABSTRACT

It is clear that Cryptography is computationally intensive. It is also known that embedded systems have slow clock rates and less memory. The idea for this thesis was to study the possibilities for analysis of cryptography on embedded systems. The basic approach was the implementation of cryptographic algorithms on high-end, state-of-the-art, DSP chips in order to study the various parameters that optimize the performance of the chip while keeping the overhead of encryption and decryption to a minimum.

Embedded systems are very resource sensitive. An embedded system is composed of different components, which are implemented in both hardware and software. Therefore, hardware-software co-synthesis is a crucial factor affecting the performance of embedded systems. Encryption algorithms are generally classified as data-dominated systems rather than ubiquitous control-dominated systems. Data-dominated systems have a high degree of parallelism. Embedded systems populate the

new generation gadgets such as cell phones and Smartcards where the encryption algorithms are obviously an integral part of the system. Due to the proliferation of embedded systems in all the current areas, there is a need for the systematic study of encryption techniques from the embedded systems point of view.

This thesis explored the different ways encryption algorithms can be made to run faster with much less memory. Some of the issues investigated were overlapped scheduling techniques for high-level synthesis, structural partitioning, real-time issues, reusability and functionality, random number and unique key generators, seamless integration of cryptographic code with other applications and architecture specific optimization techniques.

CHAPTER 1

INTRODUCTION

Ever since man developed his communication skills, he has embarked on a journey of technological developments. These communication skills have been developed to such an extent that the information passed must, at times, be secret and authenticable. The new conditions of secrecy, authenticity and integrity have given rise to a new field of science called cryptology. Cryptology is divided into cryptography and cryptanalysis. Cryptography, deals with the art and science of encoding and decoding information, whereas, cryptanalysis deals with breaking the encoded information.

As the human race advanced, it developed machines to perform strenuous physical tasks and computers to perform logical tasks. Presently, technology has advanced to a level where computers have invaded all spheres of science and technology. In the future computing power will increase and become more pervasive through transformations in the form of embedded systems. An embedded system can be defined as a computing system assigned to a specific task, which is embedded in a larger multifarious system. A typical example of an embedded system is a router in a local area network.

As the need for secure data transmission grows, there is a major urgency of integrating cryptography into the embedded systems, in order to enable secure and reliable data transfer. This research explored the different factors that would enable a propitious insertion of the cryptography into the embedded systems.

Embedded systems are comprised of microprocessors, microcontrollers, DSPs and FPGAs. The software that runs on these hardware devices must be both concise and precise. The cryptographic modules that help to encode and decode the data must be designed and implemented in a transparent manner in order not to consume too much of the memory and processing resources.

The basic structure of this research was to incorporate the advanced encryption

standard (AES) algorithm onto the network DSP, StarCore, of Motorola. The hardware-software co-design formed the major breakthrough of this research. The AES algorithm was divided into different sub-modules, which could be run in parallel and scheduled in such a manner that processing resources were consumed only when needed. The objective of this research was to run the AES algorithm fast enough to enable the embedded system to work in a real-time environment without compromising either the secure transfer of the data or incurring any data loss.

The current technology uses specialized ASICs running DES and 3-DES algorithms for encryption. With the advent of AES, as successor to DES, and the prospects of embedded systems, this research holds a prominent position in the evaluation and analysis of the structure of the Rijndael algorithm (AES) from the embedded systems point of view. Implementing a mere algorithm on hardware doesn't ensure that the system is secure. Most of the cryptographic systems on the market are not as secure as they claim. This is due to the lack of importance given to the cryptography since the programmers deal with it as just another component of the program. A system cannot be made absolutely secure unless the cryptographic issues are kept in mind from the conception to completion. Cryptographic systems are very much different than other products. There is no outward difference between a strong cryptographic and a weak cryptographic system. Even though both may use the same algorithm and the same hardware, the secure system needs to consider all aspects of attacks and the means to prevent them. After all, a cryptographic system is only as strong as its weakest point. What makes implementing a cryptographic system challenging is that attackers do not follow any rules. Attackers try to breach the security protocols and tamper with the system in new ways that the designer might not even have thought about.

Many algorithms appear to be very strong from the mathematical point of view. The most often neglected part is the implementation of these algorithms in a successful manner. The first step for a secure system is to define the threat model. The threat model should comprehensively consider how secure the data should be and what are the motivations of the attackers. Consideration of how to detect an attack and prevent system crashes is crucial. The threat model differs for different applications and roles. A good cryptographer is one who is adept in areas such as number theory, complexity

theory, information theory, probability theory and abstract algebra. Implementing a good cryptographic system entails far more than just understanding the algorithm. A simple flaw like a poor random number generator or not discarding the key after its use can render the system useless. Therefore, it was a major effort of this thesis to study the aspects, which the programmers have to deal with when attempting to implement a better cryptographic system.

CHAPTER 2

EMBEDDED SYSTEMS

An embedded device is a computing system that is part of a bigger system. The major difference between embedded devices and computers is that, unlike computers, embedded devices are designed and developed for fast and efficient execution of the assigned specific task. Generally, a single embedded device is assigned a fixed specific task for its lifetime.

An embedded device must be quick enough to respond to the high priority events. In order to do so care must be taken to reduce the functional overhead as much as possible. The functions of the embedded device must be kept in mind from the hardware design inception until the end of software execution [SJBW96]. There is a great deal of hardware-software interaction involved in the sharing and execution of the algorithms to be run over the embedded devices. Therefore, programming for an embedded device is different from programming a conventional computer.

The application and the controllers of the embedded systems are integrated both into the hardware and software. Therefore, the embedded systems programmer must have a thorough knowledge of both the advantages and limitations of the hardware architecture. An embedded system's efficiency, [RL00], is invariably related to the extent of the code. The efficiency of a program increases, as the code size decreases and the execution speed increases. Therefore, implementation of programs with very tight memory constraints is a challenge and a requirement for every embedded systems programmer.

2.1 Hardware/Software Co-design

Some functions are better suited to run on hardware, and some others are suited to run on software. Co-design considers both the abilities of the hardware and the flexibility of the software so as to optimize the performance of the system. Some functions are implemented on both hardware and software. Such seamless integration forms the core of the type of programming required for embedded systems.

2.2 Structural Partitioning

Interfaces with the outside world are as important as the internal structure. Special attention must be given to the interfaces and in almost all cases; they are kept outside the internal processing. This kind of structural partitioning helps in uninterrupted execution of assigned tasks, irrespective of possible overload in other structures.

2.3 Functional Partitioning

Some applications are better described by functionality rather than structure. Such functional partitioning yields simpler hardware design and results in time multiplexing of the signals. Digital Signal Processors are better suited for this type of behavioral synthesis.

2.4 Classification

Embedded systems are mainly classified as follows:

- Microcontroller Design: These systems are principally used for control-dominated systems. They have a rich set of instructions for efficient bit-level data manipulation. They usually have Complex Instruction Set Computer, (CISC), architectures.
- RISC Architecture Systems: These systems have Reduced Instruction Set Computer, (RISC), architectures and are suited for fast execution. They usually have a large number of registers in order to speed up instruction execution. The instruction set is composed of a deliberately chosen set of instructions capable of executing multiple tasks. In other words, a single RISC instruction is equivalent to multiple CISC instructions.

- Digital Signal Processors: These kinds of systems are used for arithmetic-intensive systems such as speech analysis, encryption and image processing. The DSP architecture supports hardware multiplication, address generation units and separate data and address buses.
- Field Programmable Gate Arrays: These kind of systems often run different functions in parallel to maximize system performance. These systems are reconfigurable in nature, which means that their functionality can be upgraded or changed altogether when needed. This provides them with fault-tolerance ability since they can be reconfigured to remove a hardware or software fault.
- Application Specific Processors: These kinds of systems are composed of specially designed integrated circuits called Application Specific Integrated Circuits or ASICs. ASICs are often specialized enough that they are used as common of the shelf components. These systems do not have the capability of being upgraded and are mostly hardware oriented.

2.5 Programming Languages

Programming of an embedded system is a very important task. Although assembly-level programming gives the optimum level of performance, high-level programming is still needed to design the program structure at higher abstract levels. Some of the programming languages that have gained importance in the design of embedded systems are as follows:

- C: The C language is a well-established and proven language in the programming community. The main advantages of the C language are that it is very easy to learn and offers an almost assembly like code to the processors. The strong point of the C language is direct memory access through pointers.
- C++: C++ evolved from C. When applied to the programming of embedded systems, C++ generates a considerable amount of overhead that is detrimental for embedded programming. In order to increase the run-time efficiency and reduce the code size, a new standard called EC++, which stands for Embedded C++, is being developed.

- Java: Java is an object-oriented programming language that was designed for Internet applications. However, it has become quite popular due its flexibility and some enthusiasts are applying Java to embedded systems programming. A modified version called embedded java is gaining popularity due to such features as portability and software reuse.

CHAPTER 3

CRYPTOGRAPHY

Cryptography is the science and art of encoding and decoding data in order to attribute the properties of secrecy to the data. The data to be encoded is termed as plain text. The encoded data is known as cipher text. Thus, the process of encoding and decoding can also be termed encryption and decryption respectively. The system for encrypting or decrypting is called a cryptosystem and the persons who design such systems are classified as cryptographers. The process of encryption involves an algorithm for combining the plain text with a key resulting in the cipher. A key is a selected number or string of characters that should be known only to the sender and the recipient.

Cryptanalysis deals with the techniques of breaking the codes in order to extract the plaintext from the cipher without the consent of the sender or recipient. The persons who are adept at cryptanalysis are called as cryptanalysts. Both cryptography and cryptanalysis fall under the broad science of cryptology.

The essence of using cryptography in this world is to validate the following three conditions:

- Secrecy: The data transmitted must be secret and any eavesdropper should not be able to understand it.
- Authenticity: The recipient must be guaranteed that the transmitted data is from an authentic sender and is not from any other person.
- Integrity: The transmitted data must be tamper resistant and any eavesdropper should not be able to meddle with the cipher.

The security of a cryptographic algorithm should be based on the key and not on the secrecy of the algorithm. This means that the algorithm should be made public and allowed for scrutiny by the intellectual community. The types of attacks it can withstand

are the only gauges of the strength of an algorithm. An ideal algorithm should be so strong that it can only be cracked by knowing the key. This type of attack, which exploits the different combinations of keys, is known as a bruteforce attack. The classification of cryptographic algorithms is based on the keys and is discussed next.

3.1 Symmetric Algorithms

Symmetric algorithms, are those where the encryption key and the decryption key are the same or are calculated from each other. If the algorithm uses a single key for both encryption and decryption, then it is called as a single-key algorithm.

Symmetric algorithms can be divided into two types:

- Block algorithms: These algorithms operate on a block of data each time in order to convert it into a block cipher. A typical block size would be 64 bytes.
- Stream algorithms: These algorithms operate on a stream of data at a single bit or a single byte at a time.

3.2 Asymmetric Algorithms

Asymmetric algorithms are those that use different keys for encryption and decryption and one key cannot be computed from the other key. They are also called public-key algorithms due to the fact that the encryption key can be made public. The recipient has the private key with which only she can decrypt the cipher.

3.3 Building Blocks of an Algorithm

The basic building blocks of an algorithm consist of activities such as:

- Substitution Cipher: This is a cipher where each character of the plain text is substituted for a preconceived cipher character. Substitution falls under the confusion scheme with the purpose of frustrating the eavesdropper.
- Transposition Cipher: This is a cipher where the order of the characters of a fixed block plain text is shuffled in a predetermined fashion. Transposition falls under the diffusion scheme, which removes the redundancies in the cipher.
- Exclusive-OR Cipher: This is a cipher formed by a simple bit-wise XOR operation on the plain text and the keyword.

3.4 Key Length

A bruteforce attack should be made as complex as possible. This is achieved by increasing the length of the key. If the key size is 60 bits, then there are 2^{64} possible keys, which would require considerable processing time in order to test all the keys. If the key size is increased to 128 bits, then the number of possible keys is 2^{128} and this huge figure makes it very difficult to find the right key. As technology grows, computing power also increases. Therefore, care must be taken to consider the technological growth and the key-size should be aptly decided so that the algorithm will remain strong into the future.

3.5 Algorithm Modes

Even though a basic algorithm is designed to be efficient, different cipher modes are implemented in order to make the algorithm efficient in concealing the patterns. The different types of cipher modes are as follows:

3.5.1 Electronic Code Book

Electronic Code Book is a straightforward method of converting a block of plaintext into cipher text. The advantage of this mode is asynchronous enciphering of the data. However, this mode is vulnerable to attacks since the same plain text is always converted to the same cipher text. Figure 3.5.1 presents the block diagram for the Electronic Code Book (ECB) mode.

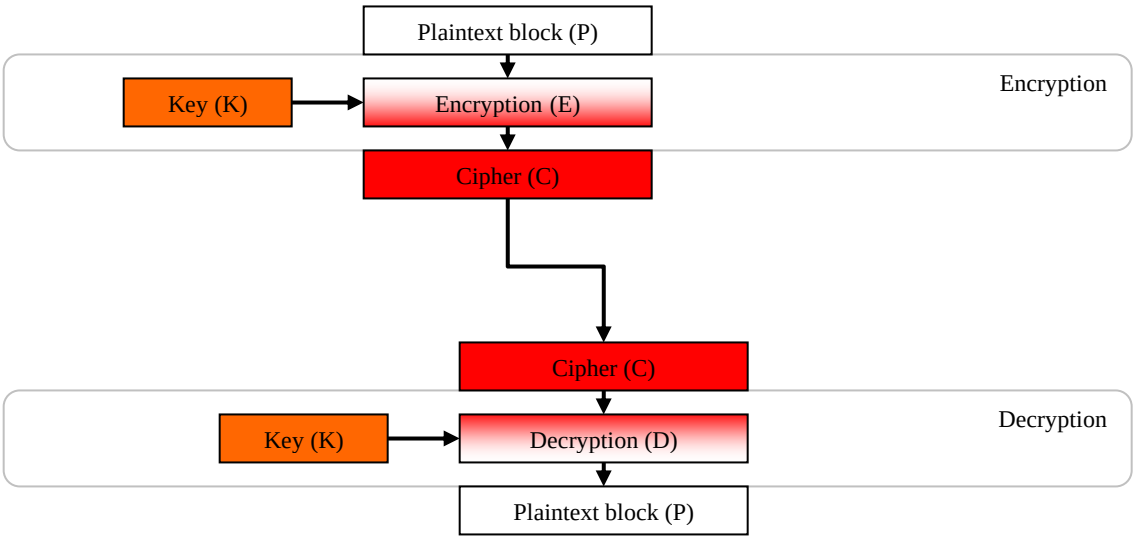

Figure 3.5.1: Block cipher encryption in electronic code book (ECB) mode

3.5.2 Cipher Block Chaining

In Cipher Block Chaining mode, the plain text is XORed with the previous cipher text block before encryption. Thus, the encryption of each block depends on all the previous blocks.

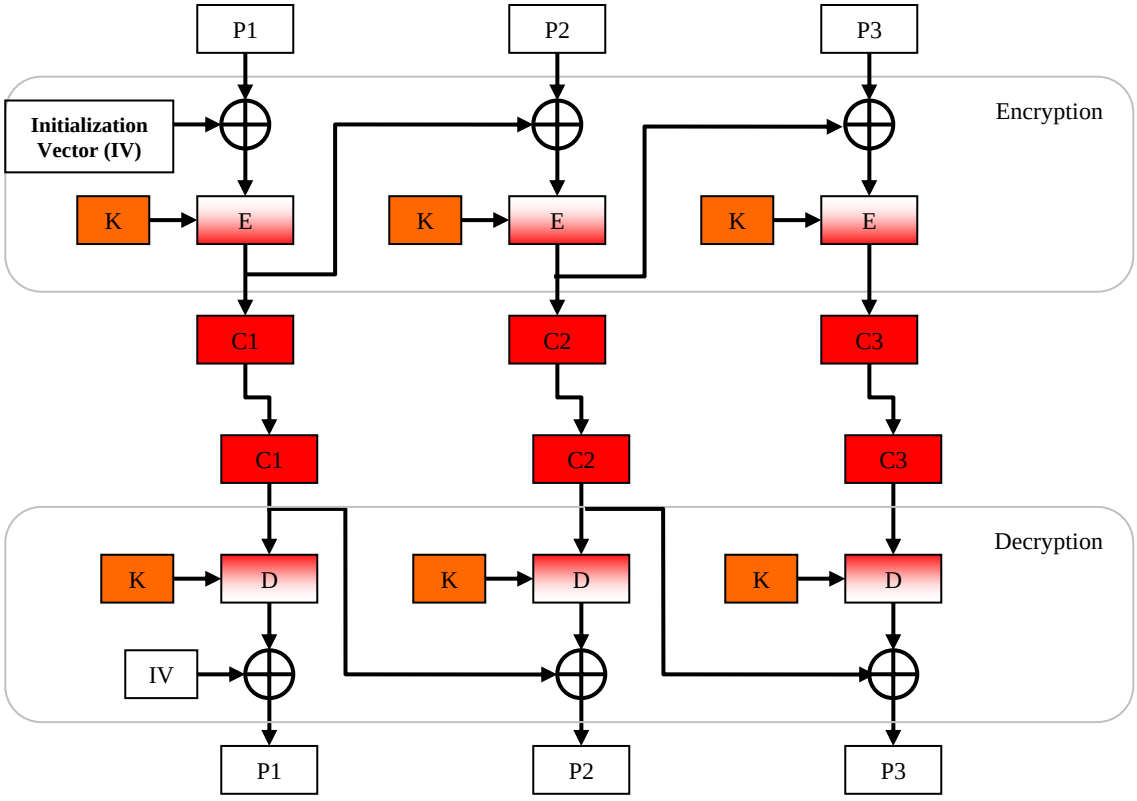

Figure 3.5.2: Block cipher encryption in cipher block chaining (CBC) mode

Figure 3.5.2 presents the block diagram for the Cipher Block Chaining (CBC) mode. This mode removes the patterns when compared to the ECB mode. If a bit error occurs during encryption, it will affect all the subsequent blocks. However, during decryption, the effect is reversed and the recovered plain text will only have a single error.

3.5.3 Cipher Feedback Mode

Cipher Feedback Mode is used when data must be transmitted in blocks smaller than as a full block. The incoming byte, or a group of bytes, of plaintext is XORed with the LSB of the self-synchronizing stream cipher. A self-synchronizing stream cipher has

a key stream in which every bit is a function of a fixed number of previous cipher texts. Figure 3.5.3 presents the block diagram for the Cipher Feedback (CFB) mode.

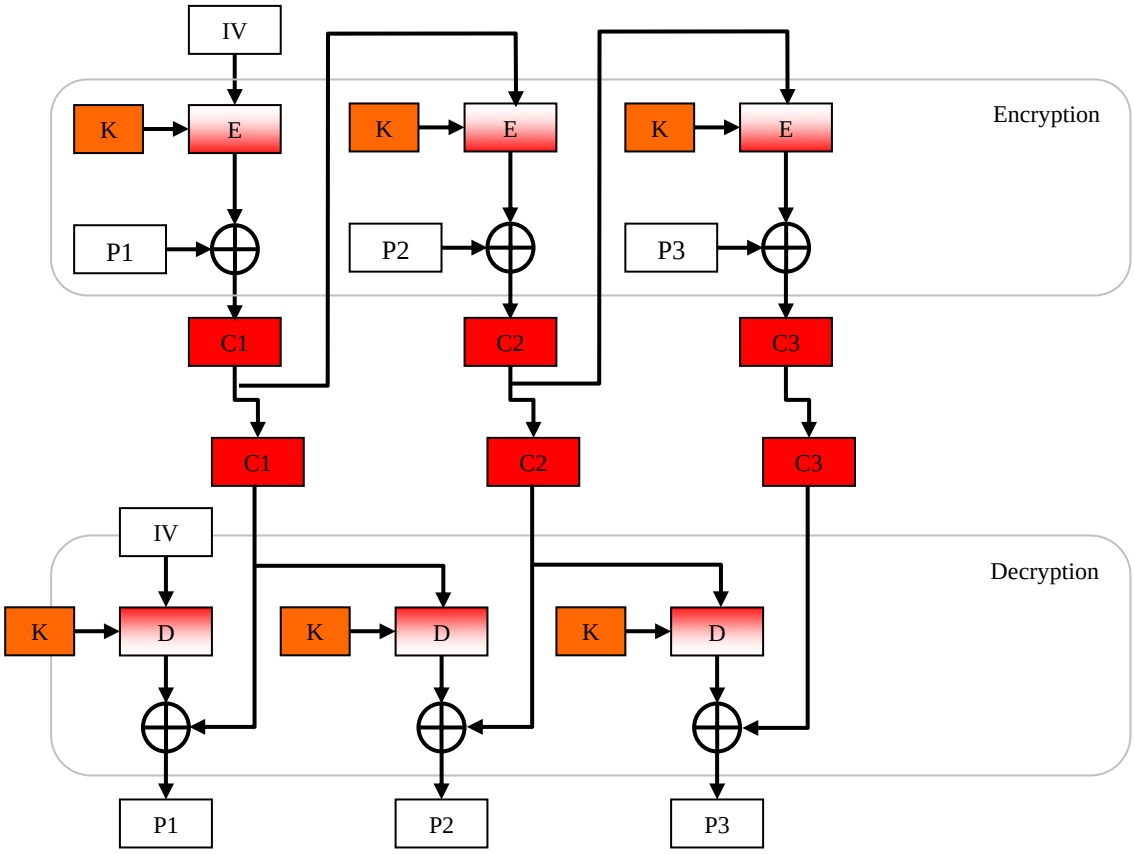

Figure 3.5.3: Stream cipher encryption in cipher feedback (CFB) mode

3.5.4 Output Feedback Mode

The Output Feedback Mode (OFB) also uses a synchronous stream cipher but the feedback mechanism is independent of the plain text and the cipher text stream. Instead of inserting the cipher text bits as feedback, the *n* bits of the output block are moved to the right side of the shift register. Therefore, this mode provides ease of processing even before the plain text arrives. When the plain text arrives, it is simply XORed with the output bits of the algorithm in order to form the cipher text. Figure 3.5.4 presents the block diagram for the Output Feedback mode.

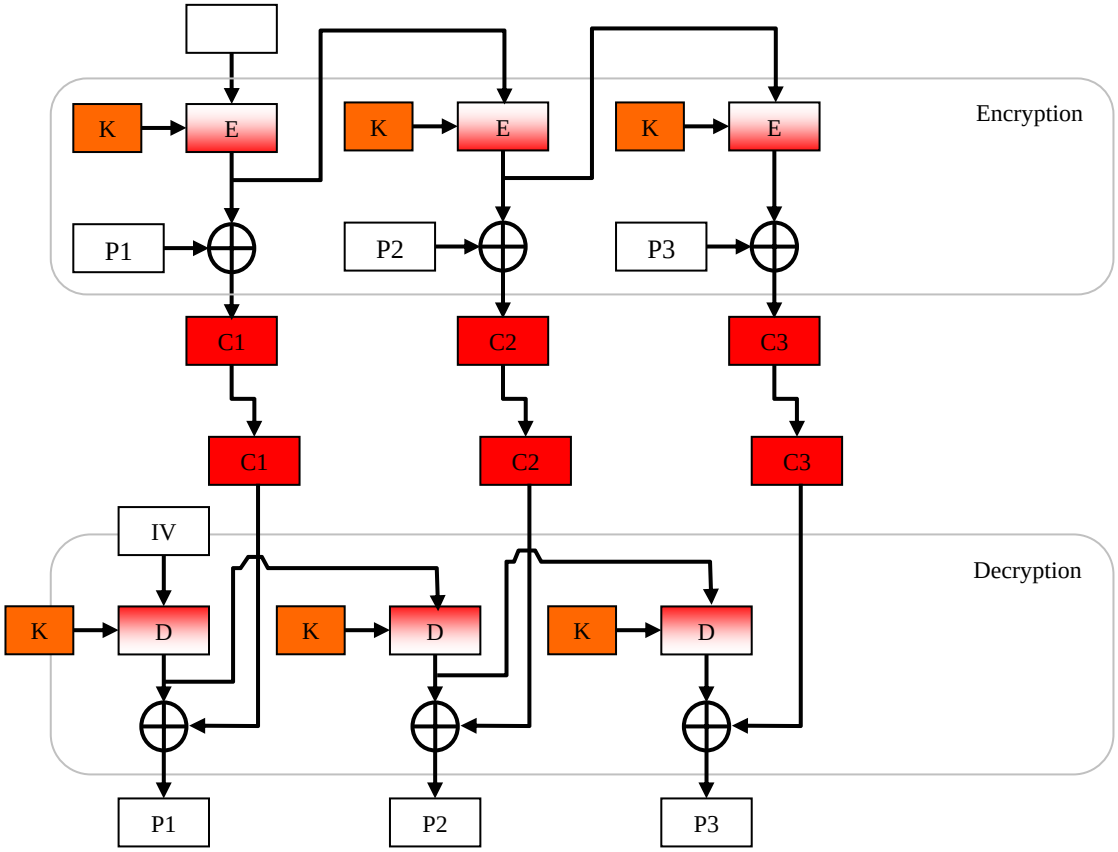

Figure 3.5.4: Stream cipher encryption in output feedback (OFB) mode

3.5.5 Counter Mode

All the above modes were used in a pre-AES time. Except for the ECB mode, they all involve feedback, which is comparatively insecure and performance delay are observed. Therefore, a new type of mode was proposed, which is termed the counter mode. In this mode, a counter is encrypted to generate a key stream, which is simply XORed with the plain text in order to generate the cipher text. The advantage of counter mode is that there is no feedback or chaining. Figure 3.5.5 presents the block diagram for the Counter (CTR) mode.

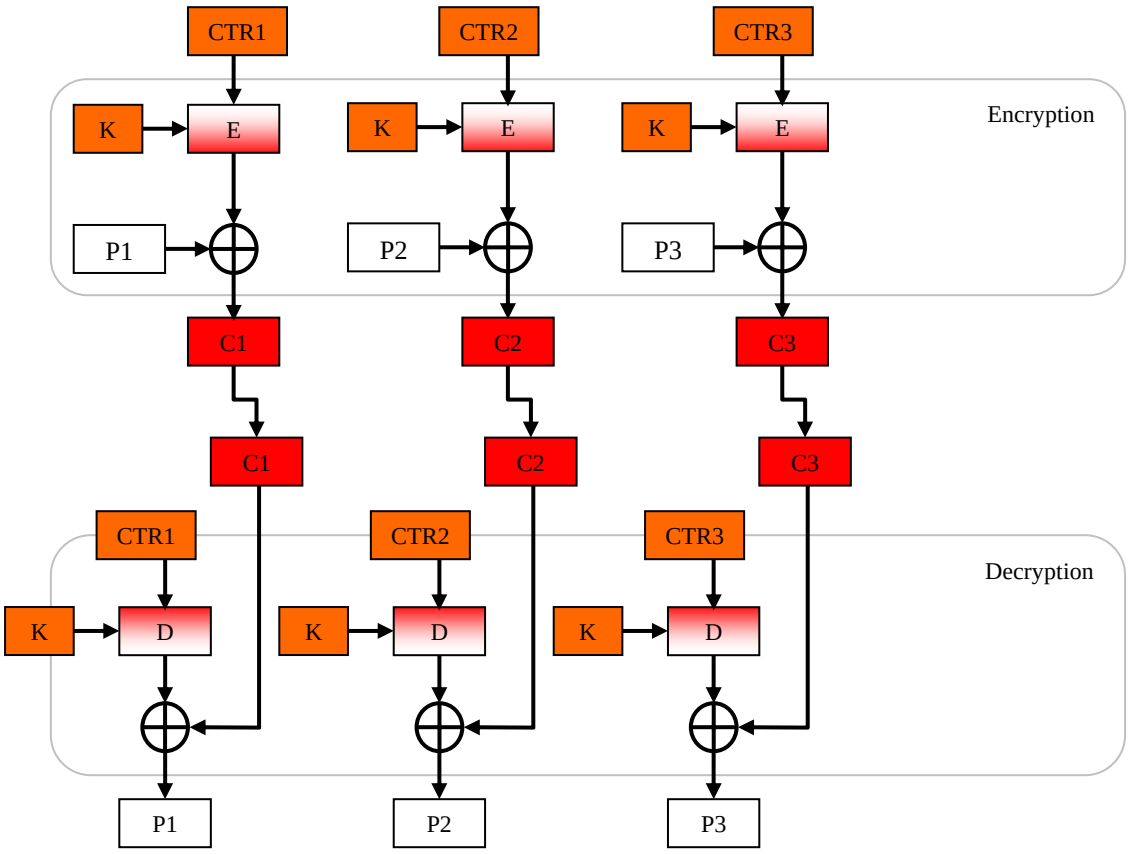

Figure 3.5.5: Stream cipher encryption in counter (CTR) mode

3.6 Selection of Algorithm

The strength of the algorithm is as important as the length of the key. Therefore, selection of an algorithm to design a cryptosystem forms the first and most formidable task. There are many aspects that require consideration in choosing a particular algorithm, [BS96], such as:

- Relying on a published algorithm and hoping that the published algorithm is open to public scrutiny and hasn't been broken
- Relying on some commercial product
- Relying on the algorithms proposed as the standards
- Writing a new algorithm

The only way an algorithm can be considered reliable is when it has been scrutinized thoroughly and the intellectual community finds no known attacks.

3.7 Hardware/Software Co-synthesis

3.7.1 Hardware Encryption

Until recently, encryption was performed through specialized hardware chips. The inherent advantage of using hardware for encryption is speed. These hardware devices were connected at the data transfer links in order to enable encryption and decryption. Hardware is also preferred in order to make the encryption system tamper-proof.

3.7.2 Software Encryption

Software encryption is currently being widely used due to the software features of portability and flexibility. However, software encryption is very slow and is insecure in many aspects of key management and program manipulation.

3.7.3 Hardware/Software Encryption

A new blend of hardware and software is currently being used for encryption in order to combine the best features of both. This provides a considerable speed advantage and security. Additionally, it provides for ease of programming. Future devices will contain encryption modules along with other applications.

3.8 Advanced Encryption Standard

The National Institute of Standards and Technology has selected the Rijndael algorithm, [DR01], as its current encryption standard. Henceforth this algorithm will be called the Advanced Encryption Standard, (AES). The AES is supposed to be the current encryption standard, which is deemed stronger than the old DES and triple-DES standards. The Rijndael algorithm, which was invented by Vincent Rijmen and Joan Daemen, consists of data block and key lengths of 128, 160, 192, 224 or 256 bits. However, for the AES a fixed data block length of 128 bits was standardized with a variation in key sizes of 128, 192 and 256 bits. Thus, the corresponding names are AES-128, AES-192 and AES-256. A detailed analysis of the AES is provided in chapter 4.

CHAPTER 4

SYSTEM DESIGN

4.1 StarCore-Hardware Overview

The Motorola MSC8101 is a 16-bit digital signal processor. This is based on the StarCore™ SC140 DSP core and is a fully static low-power CMOS device that operates from 0 to 300MHz. Figure 4.1.1 presents the block diagram of the MSC8101 processor.

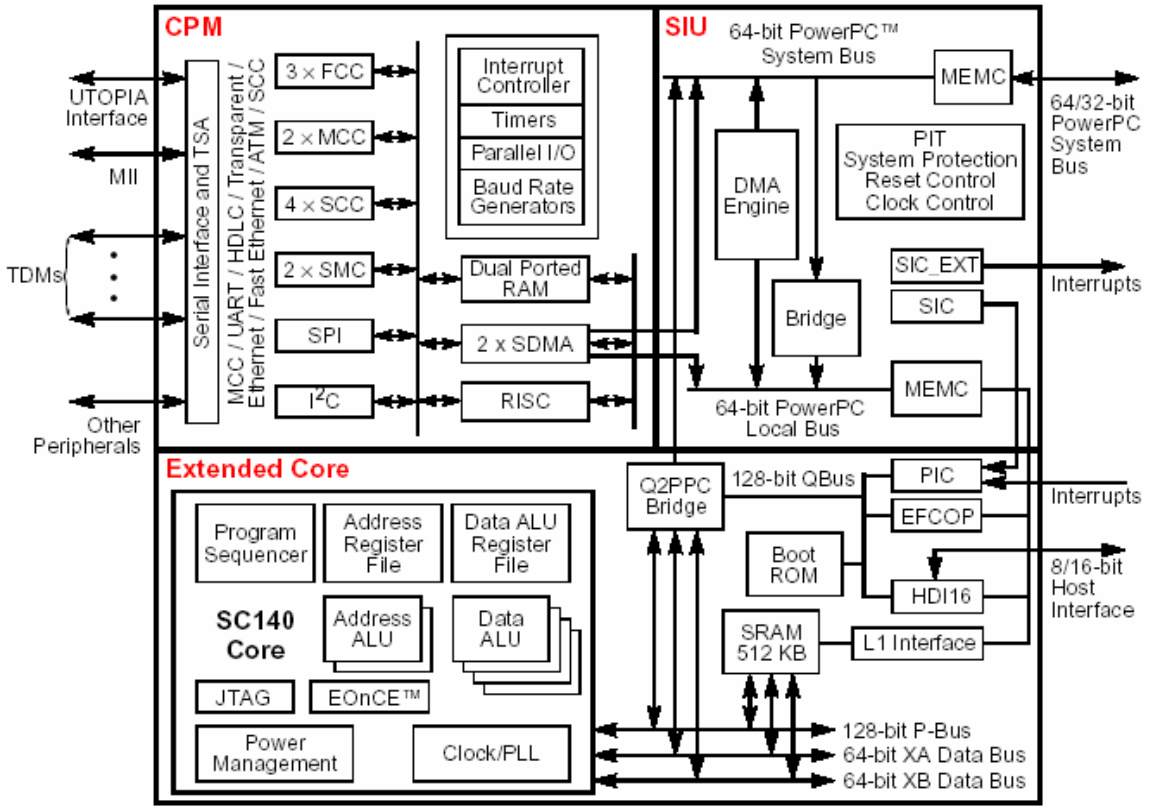

Figure 4.1.1: Block diagram of MSC8101 – courtesy of Motorola Inc.

4.1.1 SC140 Core

The SC140 core consists of the Data Arithmetic Logic Unit (Data ALU), the Address Generation Unit (AGU) and the Program Sequencer (PSEQ).

- The Data ALU performs the core's arithmetic and logical operations on the data. It has sixteen 40-bit registers and four ALUs that execute in parallel. This provides the flexibility of executing four Multiply Accumulate (MAC) instructions in a single clock cycle.

  Each ALU consists of a MAC sub-unit and a bit field sub-unit (BFU).

  - The MAC unit contains a high-speed adder and a multiplier that implement integer and fractional arithmetic instructions.
  - The BFU handles the logical operations for the ALU.
- The AGU consists of two address arithmetic units (AAUs), two stack pointers, a bit mask unit (BMU) and sixteen 32-bit address registers.
  - An AAU calculates the effective address for memory access.
  - The AGU has two stack pointers. One pointer for normal mode execution (NSP) and one pointer for exception mode processing (ESP).
  - The BMU performs the setting and resetting of the bits in any destination register.
- The Program Sequencer fetches and executes the instructions. The PSEQ has a Program Counter, (PC), which has four pairs of 32-bit loop start address registers and four hardware-based loop counters.

The StarCore has an on-chip memory bank of 512 KB, which helps in running longer programs without a need for external memory.

The SC140 has two extended components:

- The enhanced filter coprocessor (EFCOP) implements a real/complex adaptive filter machine in parallel with the SC140 core.
- The HDI16 provides a 16-bit parallel interface that allows the device to interconnect with other microcontrollers, microprocessors and DSPs.

4.1.2 System Interface Unit

The system interface unit, (SIU), provides the control and data signals necessary for the processor to interact with other peripherals.

4.1.3 Communications Processor Module

The communications processor module, (CPM), is a 32-bit RISC processor that controls and manages the external interfaces for the device. The CPM controls the following modules:

- 155 Mbps ATM interface (including AAL 0/1/2/5)
- 10/100 Mbit Ethernet interface
- Up to four E1/T1 interfaces or one E3/T3 interface and one E1/T1 interface
- HDLC support up to T3 rates, or 256 channels

4.1.4 Buses

The buses of the SC140 perform the following functions:

- The SC140 uses two different buses to access memory and data from the cache.
- SC140 has one 128-bit Program bus and two 64-bit data buses.
- An internal 64-bit PowerPC local bus moves data among the CPM, the DMA engine and the on-chip cache.
- The 64-bit PowerPC system bus manages data transfers among external memory/peripherals, the DMA engine and the SC140 core.
- A 128-bit QBus manages communications between the SC140 core and the extended core devices, EFCOP and HDI16. The QBus is also the interface between the processor core and the PowerPC system bus.

4.2 Advanced Encryption Standard

The Advanced Encryption Standard,, or, (AES,), operates on 128-bit data with variable key lengths of 128, 192 and 256 bits. The input plain text of 128-bits is arranged in a rectangular array of bytes that is called a `state`. A `state` has four rows and the number of columns is denoted by *Nb*, which is equal to the block length divided by 32 [G99]. Let the plaintext block be denoted by

$$p_0 p_1 p_2 p_3 \ldots p_{4.Nb-1}.$$

Where p_0 denotes the first byte and $p_{4.Nb-1}$ denotes the last byte of the plaintext. Figure 4.2.1 presents the arrangement of the input bits in a two-dimensional array form.

p_0	p_4	p_8	p_{12}
p_1	p_5	p_9	p_{13}
p_2	p_6	p_{10}	p_{14}
p_3	p_7	p_{11}	p_{15}

Figure 4.2.1: Input data layout in a 2-D array

Similarly, the key is arranged into a rectangular array of bytes in four rows and a subsequent number of columns. The number of columns varies as the length of the key varies.

Table 4.2.1: Different key lengths and corresponding number of rounds

	Block size (bits)	Key Length (bits)	Number of Rounds
AES-128	128	128	10
AES-192	128	192	12
AES-256	128	256	14

4.2.1 Round Transformations

The whole AES algorithm is divided into a fixed number of round transformations, which depends on the key length. Table 4.2.1 shows the number of round transformations for different key lengths. Each round transformation consists of four different transformations. The final round is a bit different from the rest of the rounds. Code Sample 4.2.1 presents the pseudo code for an AES encrypt round transformation.

4.2.2 Key Expansion

The `KeyExpansion()` function generates a key schedule for different rounds from the cipher key. The Key Expansion generates a total of *Nb(Nr + 1)* words: the algorithm requires an initial set of *Nb* words, and each of the *Nr* rounds requires *Nb* words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted by *W[i]*, with *i* in the range $0 \leq i < Nb(Nr + 1)$. Code Sample 4.2.2 shows the code for `KeyExpansion()`.

```
Round(state, roundkey)
      {
            SubBytes (state);
            ShiftRows(state);
            MixColumns(state);
            AddRoundKey(state,roundkey);
      }
/* Final round is special; there is no MixColumns  */
FinalRound(state, roundkey)
      {
            SubBytes (state);
            ShiftRows(state);
            AddRoundKey(state, roundkey);
      }
```

Code Sample 4.2.1: Pseudo-code for an AES encrypt round transformation

```
KeyExpansion(word8 k[4][MAXKC],
                    word8 W[MAXROUNDS+1][4][MAXBC])
{
int i, j, t=0, RCpointer = 1;
                        word8 tk[4][MAXKC];

for(j=0; j< KC; j++)
      for (i=0; i<4; i++)     tk[i][j] = k[i][j];

for (j=0; (j<KC) && (t <(ROUNDS+1) * BC ); j++, t++)
      for ( i=0; i<4; i++) W[t / BC][i][t % BC] = tk[i][j];
while (t < (ROUNDS + 1)*BC)
      {
      for(i=0; i<4; i++)
            tk[i][0] ^= S[tk[(i+1)%4][KC-1]];
      tk[0][0] ^= RC[RCpointer++];
      if (KC <= 6 )
            for (j=1; j < KC; j++)
              for(i=0; i<4; i++) tk[i][j] ^= tk[i][j-1];
      else {
            for (j=1; j < 4; j++)
              for(i=0; i<4; i++) tk[i][j] ^= tk[i][j-1];
                    for(i=0; i<4; i++) tk[i][4] ^= S[tk[i][3]];
              for (j=5; j < KC; j++)
                    for(i=0; i<4; i++) tk[i][j] ^= tk[i][j-1];
            }
            /* copy values into round key array */
      for (j=0; (j < KC) && (t<(ROUNDS+1)*BC); j++, t++)
            for(i=0; i<4; i++) W[t/BC][i][t%BC] = tk[i][j];
      }
}
```

Code Sample 4.2.2: Code for `KeyExpansion( )`

4.2.3 SubBytes( ) Transformation

This is a non-linear byte-wise substitution of all bytes in the `state`. The substitution transformation is simply called as an S-box. This transformation acts on the individual bytes of the `state`. Figure 4.2.2 presents the `SubBytes()` Transformation.

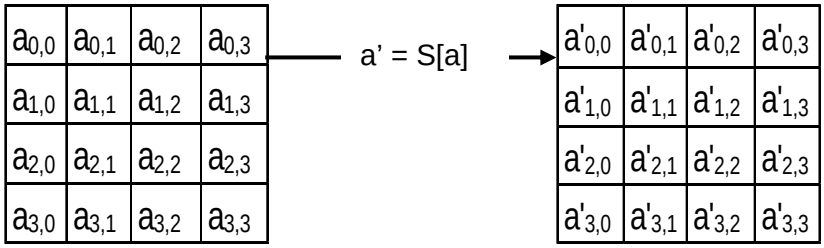

Figure 4.2.2: `SubBytes()` Transformation acts on the individual bytes

The S-box can be implemented by a look-up table or by the following formula.

$$\begin{bmatrix} a'_7 \\ a'_6 \\ a'_5 \\ a'_4 \\ a'_3 \\ a'_2 \\ a'_1 \\ a'_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

The inverse of the S-box needs to be performed in decryption and is implemented by the following formula.

$$\begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} a'_7 \\ a'_6 \\ a'_5 \\ a'_4 \\ a'_3 \\ a'_2 \\ a'_1 \\ a'_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

4.2.4 ShiftRows( ) Transformation

In the `ShiftRows()` transformation, each row of the `state` is considered separately and the bytes in that row are cyclically shifted to the left based upon the key-size of the algorithm. For the 128-bit key, the first row is unchanged. However, the second, third and fourth rows are shifted by 1, 2 and 3 bytes respectively. Figure 4.2.3 presents the `ShiftRows()` operation on the `state`.

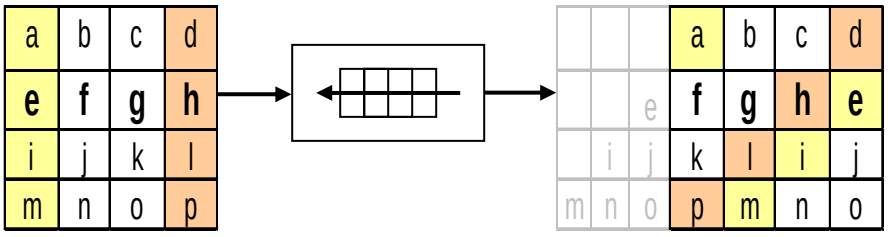

Figure 4.2.3: ShiftRows( ) transformation operating on individual rows

The inverse of ShiftRows is called InvShiftRows. The bytes are shifted towards the right in a cyclic shift in the InvShiftRows transformation. Figure 4.2.4 presents the `InvShiftRows( )` operation on the `state`.

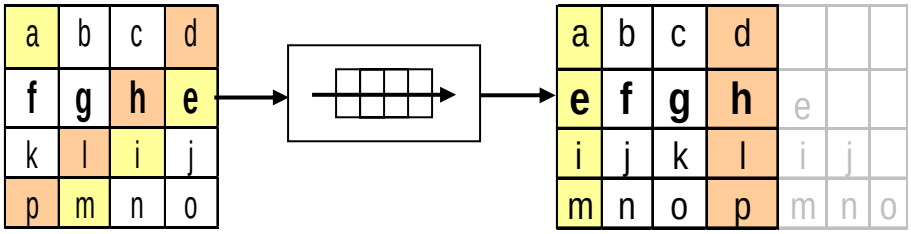

Figure 4.2.4: InvShiftRows( ) transformation operating on individual rows

4.2.5 MixColumns( ) Transformation

The `MixColumns()` Transformation is a bricklayer permutation operating on each column of the `state.` This operation is depicted in the Figure 4.2.5.

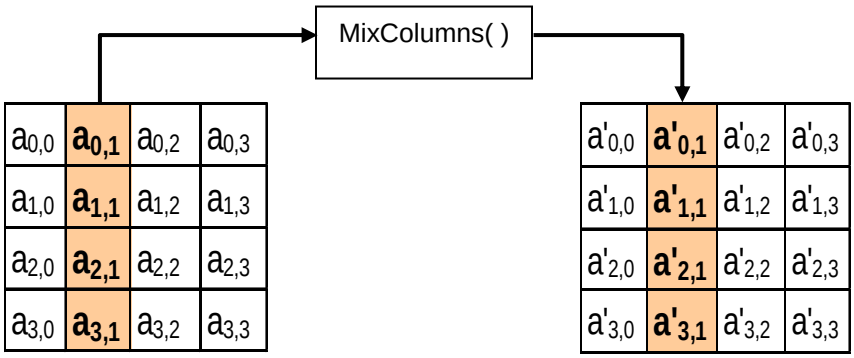

Figure 4.2.5: `MixColumns( )` operation on each column of the `state`

The columns of the `state` are considered as polynomials over GF (2^8) and multiplied modulo x^4+1 with a fixed polynomial c(x). The polynomial *c(x)* is given by

$$c(x) = 03.x^3 + 01.x^2 + 01.x + 02$$

The `MixColumns` operation is implemented by:

$$a'(x) = c(x).a(x) (\text{mod} \quad x^4 + 1)$$

$$\begin{bmatrix} a'_0 \\ a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The inverse operation for `MixColumns` is called `InvMixColumns`. It is implemented by the following formula:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} a'_0 \\ a'_1 \\ a'_2 \\ a'_3 \end{bmatrix}$$

4.2.6 AddRoundKey( ) Transformation

The AddRoundKey( ) Transformation is a simple bitwise XOR operation of the `state` and the round key. A round key is a special key generated for a particular round by the KeyExpansion( ). The length of the round key is equal to the block length.

Figure 4.2.6 shows the `AddRoundKey()` Transformation. It performs bit-wise XOR of the `state` with the `roundkey`. The AddRoundKey is its own inverse and hence the same transformation is also used in decryption.

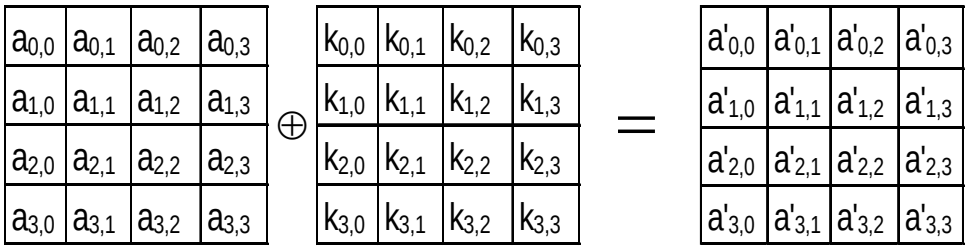

Figure 4.2.6: `AddRoundKey()` transformation

4.2.7 Inverse Cipher

Reversing the encryption steps through the use of their inverse transformations performs decryption. Code Sample 4.2.3 presents the pseudo code for decryption round transformations.

```
int Decrypt (word8 a[4][MAXBC],
             word8 rk[MAXROUNDS+1][4][MAXBC])
{
      int r;

      AddRoundKey(a, rk[ROUNDS]);
      InvSubBytes(a);
      InvShiftRows(a);

      for ( r=ROUNDS-1; r > 0; r--)
      {
            AddRoundKey(a,rk[r]);
            InvMixColumns(a);
            InvSubBytes (a);
            InvShiftRows(a);
      }
      AddRoundKey(a,rk[0]);
}
```

Code Sample 4.2.3: Pseudo-code for AES decryption

CHAPTER 5

IMPLEMENTATION

5.1 Structure

Any embedded system has a set of specific tasks to perform. The objective here is to study the implementation issues of including the cryptographic modules into the embedded system.

An embedded system possesses various functions. At some point of its execution, it may need to use the cryptographic module, either for encryption or decryption of the data. Figure 5.1.1 presents the cryptographic module that was developed and analyzed. The inputs from other modules were taken into the cryptographic module in fixed block sizes. The encryption or decryption was performed on the data and then given back to the host function through the output buffer.

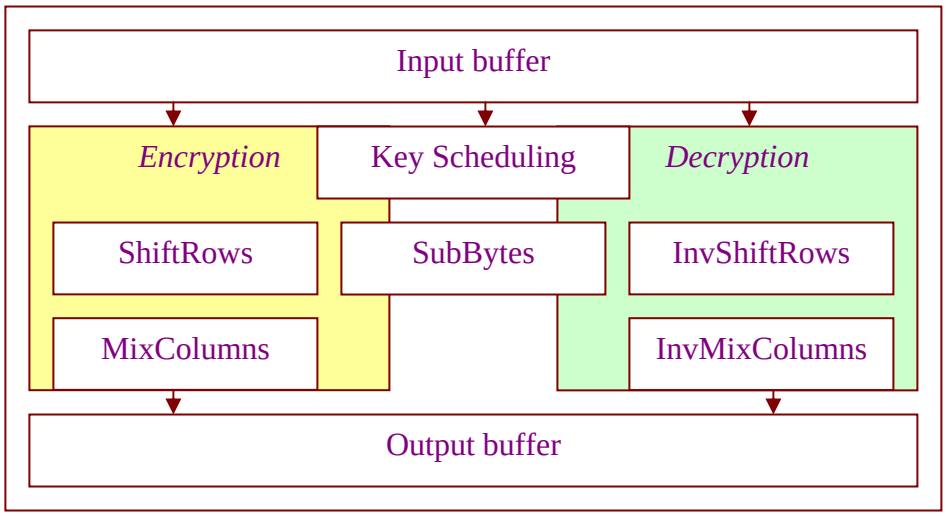

Figure 5.1.1: Encryption and decryption modules

5.2 Development Process

The coding of the program was performed in the C language, which is a high-level language defined at higher abstract levels and is programmer-friendly. The high-level language needs to be compiled into a low-level language before execution. A low-

level language is defined at the register level in order to achieve optimum performance in terms of processing speed, low-memory requirements or both. Figure 5.2.1 presents the development process of the cryptographic system.

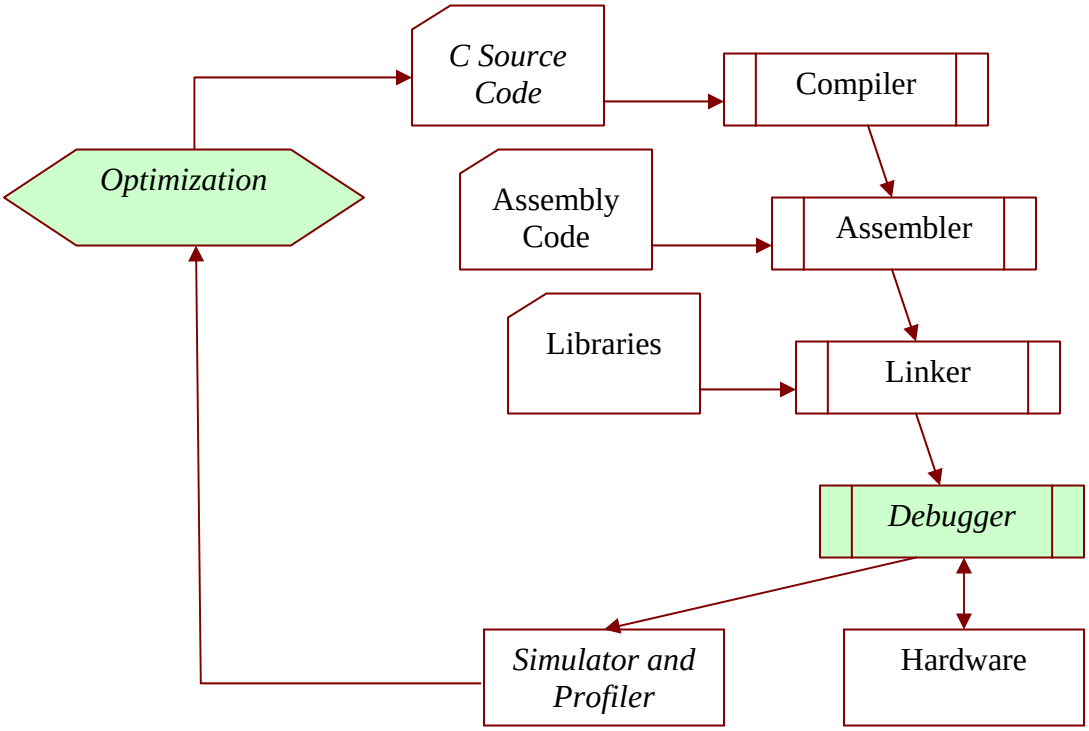

Figure 5.2.1: Block diagram of the system development process

The AES algorithm was implemented and compiled in C. Then the code was linked using the CodeWarrior Integrated Development Environment.

5.2.1 High-level Synthesis

The AES was implemented in C rather than assembly due to the high-level nature of the C language. High-level programming gives the programmer a higher level of flexibility in terms of defining the objective of the code. A compiler that generated the machine code compiled the high-level language. High-level synthesis helped in realizing the project objectives in a lesser amount of time. The main advantage of using a high-level language is code portability.

5.2.2 Low-level Synthesis

Low-level programming involves coding in machine-level instructions. This requires a thorough knowledge at the register levels of the hardware system. Since low-

level synthesis does not support the portability feature, it was not used in this implementation. In terms of very high levels of optimizations, it is advisable to code the program in low-level architecture-specific instructions.

5.2.3 Portability

Portability is the ability of the code to be transferred to a different system or environment with minimal amounts of modification and redevelopment. Since the competition for quick release of products is so tremendous, rapid prototyping and timely marketing defines the success of a product. Therefore, portability aspects were strictly adhered to while developing the system.

5.2.4 Modularization

Modularization is the technique of splitting a large program into smaller modules. The advantage of modularization is the ease of maintenance and code debugging. Modularization helps in code-reuse, which reduces run-time memory. A cryptographic system is developed as a separate module with sub-modules implementing the details. In the C language, modularization is achieved by dividing the code into various functions. When the embedded system needs to encrypt or decrypt data it invokes the corresponding module, which executes its tasks and then returns the output to the host function. The main program was divided into different modules termed `encrypt()`, `decrypt()` and `KeyExpansion()` functions.

5.2.5 Compiler Exploitation

The coding of the program in a high-level language should be such that the compiler would be able to optimize the code to the highest levels possible. Care must be taken to consider the abilities and the limitations of the compiler when the program is being optimized. The StarCore architecture has four ALUs, which can perform four operations in parallel. Thus, the algorithm coding was modified to take advantage of the four ALUs.

5.3 Optimizations

The AES algorithm was implemented in C for generalized key lengths of 128, 196 and 256-bits. The length of the key can be changed by the input parameter

specifications. For optimization purposes, the key length was fixed for 128-bits and the various optimization techniques were applied and analyzed. The main aspects that were considered for optimization were:

- Exploring parallelism in the algorithm
- Multi sample processing and split summation
- Speed optimization at the expense of increasing the code size
- Function call; argument passing increases overhead
- Compiler may use inline functions
- Task priority
- Interrupt service management
- Time-sliced multi-tasking
- I/O queues management
- Interrupt disable while generating the key

5.3.1 Structural Partitioning

Input and output buffers were kept aside so that the core could be processed without any interrupts. The external tasks must not be allowed to enter the critical path.

5.3.2 Critical Paths

In many cases, programs have a high-cost critical path that needs to be optimized. It makes sense to optimize the critical paths to a higher extent than the less critical paths. The `MixColumns()` function takes much longer than other sub-modules. This is due to the `mul()` function that needs to be called numerous times. The optimization for the `MixColumns()` function is presented in Code Sample 5.1.

5.3.3 Computational Complexity

Many programs need to perform highly complex sets of arithmetic functions. Such complex functions can be made simpler by exploring other alternatives such as look-up tables and bit-manipulation. The `SubBytes()` can be implemented by using the formula but it consumes lot of processor cycles. So, `SubBytes()` was implemented by using a look-up table.

```
Initial code:
      word8 b[4][MAXBC];
      int i, j;
      for(j=0; j<BC; j++)
            for(i=0; i<4; i++)
                  b[i][j] = mul(2,a[i][j])
                        ^ mul(3,a[(i+1)%4][j])
                        ^ a[(i+2) % 4][j]
                        ^ a[(i+3) % 4][j];
            for(i=0; i<4; i++)
                  for(j=0; j< BC; j++) a[i][j] = b[i][j];
Modified code:
      word8 b[4];
      word8 temp1[4], temp2[4];
      int  j;
      for(j=0; j< BC; j++)
      {
            b[0]        =  mul(2,a[0][j]);
            temp1[0]    =  mul(3,a[1][j]);
            temp2[0]    =  a[2][j] ^ a[3][j];
            temp2[0]    ^= temp1[0];
            b[0]        ^= temp2[0];

            b[1]        =  a[0][j] ^ a[3][j];
            temp1[1]    =  mul(2,a[1][j]);
            temp2[1]    =  mul(3,a[2][j]);
            temp2[1]    ^= temp1[1];
            b[1]        ^= temp2[1];

            b[2]        =  a[0][j]  ^ a[1][j];
            temp1[2]    =  mul(2,a[2][j]);
            temp2[2]    =  mul(3,a[3][j]);
            temp2[2]    ^= temp1[2];
            b[2]        ^= temp2[2];

            b[3]        =  mul(3,a[0][j]);
            temp1[3]    =  a[1][j]  ^ a[2][j];
            temp2[3]    =  mul(2,a[3][j]);
            temp2[3]    ^= temp1[3];
            b[3]        ^= temp2[3];

            a[0][j]     =  b[0]   ;
            a[1][j]     =  b[1]   ;
            a[2][j]     =  b[2]   ;
            a[3][j]     =  b[3]   ;
      }
```

Code Sample 5.1: Modification of `MixColumns( )`

5.3.4 Reusability and Functionality

Programming should be performed in such a way that the program modules are flexible so they can be used again in the application. The `mul()` function was made common to both `MixColumns()` and `InvMixColumns()`. Since the `mul()` function is just one line code, it was optimized as an inline function by the compiler.

5.3.5 Parallel Tasks

Hardware devices typically have a high-level of parallelism when compared to software devices. Design of an embedded device should include consideration of such parallelism found in the hardware. `ShiftRows()` operates on each individual row at a time. So parallel implementation of four subsequent `ShiftRows()` was feasible.

5.3.6 Instruction-level Parallelism

In the `MixColumns()` function, each byte of the column is a function of four bytes of that column. A matrix multiplication was performed to get the result. The single instruction was broken into different instructions capable of being executed in parallel. The result was the XOR of the individual multiplications. Different register sets were used to perform individual instruction, which resulted in instruction-level parallelism that made the code efficient for multi-processors.

5.3.7 Recursive Tasks

Some tasks in a program need to be executed a finite number of times. Such tasks are called as recursive tasks. Recursive tasks have an overhead that needs to be checked when the instruction sequence should jump out of the loop.

- Loop Unrolling: For a small number of repetitions, the overhead could be removed altogether by replacing the loop with the code components for that fixed number of times. This technique is called loop unrolling. Code Sample 5.2 presents the loop unrolling for `AddRoundKey()` and Code Sample 5.3 presents loop unrolling for `SubBytes()`.

```
Initial code:
      int i, j;
      for (i=0; i<4; i++)
            for(j=0; j<4; j++) a[i][j] ^= rk[i][j];

Modified code:
      int i;
            for (i=0; i<4; i++)
            {
                  a[i][0] ^= rk[i][0];
                  a[i][1] ^= rk[i][1];
                  a[i][2] ^= rk[i][2];
                  a[i][3] ^= rk[i][3];
            }
```

Code Sample 5.2: Loop unrolling for `AddRoundKey( )`

```
Initial code:
            int i, j;
            for (i=0; i<4; i++)
                  for(j=0; j< BC; j++)
                        a[i][j] = box[a[i][j]];

Modified code:
            int i;
            for (i=0; i<4; i++)
            {
                  a[i][0] = box[a[i][0]];
                  a[i][1] = box[a[i][1]];
                  a[i][2] = box[a[i][2]];
                  a[i][3] = box[a[i][3]];
            }
```

Code Sample 5.3: Loop unrolling for `SubBytes( )`

- Loop Merging: When two loops are being executed with similar tasks that can be sequentially adjusted, it is better to combine the two loops into a single loop. This technique is called loop merging. This reduces the total overhead time of executing multiple loops to the overhead of a single loop. Code Sample 5.4 presents the loop unrolling and merging with constants substitution for `ShiftRows( ).`

5.3.8 Pipelining Tasks

Two pipelining tasks were considered in order to reduce code execution time.

- Multi-sample Processing: Sometimes, different samples can be executed simultaneously when there is no inter-dependency among them. This helps in conserving valuable clock cycles.

```
Initial code:
            int i, j;
            for (i=0; i<4; i++)
            {
                  for(j=0; j< BC; j++)
                  tmp[j] = a[i][(j + shifts[BC-4][i]) % BC];
                  for(j=0; j< BC; j++) a[i][j] = tmp[j];
            }

Modified code:
            int i;
            for (i=1; i<4; i++)
            {
                  tmp[0] = a[i][(0 + i) % BC];
                  tmp[1] = a[i][(1 + i) % BC];
                  tmp[2] = a[i][(2 + i) % BC];
                  tmp[3] = a[i][(3 + i) % BC];

                  a[i][0] = tmp[0];
                  a[i][1] = tmp[1];
                  a[i][2] = tmp[2];
                  a[i][3] = tmp[3];
            }
```

Code Sample 5.4: Loop unrolling and merging

- Split Summation: A complex equation can be made simple by dividing it into smaller components so that they can be executed in parallel. Another advantage is that by dividing into smaller components, different registers can be used. This minimizes the number of memory transfers, which consume more cycles than simple register calls. Split summation was performed on `MixColumns()` and `InvMixColumns()`. Code Sample 5.5 presents the modifications in `InvMixColumns()`.

5.3.9 Conditional Tasks

Some tasks use conditional statements like if-then-else, which consume a lot of cycles. A better way is to remove the conditional statements as much as possible. Code Sample 5.6 presents the removal of the if-then-else statement from `ShiftRows()`.

```
Initial code:
      word8 b[4][MAXBC];
      int i, j;

      for(j=0; j<BC; j++)
            for(i=0; i<4; i++)
                  b[i][j] = mul(0xe,a[i][j])
                        ^ mul(0xb,a[(i+1) % 4][j])
                        ^ mul(0xd,a[(i+2) % 4][j])
                        ^ mul(0x9,a[(i+3) % 4][j]);
            for(i=0; i<4; i++)
                  for(j=0; j< BC; j++) a[i][j] = b[i][j];

Modified code:
      word8 b[4];
            word8 temp0[3], temp1[3], temp2[3], temp3[3];
            int  j;
            for(j=0; j< BC; j++)
            {
                  b[0]    = mul(0xe,a[0][j]);
                  temp0[0]= mul(0xb,a[1][j]);
                  temp0[1]= mul(0xd,a[2][j]);
                  temp0[2]= mul(0x9,a[3][j]);
                  temp0[1]^=temp0[0];
                  b[0]    ^=temp0[2];
                  b[0]    ^=temp0[1];

                  b[1]    = mul(0x9,a[0][j]);
                  temp1[0]= mul(0xe,a[1][j]);
                  temp1[1]= mul(0xb,a[2][j]);
                  temp1[2]= mul(0xd,a[3][j]);
                  temp1[1]^=temp1[0];
                  b[1]    ^=temp1[2];
                  b[1]    ^=temp1[1];

                  b[2]    = mul(0xd,a[0][j]);
                  temp2[0]= mul(0x9,a[1][j]);
                  temp2[1]= mul(0xe,a[2][j]);
                  temp2[2]= mul(0xb,a[3][j]);
                  temp2[1]^=temp2[0];
                  b[2]    ^=temp2[2];
                  b[2]    ^=temp2[1];

                  b[3]    = mul(0xb,a[0][j]);
                  temp3[0]= mul(0xd,a[1][j]);
                  temp3[1]= mul(0x9,a[2][j]);
                  temp3[2]= mul(0xe,a[3][j]);
                  temp3[1]^=temp3[0];
                  b[3]    ^=temp3[2];
                  b[3]    ^=temp3[1];

                  a[0][j] = b[0]   ;
                  a[1][j] = b[1]   ;
                  a[2][j] = b[2]   ;
                  a[3][j] = b[3]   ;
            }
```

Code Sample 5.5: Modification of `InvMixColumns( )`

```
Initial code:
      ShiftRows(a,0);   // for ShiftRows
      ShiftRows(a,1);   // for InvShiftRows

      void ShiftRows(word8 a[4][MAXBC], word8 d)
      {
            word8 tmp[MAXBC];
            int i, j;

            if ( d==0)  // for ShiftRows-encryption
            {
                  ...
                  ...
            }
            else        // for ShiftRows-decryption
            {
                  ...
                  ...
            }
      }


Modified code:
      ShiftRows(a);     // for ShiftRows
      InvShiftRows(a);  // for InvShiftRows

      void ShiftRows(word8 a[4][MAXBC]) // for ShiftRows-encryption
      {
            ...
            ...
      }

      void InvShiftRows(word8 a[4][MAXBC]) // for InvShiftRows
      {
            ...
            ...
      }
```

Code Sample 5.6: Removal of If-Then-Else Conditions from `ShiftRows( )`

5.4 Critical Issues

5.4.1 Interrupt Service Management

The cryptographic related modules should be given the highest priority. If the case arises to perform some other critical task, then an interrupt routine should be programmed to check whether any cryptographic module is running at that time. If so, then all cryptic data should be deleted until completion of the interrupt routine. Then the

cryptographic module should be executed again. Under no circumstances should the cryptographic data be sent to the stacks in order to perform interrupt routines.

5.4.2 Time-sliced Multi-tasking

Time-sliced multitasking of a cryptographic module with other applications also presents vulnerability to attacks. Time slicing could help the attacker to read the data of the registers in order to obtain crucial information, which could lead to knowledge of the key.

5.4.3 I/O Queues Management

In order to run the cryptographic modules efficiently, the input and output modules should be structurally separated. When the embedded device has multi-processor capability, separate processing should be catered for I/O data management.

CHAPTER 6

ATTACKS AND COUNTER MEASURES

A deliberate Cryptanalysis to break a cryptosystem is called an attack. While developing a system, cryptography should not be thought of as the final task. Care should be taken to consider the cryptographic issues from the conception to the completion of the system. An insecure system is no different than a secure system when the system functionalities are considered.

Security is the foremost priority in a cryptosystem. The evaluation of security cannot be made by the system functionality. Security is different than functionality. Any designer can design a system with specified functionalities. For a cryptographic system, functionality is necessary but not sufficient. A cryptographic system should be designed with a view to the kind of attacks it might face and the countermeasures to defeat the attacks. A good algorithm is only the starting point.

6.1 Implementation Attacks

Attacks that rely on the flaws in implementation procedure are termed Implementation Attacks. In the process of code optimization for optimum speed and memory, programmers often neglect the transfer of variables and the deletion of trace elements. Care must be taken to delete the round keys and the main key when the encryption or decryption process is complete. It is also to the best advantage if only the specific authorized modules have access to the cryptographic modules. If any attack or discrepancy is detected in the run-time environment, a specific data log must be created or appended and key scheduling must be executed again. Key scheduling takes care of deleting the prior keys and generating or obtaining a new set of keys.

6.2 Side-channel Cryptanalysis

Attacks based on implementation flaws rather than algorithms are called as side-channel attacks [K01]. These attacks rely upon the leaking of side-channel information such as execution time and power consumption.

6.2.1 Timing Attacks

Timing attacks exploit the execution times of the application. The execution time of the algorithm is measured in order to obtain information about the key. In the `MixColumns()` of AES, there are different multiplication sequences that might take different times to execute depending upon the key. Such an attack is very powerful and will often compromise the security of the whole system.

6.2.2 Power Attacks

Power analysis attacks explore the power intake of the system and can obtain vital information of the inner working [GLIPV03]. The power consumption of the device is measured to obtain information about the key.

- Simple Power Analysis: This deals with analyzing the recorded power data and the data sets.
- Differential Power Analysis: This deals with the statistical analysis of the power data by comparing different plain texts and ciphers.

Since power analysis attacks are non-invasive, they are virtually non-detectable. This poses a considerable threat to the security of the system since the damage caused cannot be assessed.

6.2.3 Probing Attacks

A probing attack is a direct physical attack where probes are inserted onto the hardware to examine the memory content and the data transfers on the buses. This can be avoided by physical shielding of the device. Optical probing is the newest technique for probing attacks. Apart from physical shielding, other methods of attack detection must be incorporated in case the attacker breaks the physical shield.

6.2.4 Fault Induction Attacks

Faults or errors are introduced into the device by crude means like exposure to radiation [GLIPV02]. Errors are introduced into the system and the outcomes are analyzed. With a statistical induction of errors, relevant information may be extracted.

6.3 Counter Measures

6.3.1 Constant-time Implementation

Encryption time should be made independent of the value of the key. This can be accomplished by careful implementation of instructions by making them time-independent. One way of achieving this is to use a look-up table that should take a fixed amount of time for every execution.

6.3.2 Power Attacks

Preventing Power attacks can be accomplished in the following ways:

- Physical shielding of the device so that the leaking signal size is reduced.
- Adding noise to the power measurements. This ensures that the attacker will require more samples for analysis.
- Temporal obfuscation of the instructions. This is achieved by randomizing the execution of the instructions. This model controls the power attacks if implemented effectively. If the randomization is not spread properly, it might even aid the attacker in obtaining the relevant information. Temporal obfuscation can also be achieved by randomized clock signals.

6.3.3 Probing Attacks

The system must have the hardware architecture designed in such a way that optical probing shouldn't reveal the state of a bit. A bit should be made as 'HL' or 'LH' instead of a single 'H' or 'L'.

6.3.4 Random Number and Unique Key Generators

Random number generation forms a formidable task in cryptosystems. To state the truth, no finite machine could ever produce a true random number. Any number

generated, which seems to be a random number is called as pseudo-random number. One way of generating pseudo-random numbers is through the use of an algorithm utilizing the linear feedback shift registers. Another concept of generating random numbers is to use a conventional cryptographic algorithm. As input to the cryptographic algorithm, a user-generated number is used and this number is called the seed to the pseudo-random number generator. The seed should be carefully selected to prevent any malicious user from guessing it or reproducing it by any other means [KSWH98]. The keys for cryptosystems should be based on random numbers and care must be taken that such numbers pass the random number tests. The unique key generator for the cryptosystem should carefully perform the task of generating the session keys. The longer the session key is in use, the more vulnerable is the cryptosystem to the attacks. The registers that hold the keys should be volatile and the key should be deleted when it is no longer needed.

CHAPTER 7

RESULTS AND DISCUSSION

7.1 Results

The Advanced Encryption Standard (AES) was implemented in C and various optimization techniques were applied without compromising for the security issues. The initial program is called by the name ‘opt-a’ and different code optimizations were performed at different stages and the different stages of the code are named as ‘opt-c’, ‘opt-d’...‘opt-h’. The suffix to the code name indicates the compiler optimization level. The suffix ‘0’ indicates that no compiler optimizations have been performed. For instance, ‘LEVEL 3’ indicates that compiler optimizations of scheduling, pipelining and bundling are performed on the code. The suffix ‘space’ indicates that space optimizations were performed rather than speed optimizations. The objective was to optimize the code in terms of speed without degrading the space parameter. The compiler option of space optimization was considered to show how the space optimization techniques affect the speed optimization process.

The original unoptimized code was conceived into ‘Opt a’. The program was compiled using the CodeWarrior IDE. The profiler of the CodeWarrior aided in obtaining the statistical information about the program execution.

Table 7.1.1 presents the different stages of the code development and the corresponding execution time in clock cycles.

Table 7.1.1: Execution time in clock cycles at various stages of code development

Stages	main( )	encrypt( )	decrypt( )
LEVEL 0	453897	112706	153576
LEVEL 3	217617	33057	52721
LEVEL 3 Space	265766	43019	66184
Opt-a 0	630850	111987	154316
Opt-a 3	231068	38040	54457
Opt-c 3	215032	34238	50244
Opt-d 0	599351	106537	143926
Opt-d 3	215032	34238	50244
Opt-d 3 Space	260780	42738	63989
Opt-d Space	271728	44693	67367
Opt-e 0	557991	96577	133206
Opt-e 3	211002	32801	49684
Opt-e 3 Space	258734	42146	63558
Opt-e Space	268471	43926	66504
Opt-f 0	536111	91134	127709
Opt-f 3	207203	31761	48802
Opt-f 3 Space	254034	41069	62291
Opt-f Space	265502	43189	65758
Opt-g 0	516591	86264	122819
Opt-g 3	194268	28676	45432
Opt-g 3 Space	242633	38223	59405
Opt-g Space	252066	39813	62432
Opt-h 0	523464	85133	127651
Opt-h 3	197996	29125	46882
Opt-h 3 Space	246797	37966	61774
Opt-h Space	253263	39656	63186

Table 7.1.2 presents the profiling information for the ‘opt-g 3’ stage. The table depicts the function (F) and descendent (D) time in clock cycles of various functions.

Table 7.1.2: Profiling information for ‘opt-g 3’ stage

Function	Calls	F time	F+D time	% F time	% F+D time	Avg. F time	Avg. F+D ti...
main	1	3118	194268	1.60	100.00	3118	194268
Decrypt	2	260	90865	0.13	46.77	130	45432
InvMixColumns	18	69521	69521	35.79	35.79	3862	3862
Encrypt	2	238	57352	0.12	29.52	119	28676
printf	75	1184	42048	0.61	21.64	15	560
__doprnt	75	20429	40639	10.52	20.92	272	541
MixColumns	18	36090	36090	18.58	18.58	2005	2005
InvShiftRows	20	18960	18960	9.76	9.76	948	948
ShiftRows	20	18900	18900	9.73	9.73	945	945
fwrite	75	12049	13874	6.20	7.14	160	184
fill_oh_word	64	3936	3936	2.03	2.03	61	61
memcpy	146	2527	2527	1.30	1.30	17	17
AddRoundKey	44	2288	2288	1.18	1.18	52	52
SubBytes	40	1960	1960	1.01	1.01	49	49
isdigit	128	1344	1344	0.69	0.69	10	10
exit	1	20	885	0.01	0.46	20	885
__stdio_clea...	1	798	865	0.41	0.45	798	865
__write	10	340	340	0.18	0.18	34	34
__get_stdout	77	230	230	0.12	0.12	2	2
fflush	2	59	59	0.03	0.03	29	29
atexit	1	14	14	0.01	0.01	14	14
__get_stderr	2	2	2	0.00	0.00	1	1
__get_stdin	1	1	1	0.00	0.00	1	1

The CodeWarrior profiler also displays a graphical representation of the function and its descendents. Figure 7.1.1 presents the graphical display of the `encrypt()` of the ‘opt-g 3’ stage. It can be deduced that the `encrypt()` is consuming 57352 cycles per two calls, which means the average execution time for `encrypt()` is 28676 clock cycles. The main metrics for this thesis is the execution time defined in terms of the number of clock cycles taken by the hardware to execute a particular function

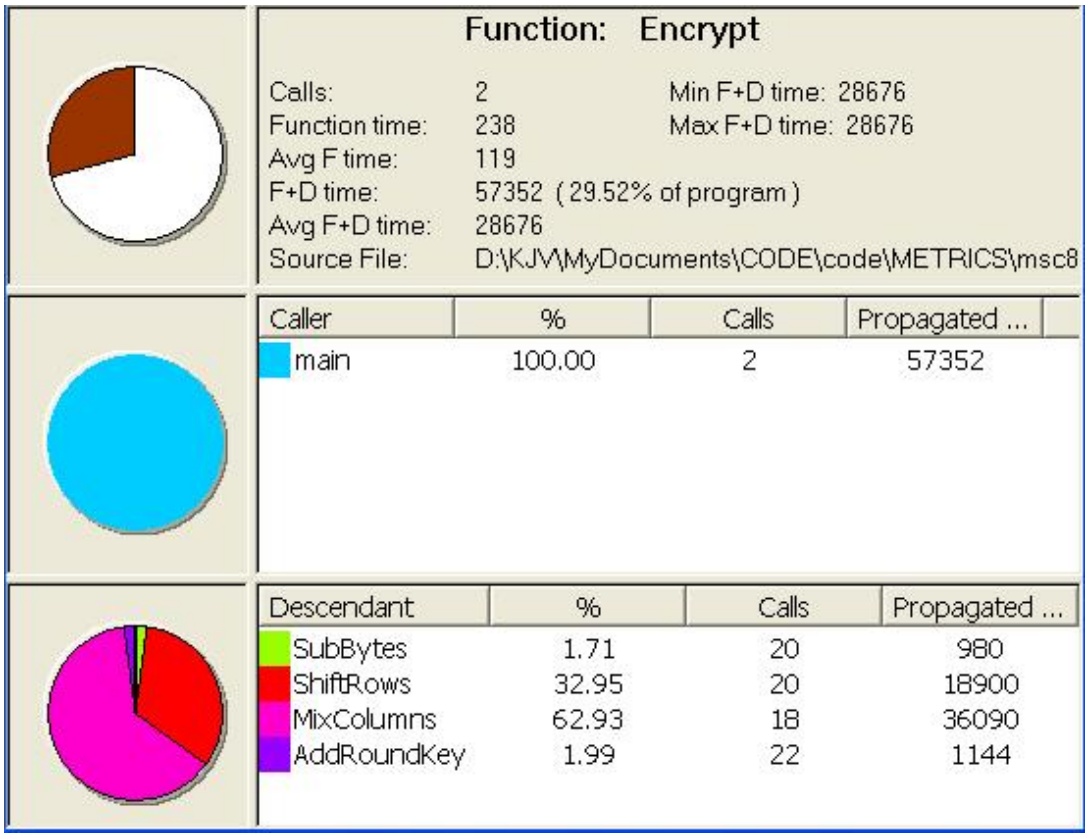

Figure 7.1.1: Graphical profile for `encrypt()` of 'opt-g 3' Stage

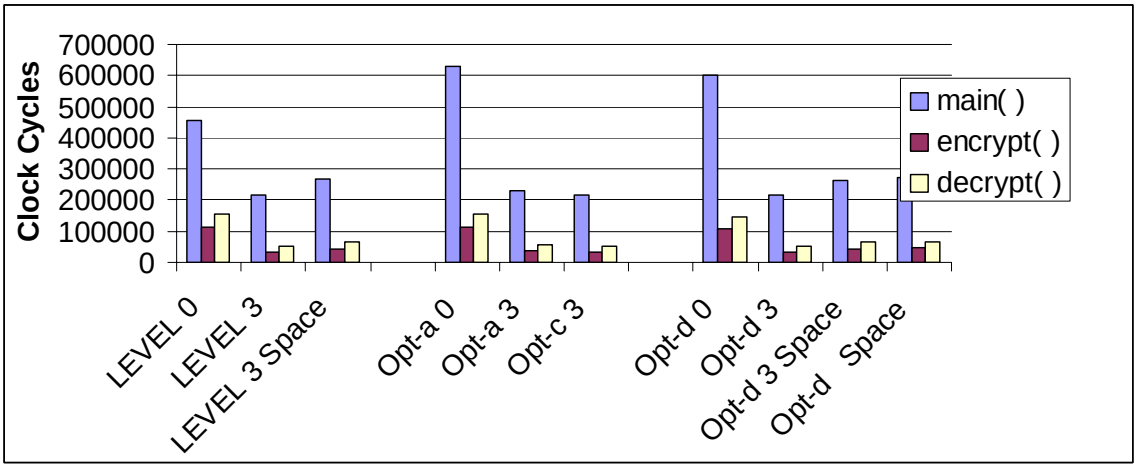

Figure 7.1.2: Functions `main()`, `encrypt()` and `decrypt()` from stages Level 0 to Opt-d Space stages

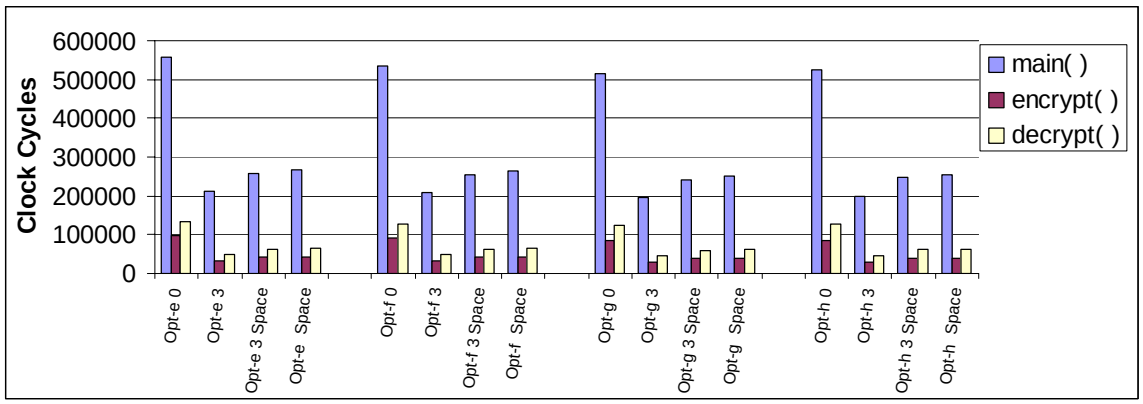

Figure 7.1.3: Functions `main()`, `encrypt()` and `decrypt()` from stages Opt-e to Opt-h Space stages

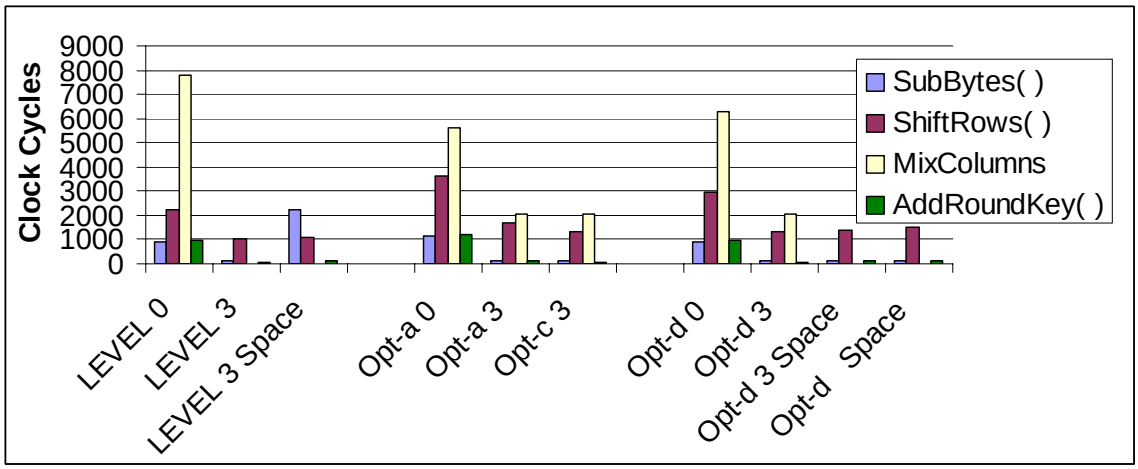

Figure 7.1.4: Cryptographic modules from Level 0 to Opt-d Space stages

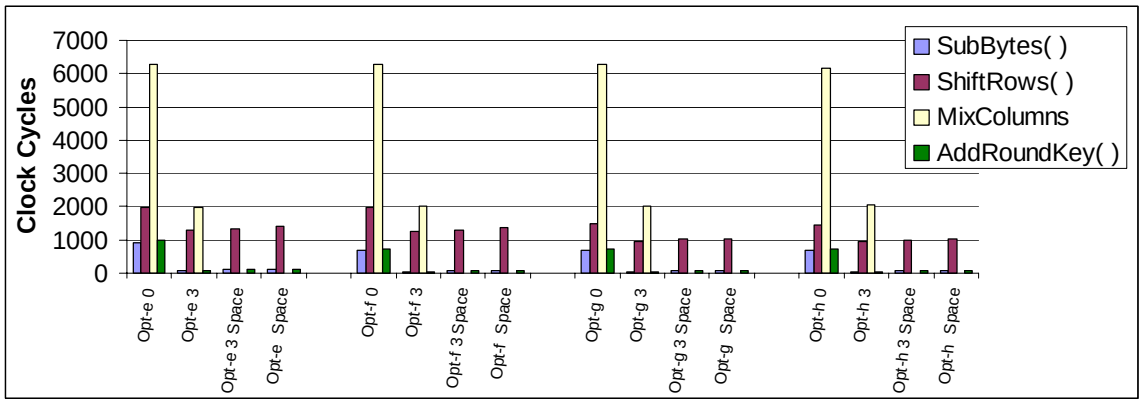

Figure 7.1.5: Cryptographic modules from Opt-e 0 to Opt-h Space stages

The `encrypt()` function is divided into four sub-functions: `SubBytes()`, `ShiftRows()`, `MixColumns()` and `AddRoundKey()`. The function and descendent times of various functions are depicted in Figure 7.1.2 through Figure 7.1.5. Observations from the various stages of the code optimizations revealed that the `MixColumns()` function was consuming more time than other sub-modules combined in the `encrypt()` function. This was due to the `mul()` function in the `MixColumns()` function, which was used to perform the Galois Field (GF) multiplication on the data operands. GF multiplication was performed by implementing a look-up table to defeat any timing attacks. Figure 7.1.2 reveals that `decrypt()` takes more time than `encrypt()`. This is due to the added complexity of the GF multiplication in `InvMixColumns()` of `decrypt()`. The `InvMixColumns()` needs to perform four multiplications while the `MixColumns()` needs to perform only two multiplications per each byte of the `state`.

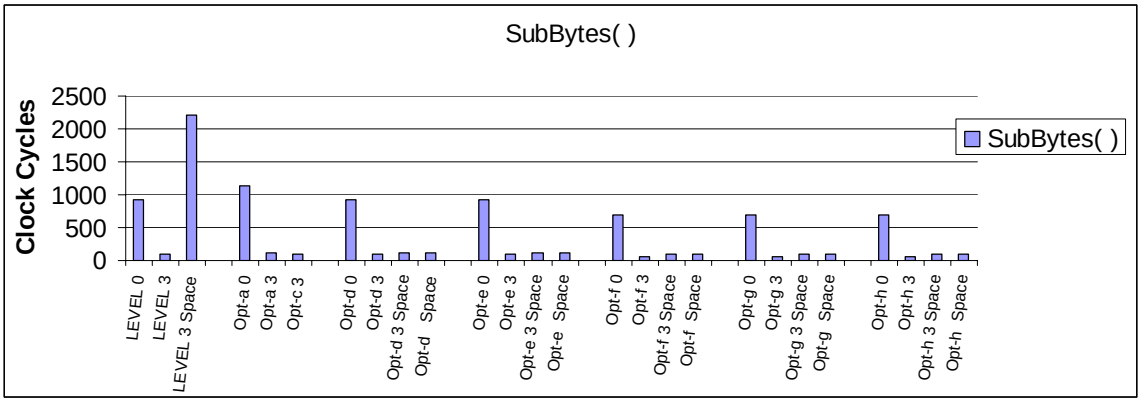

Figure 7.1.6: `SubBytes()` module at all stages

The `SubBytes()` initial function consumed 927 cycles without compiler optimization and 93 cycles with compiler-optimization. After optimization techniques were applied, the function required 684 clock cycles without compiler optimization and 49 cycles with compiler optimization. Thus `SubBytes()` showed a performance gain of 26% without-compiler optimization and 47% with-compiler optimization in terms of execution speed. Figure 7.1.6 presents the execution time of `SubBytes()` at all stages of code optimization.

The `ShiftRows()` initial function consumed 2974 cycles without compiler optimization and 1354 cycles with compiler-optimization. After optimization techniques

were applied, the function required 1453 clock cycles without compiler optimization and 945 cycles with compiler optimization. Thus `ShiftRows()` displayed a performance gain of 51% without-compiler optimization and 30% with-compiler optimization in terms of execution speed. Figure 7.1.7 presents the execution time of `ShiftRows()` at all stages of code optimization.

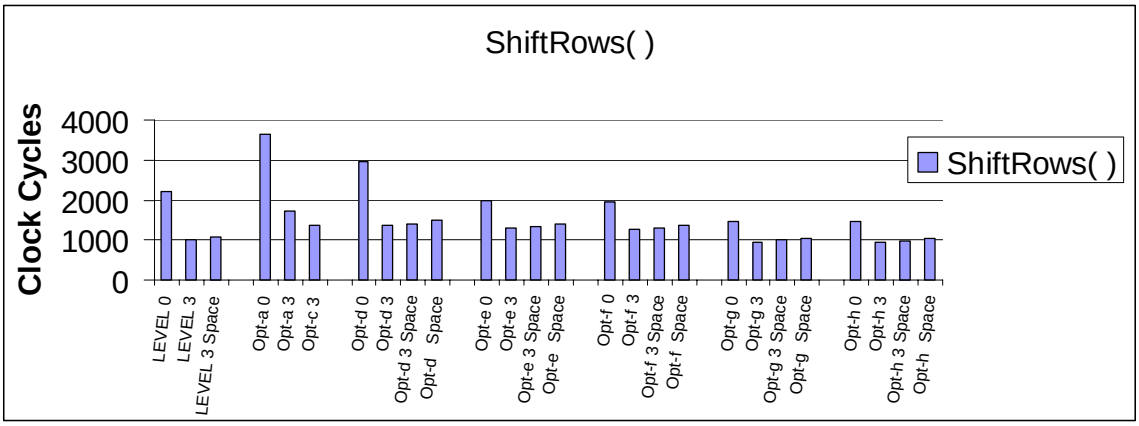

Figure 7.1.7: `ShiftRows()` module at all stages

The `AddRoundKey()` initial function consumed 975 cycles without compiler optimization and 83 cycles with compiler-optimization. After optimization techniques were applied, the function required 705 clock cycles without compiler optimization and 52 cycles with compiler optimization. Thus `AddRoundKey()` displayed a performance gain of 27% without-compiler optimization and 37% with-compiler optimization in terms of execution speed. Figure 7.1.8 presents the execution time of `AddRoundKey()` at all stages of code optimization.

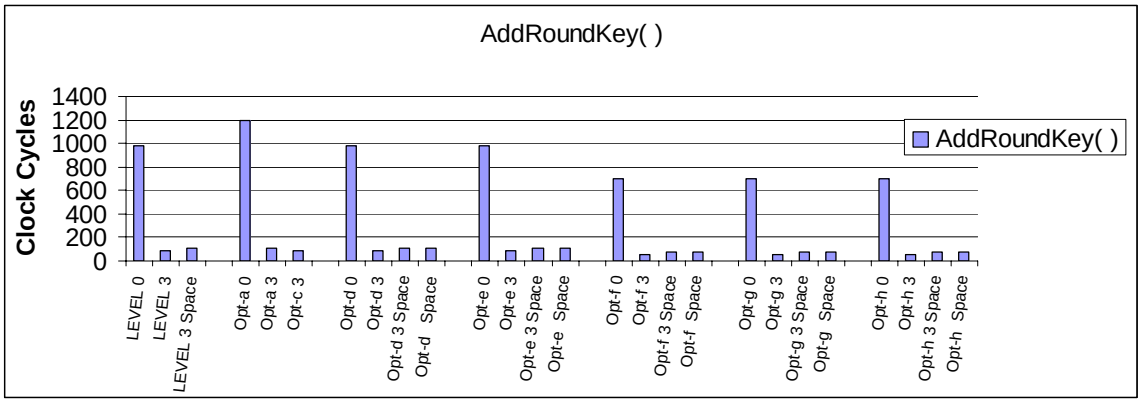

Figure 7.1.8: `AddRoundKey()` module at all stages

Figure 7.1.9 presents execution times of `MixColumns()` at all stages of code optimization.

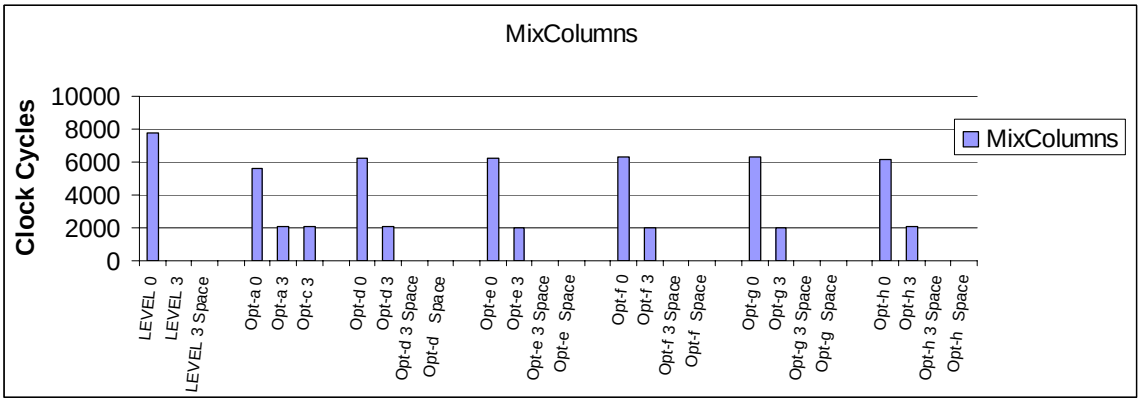

Figure 7.1.9: `MixColumns()` module at all stages

Figure 7.1.10 presents the execution time of `encrypt()`at all stages of code optimization.

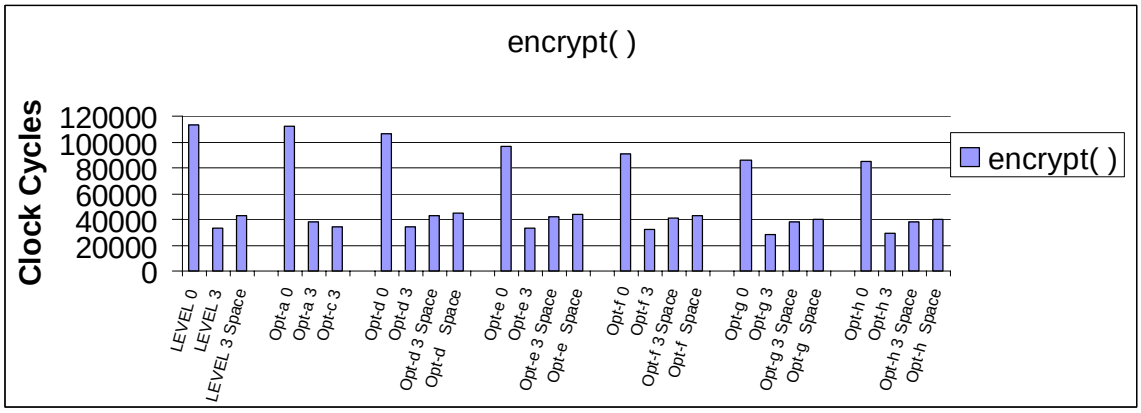

Figure 7.1.10: `Encrypt()` module at all stages

Figure 7.1.11 shows the execution time of `decrypt()`at all stages of code optimization.

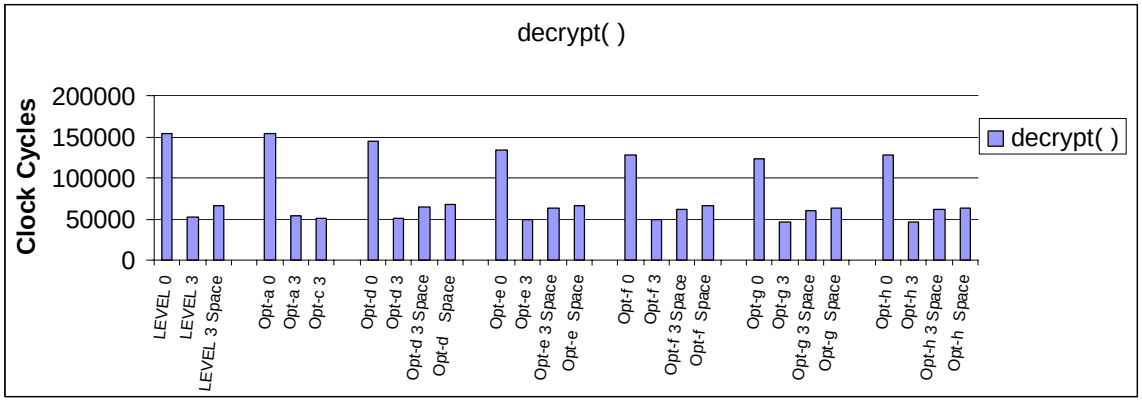

Figure 7.1.11: `Decrypt()` module at all stages

The stages that were optimized by the compiler are illustrated in Figure 7.1.12 through Figure 7.1.18.

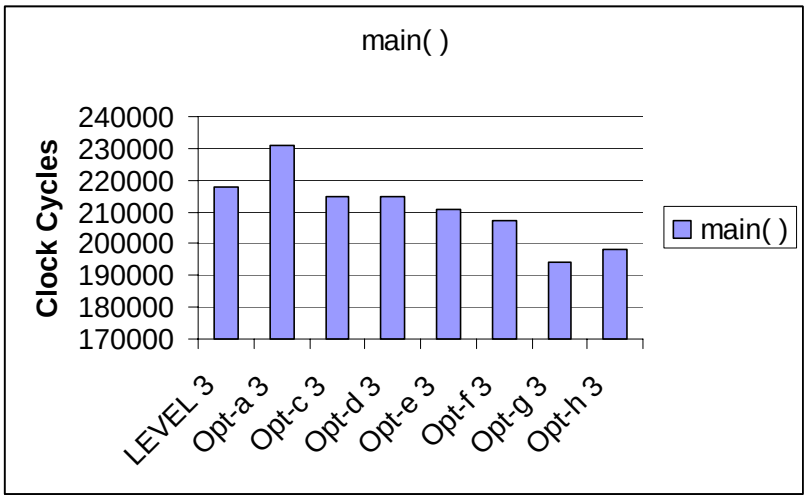

Figure 7.1.12: `Main()` function for compiler optimization

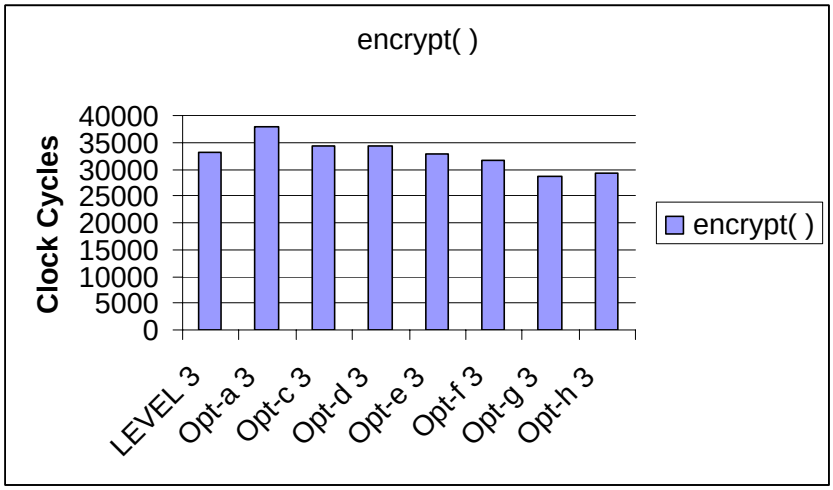

Figure 7.1.13: `Encrypt()` function for compiler optimization

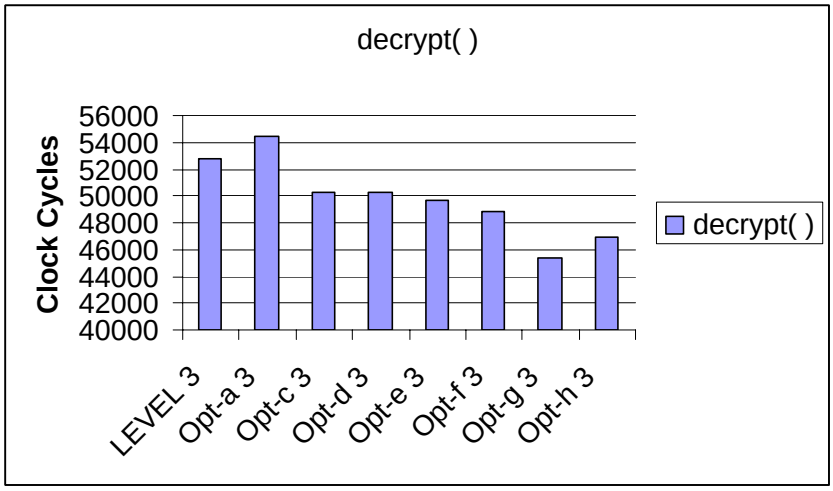

Figure 7.1.14: `Decrypt()` function for compiler optimization

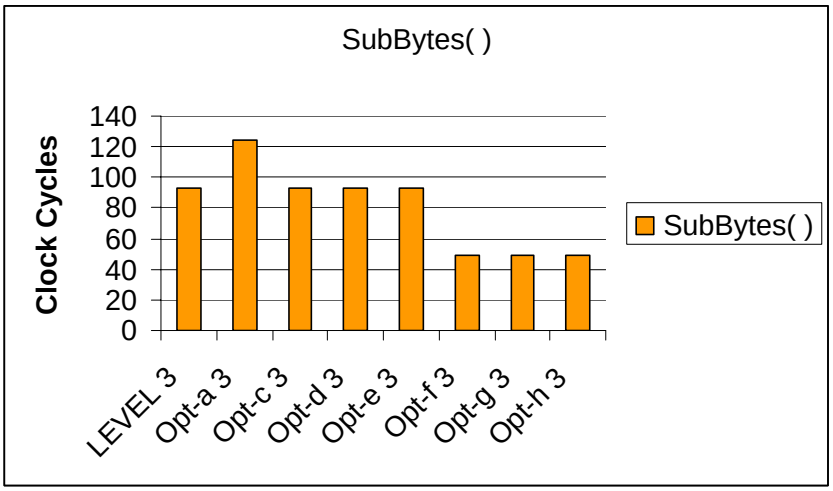

Figure 7.1.15: `SubBytes()` function for compiler optimization

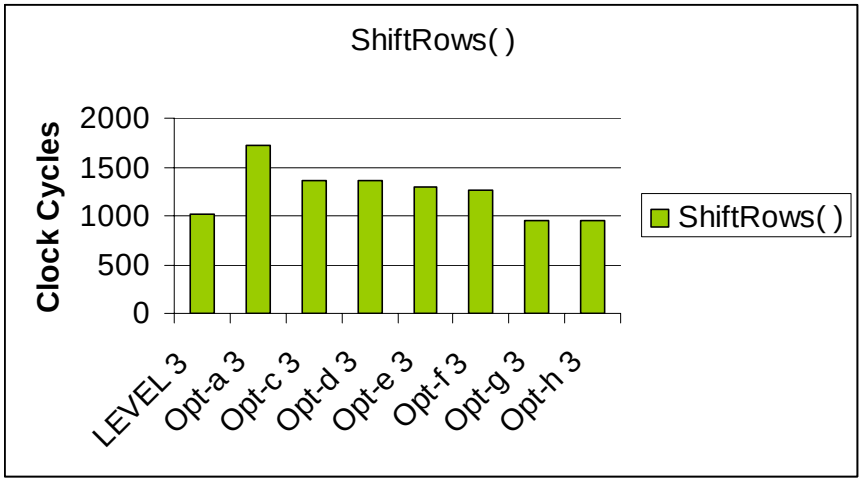

Figure 7.1.16: `ShiftRows()` function for compiler optimization

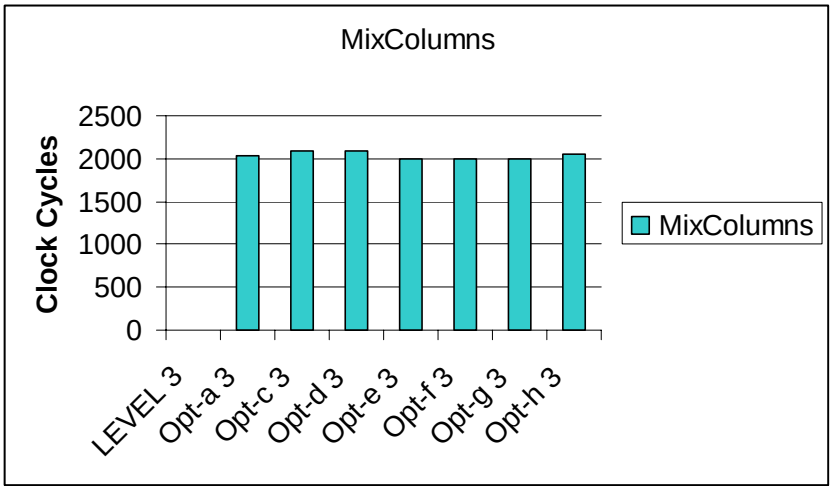

Figure 7.1.17: `MixColumns()` function for compiler optimization

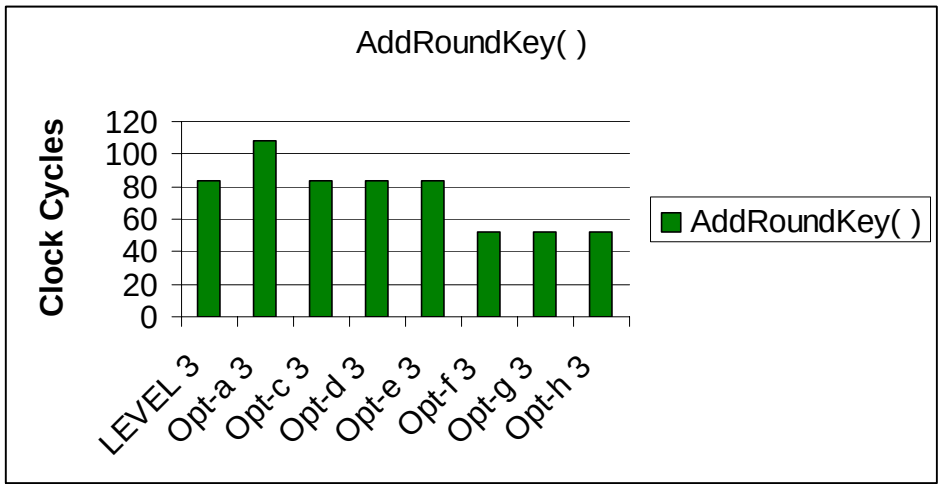

Figure 7.1.18: `AddRoundKey()` function for compiler optimization

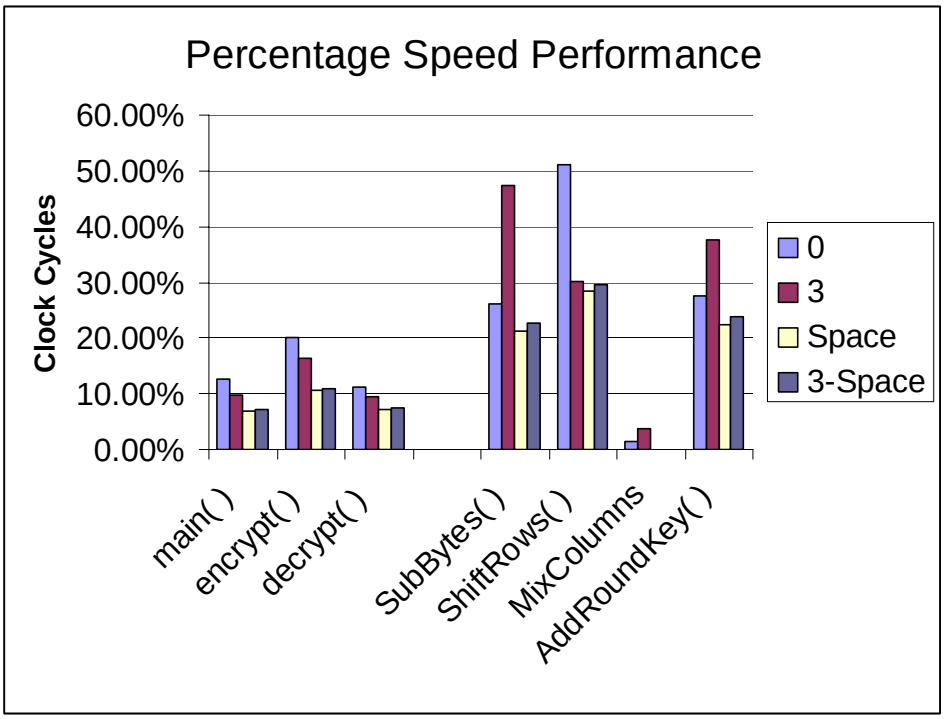

Figure 7.1.19: Speed performance of various modules

Figure 7.1.19 shows the speed performance gained for the AES for different compiler options. This shows that a 20% performance gain was obtained by optimizations for

`encrypt()` and an 11% performance gain by optimizations for `decrypt()`. `ShiftRows()` was optimized better than all other sub-functions. `MixColumns()` was the critical path in the code and it was the least optimized function. This was due to the implementation of a look-up table for the GF multiplication. The `MixColumns()` data for 'Space' and '3-Space' was not obtained due to the compiler's option of inline functioning for better optimization. Table 7.1.3 presents the speed performance gain in percentage for each function.

Table 7.1.3: Speed performance of various modules

Percentage Speed Performance (in %)				
	0	3	Space	3-Space
main( )	12.66153	9.656237	6.958739	7.235912
encrypt( )	20.09067	16.24511	10.56437	10.91894
decrypt( )	11.30789	9.575368	7.163731	7.325545
SubBytes( )	26.21359	47.31183	21.36752	22.68908
ShiftRows( )	51.14324	30.20679	28.47025	29.64213
MixColumns	1.405526	3.65209	NA	NA
AddRoundKey( )	27.69231	37.48634	22.3301	23.80952

7.2 Discussion

This research started with an idea of implementing a cryptographic algorithm on a DSP chip. The idea expanded from the initial concept of a data encryption standard (DES) [SASR01] on the DSP 56824 to the present form of an AES on the StarCore. The AES was implemented on the StarCore using the CodeWarrior IDE. The initial inclination was to put the code into the DSP memory. Since the idea of a secure and reliable system was maintained and thrived from the beginning of this endeavor, care was taken about certain situations where the system might be attacked. Different ways of side-channel attacks were studied and the system was made robust to such attacks by taking care of details such as constant-time implementation and key deletion after the session. Even though the system was made strong, it cannot be denied that successful attacks can be made on it.

In the context of smart attacks, the following security issues should be kept in mind while implementing a cryptographic system.

- Formulate the system to detect various types of attacks.

- Turn off the power to the internal RAM when system tampering is detected.
- The keys should always be stored in volatile memory.
- Change the keys for every session.
- Metal shield the device to prevent the leakage of high-frequency pulses.
- Implement constant-time algorithms.
- Use double sized blocks with complement data to defeat power attacks.

7.2.1 8-bit Platforms

The use of AES is imminent on the 8-bit platforms like smart cards. Smart cards are very vulnerable to side-channel cryptanalysis. This is due to their inherent weaknesses such as dependence on the external clock and their susceptibility to optical probing. In order to reduce the memory size of the 8-bit platforms, key expansion should be performed every time encryption or decryption is performed. This is due to the fact that these devices are generally asynchronous in nature. Since the encryption or decryption is performed only on a relatively small amount of data, the key should be generated and expanded each time the cryptographic module is called.

Look-up tables consume a lot of memory. Therefore, they should be generated by suitable algorithms before encryption or decryption is performed. This might increase the code size but it definitely improves the overall system performance by decreasing the dormant memory occupation.

7.2.2 32/64-bit Platforms

The devices with larger data-bus lengths and processing word lengths are generally synchronous in nature. If the data to be encrypted or decrypted is continuous, the key scheduling should be performed outside the encryption and decryption modules. This saves the repetitive task of key scheduling by moving it out of the continuous routines.

7.3.3 Optimization

There are many optimization metrics concerned with embedded systems such as:

- Production cost
- Execution speed

- Memory size
- Data throughput
- Power consumption
- Robustness

When dealing with cryptosystems, the added issues of security affect all the above metrics. The security issues become the top priority in the metrics to evaluate the system. The difference between a poorly designed and perfectly designed cryptographic system cannot be assessed until an attack is made on both of them. The only way to make the systems more robust is to learn from the previous attacks and explore the weaknesses in the present systems by performing new kinds of attacks. Thus, the ideal way of summarizing this concept is 'the real security of a system cannot be assessed until it is broken'.

CHAPTER 8

CONCLUSION AND RECOMMENDATIONS FOR FUTURE WORK

8.1 Conclusion

The AES was implemented on the StarCore using the CodeWarrior IDE. Additionally, various speed optimization techniques were applied. A study of different side-channel attacks was performed and proposals were made to counter such attacks. The system's security was given greater importance than speed optimization. The programming was divided into different stages and the results were observed at each stage. Each sub-module of `encrypt()` was optimized and the clock cycles were observed. Relevant observations of this research are:

- The compiler was able to effectively optimize a simple code.
- Code optimization techniques boosted the compiler's ability to further optimize the code.
- In-line functions are better suited for smaller functions.
- Function call parameters should be replaced by memory pointers.
- Deletion of the round keys and the main key at the termination of the program improves security.
- Critical paths should be optimized more than the less critical paths.
- Probing attacks should be defeated by physical and radiation shielding.
- Constant-time implementations should be considered to avoid timing and power attacks.

8.2 Recommendations for Future Work

The study and analysis of cryptosystems is an ongoing effort in which new attacks are constantly discovered. Future study to this research should be in the following topics:

- An extensive study must be made of side-channel attacks.
- Architectural advantages must be explored with a consideration for portability.

- Experiments should be conducted to study the effects of various types of attacks.
- New methods such as algorithm switching must be implemented. Key scheduling must be made more robust.
- Attack detection must be incorporated into the system.
- Interrupt priorities should be assigned appropriately with the highest concern for the security of the system.

REFERENCES

[BS96] Bruce Schneier: Applied Cryptography, Protocols, Algorithms and Source Code in C, John Wiley & Sons, Inc, 1996.

[DR01] J. Daemen and V. Rijmen: AES Proposal Rijndael. National Institute of Standards and Technology, July 2001.

[G99] B. Gladman: Input and Output Block Conventions for AES Encryption Algorithms, AES Round 2 public comment, June 6, 1999.

[GLIPV02] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, Vincenzo Piuri: On the Propagation of Faults and Their Detection in a Hardware Implementation of the Advanced Encryption Standard, The IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02) July 17 - 19, 2002 San Jose, California.

[GLIPV03] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, Vincenzo Piuri: Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard, IEEE Transactions on Computers 52(4), 492-505 (2003).

[K01] F. Koeune: Careful design and integration of cryptographic primitives with contributions to timing attack, padding schemes and random number generators, Ph.D. thesis, UCL, July 2001.

[KSWH98] J. Kelsey, B. Schneier, D. Wagner, and C. Hall: Cryptanalytic Attacks on Pseudorandom Number Generators, Fast Software Encryption, Fifth International Workshop Proceedings (March 1998), Springer-Verlag, 1998, pp. 168-188.

[RL00] Rainer Leupers: Code Optimization Techniques for Embedded Processors –Methods, Algorithms, and Tools, Kluwer Academic Publishers, 2000.

[SASR01] C. Sanchez-Avila and R. Sanchez-Reillo: The Rijndael Block Cipher (AES Proposal): A Comparison with DES, 35th IEEE International Conference on Security Technology. London (Reino Unido), 19-16 Octubre, 2001. pps. 229-234.

[SJBW96] Sanjaya Kumar, James H. Aylor, Barry W. Johnson, Wm. A. Wulf: The Codesign of Embedded Systems-A Unified Hardware/Software Representation, Kluwer Academic Publishers, 1996.

BIBLIOGRAPHY

Krishnendu Chakrabarty, Vikram Iyengar, Anshuman Chandra: Test Resource Partitioning for System-on-a-chip, Kluwer Academic Publishers, 2002.

Juan Carlos Lopez, Roman Hermida and Walter Geisselhardt: Advanced Techniques for Embedded Systems Design and Test, Kluwer Academic Publishers, 1998.

Keith Tizzard: C for Professional Programmers, Ellis Horwood Limited, 1986.

APPENDICES

APPENDIX A

Optimized C Code for the AES

```
/*
 * Code Optimization techniques for encryption and decryption
 */

#include <stdio.h>

typedef unsigned char word8;
typedef unsigned int word32;

/* The tables Logtable and Alogtable are used to perform
 * multiplications in GF(256)
 */
word8 Logtable[256] = {
  0,   0,  25,   1,  50,   2,  26, 198,  75, 199,  27, 104,  51, 238, 223,   3,
100,   4, 224,  14,  52, 141, 129, 239,  76, 113,   8, 200, 248, 105,  28, 193,
125, 194,  29, 181, 249, 185,  39, 106,  77, 228, 166, 114, 154, 201,   9, 120,
101,  47, 138,   5,  33,  15, 225,  36,  18, 240, 130,  69,  53, 147, 218, 142,
150, 143, 219, 189,  54, 208, 206, 148,  19,  92, 210, 241,  64,  70, 131,  56,
102, 221, 253,  48, 191,   6, 139,  98, 179,  37, 226, 152,  34, 136, 145,  16,
126, 110,  72, 195, 163, 182,  30,  66,  58, 107,  40,  84, 250, 133,  61, 186,
 43, 121,  10,  21, 155, 159,  94, 202,  78, 212, 172, 229, 243, 115, 167,  87,
175,  88, 168,  80, 244, 234, 214, 116,  79, 174, 233, 213, 231, 230, 173, 232,
 44, 215, 117, 122, 235,  22,  11, 245,  89, 203,  95, 176, 156, 169,  81, 160,
127,  12, 246, 111,  23, 196,  73, 236, 216,  67,  31,  45, 164, 118, 123, 183,
204, 187,  62,  90, 251,  96, 177, 134,  59,  82, 161, 108, 170,  85,  41, 157,
151, 178, 135, 144,  97, 190, 220, 252, 188, 149, 207, 205,  55,  63,  91, 209,
 83,  57, 132,  60,  65, 162, 109,  71,  20,  42, 158,  93,  86, 242, 211, 171,
 68,  17, 146, 217,  35,  32,  46, 137, 180, 124, 184,  38, 119, 153, 227, 165,
103,  74, 237, 222, 197,  49, 254,  24,  13,  99, 140, 128, 192, 247, 112,   7,
};

word8 Alogtable[256] = {
  1,   3,   5,  15,  17,  51,  85, 255,  26,  46, 114, 150, 161, 248,  19,  53,
 95, 225,  56,  72, 216, 115, 149, 164, 247,   2,   6,  10,  30,  34, 102, 170,
229,  52,  92, 228,  55,  89, 235,  38, 106, 190, 217, 112, 144, 171, 230,  49,
 83, 245,   4,  12,  20,  60,  68, 204,  79, 209, 104, 184, 211, 110, 178, 205,
 76, 212, 103, 169, 224,  59,  77, 215,  98, 166, 241,   8,  24,  40, 120, 136,
131, 158, 185, 208, 107, 189, 220, 127, 129, 152, 179, 206,  73, 219, 118, 154,
181, 196,  87, 249,  16,  48,  80, 240,  11,  29,  39, 105, 187, 214,  97, 163,
254,  25,  43, 125, 135, 146, 173, 236,  47, 113, 147, 174, 233,  32,  96, 160,
251,  22,  58,  78, 210, 109, 183, 194,  93, 231,  50,  86, 250,  21,  63,  65,
195,  94, 226,  61,  71, 201,  64, 192,  91, 237,  44, 116, 156, 191, 218, 117,
159, 186, 213, 100, 172, 239,  42, 126, 130, 157, 188, 223, 122, 142, 137, 128,
155, 182, 193,  88, 232,  35, 101, 175, 234,  37, 111, 177, 200,  67, 197,  84,
252,  31,  33,  99, 165, 244,   7,   9,  27,  45, 119, 153, 176, 203,  70, 202,
 69, 207,  74, 222, 121, 139, 134, 145, 168, 227,  62,  66, 198,  81, 243,  14,
 18,  54,  90, 238,  41, 123, 141, 140, 143, 138, 133, 148, 167, 242,  13,  23,
 57,  75, 221, 124, 132, 151, 162, 253,  28,  36, 108, 180, 199,  82, 246,   1,
};


word8 S[256] = {
 99, 124, 119, 123, 242, 107, 111, 197,  48,   1, 103,  43, 254, 215, 171, 118,
202, 130, 201, 125, 250,  89,  71, 240, 173, 212, 162, 175, 156, 164, 114, 192,
183, 253, 147,  38,  54,  63, 247, 204,  52, 165, 229, 241, 113, 216,  49,  21,
  4, 199,  35, 195,  24, 150,   5, 154,   7,  18, 128, 226, 235,  39, 178, 117,
  9, 131,  44,  26,  27, 110,  90, 160,  82,  59, 214, 179,  41, 227,  47, 132,
 83, 209,   0, 237,  32, 252, 177,  91, 106, 203, 190,  57,  74,  76,  88, 207,
208, 239, 170, 251,  67,  77,  51, 133,  69, 249,   2, 127,  80,  60, 159, 168,
 81, 163,  64, 143, 146, 157,  56, 245, 188, 182, 218,  33,  16, 255, 243, 210,
```

```
205,  12,  19, 236,  95, 151,  68,  23, 196, 167, 126,  61, 100,  93,  25, 115,
 96, 129,  79, 220,  34,  42, 144, 136,  70, 238, 184,  20, 222,  94,  11, 219,
224,  50,  58,  10,  73,   6,  36,  92, 194, 211, 172,  98, 145, 149, 228, 121,
231, 200,  55, 109, 141, 213,  78, 169, 108,  86, 244, 234, 101, 122, 174,   8,
186, 120,  37,  46,  28, 166, 180, 198, 232, 221, 116,  31,  75, 189, 139, 138,
112,  62, 181, 102,  72,   3, 246,  14,  97,  53,  87, 185, 134, 193,  29, 158,
225, 248, 152,  17, 105, 217, 142, 148, 155,  30, 135, 233, 206,  85,  40, 223,
140, 161, 137,  13, 191, 230,  66, 104,  65, 153,  45,  15, 176,  84, 187,  22,
};

word8 Si[256] = {
 82,   9, 106, 213,  48,  54, 165,  56, 191,  64, 163, 158, 129, 243, 215, 251,
124, 227,  57, 130, 155,  47, 255, 135,  52, 142,  67,  68, 196, 222, 233, 203,
 84, 123, 148,  50, 166, 194,  35,  61, 238,  76, 149,  11,  66, 250, 195,  78,
  8,  46, 161, 102,  40, 217,  36, 178, 118,  91, 162,  73, 109, 139, 209,  37,
114, 248, 246, 100, 134, 104, 152,  22, 212, 164,  92, 204,  93, 101, 182, 146,
108, 112,  72,  80, 253, 237, 185, 218,  94,  21,  70,  87, 167, 141, 157, 132,
144, 216, 171,   0, 140, 188, 211,  10, 247, 228,  88,   5, 184, 179,  69,   6,
208,  44,  30, 143, 202,  63,  15,   2, 193, 175, 189,   3,   1,  19, 138, 107,
 58, 145,  17,  65,  79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115,
150, 172, 116,  34, 231, 173,  53, 133, 226, 249,  55, 232,  28, 117, 223, 110,
 71, 241,  26, 113,  29,  41, 197, 137, 111, 183,  98,  14, 170,  24, 190,  27,
252,  86,  62,  75, 198, 210, 121,  32, 154, 219, 192, 254, 120, 205,  90, 244,
 31, 221, 168,  51, 136,   7, 199,  49, 177,  18,  16,  89,  39, 128, 236,  95,
 96,  81, 127, 169,  25, 181,  74,  13,  45, 229, 122, 159, 147, 201, 156, 239,
160, 224,  59,  77, 174,  42, 245, 176, 200, 235, 187,  60, 131,  83, 153,  97,
 23,  43,   4, 126, 186, 119, 214,  38, 225, 105,  20,  99,  85,  33,  12, 125,
};


word32 RC[30] = {
  0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
  0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e,
  0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
  0xfa, 0xef, 0xc5};


#define MAXBC       8
#define MAXKC       8
#define MAXROUNDS   14

static word8 shifts[5][4] = {

       0, 1, 2, 3,
       0, 1, 2, 3,
       0, 1, 2, 3,
       0, 1, 2, 4,
       0, 1, 3, 4};

static int numrounds[5][5] = {
       10, 11, 12, 13, 14,
       11, 11, 12, 13, 14,
       12, 12, 12, 13, 14,
       13, 13, 13, 13, 14,
       14, 14, 14, 14, 14};

int BC, KC, ROUNDS;

word8 mul(word8 a, word8 b) {
       /* multiply two elements of GF(256)
        * required for MixColumns and InvMixColumns
        */

       if (a && b)
```

```
        return Alogtable[(Logtable[a] + Logtable[b])%255];
        else return 0;
}


void AddRoundKey(word8 a[4][MAXBC], word8 rk[4][MAXBC]) {
        /* XOR corresponding text input and round key input bytes
         */
        int i;
        for (i=0; i<4; i++)
        {
                a[i][0] ^= rk[i][0];
                a[i][1] ^= rk[i][1];
                a[i][2] ^= rk[i][2];
                a[i][3] ^= rk[i][3];
        }
}

void SubBytes(word8 a[4][MAXBC], word8 box[256]) {
        /* Replace every byte of the input by the byte at that place
         * in the non-linear S-box
         */
        int i;

        for (i=0; i<4; i++)
        {               a[i][0] = box[a[i][0]];
                        a[i][1] = box[a[i][1]];
                        a[i][2] = box[a[i][2]];
                        a[i][3] = box[a[i][3]];
    }
}

void ShiftRows(word8 a[4][MAXBC]) {
        /* Row 0 remains unchanged
         * The other three rows are shifted a variable amount
         */
        word8 tmp[MAXBC];
        int i;
                for (i=1; i<4; i++) {

                        tmp[0] = a[i][(0 + i) % BC];
                        tmp[1] = a[i][(1 + i) % BC];
                        tmp[2] = a[i][(2 + i) % BC];
                        tmp[3] = a[i][(3 + i) % BC];

                        a[i][0] = tmp[0];
                        a[i][1] = tmp[1];
                        a[i][2] = tmp[2];
                        a[i][3] = tmp[3];
                }

}

void InvShiftRows(word8 a[4][MAXBC]) {
        /* Row 0 remains unchanged
         * The other three rows are shifted a variable amount
         */

        word8 tmp[MAXBC];
        int i;

                for (i=1; i<4; i++) {

                        tmp[0] = a[i][(0 + 4 - i) % BC];
```

```
            tmp[1] = a[i][(1 + 4 - i) % BC];

tmp[2] = a[i][(2 + 4 - i) % BC];
                  tmp[3] = a[i][(3 + 4 - i) % BC];

                  a[i][0] = tmp[0];
                  a[i][1] = tmp[1];
                  a[i][2] = tmp[2];
                  a[i][3] = tmp[3];
            }
}

void MixColumns(word8 a[4][MAXBC]) {
      /* Mix the four bytes of every column in a linear way
       */
      word8 b[4];
      word8 temp1[4], temp2[4];
      int  j;

      for(j=0; j< BC; j++)
      {

            b[0]    = mul(2,a[0][j]);
            temp1[0]= mul(3,a[1][j]);
            temp2[0]=           a[2][j] ^ a[3][j];
            temp2[0]^= temp1[0];
            b[0]    ^= temp2[0];


            b[1]    =       a[0][j] ^ a[3][j];
            temp1[1]= mul(2,a[1][j]);
            temp2[1]= mul(3,a[2][j]);
            temp2[1]^= temp1[1];
            b[1]   ^= temp2[1];


            b[2]    =       a[0][j]    ^ a[1][j];
            temp1[2]= mul(2,a[2][j]);
            temp2[2]= mul(3,a[3][j]);
            temp2[2]^= temp1[2];
            b[2]   ^= temp2[2];

            b[3]    = mul(3,a[0][j]);
            temp1[3]=       a[1][j]    ^ a[2][j];
            temp2[3]= mul(2,a[3][j]);
            temp2[3]^= temp1[3];
            b[3]   ^= temp2[3];

            a[0][j] = b[0]   ;
            a[1][j] = b[1]   ;
            a[2][j] = b[2]   ;
            a[3][j] = b[3]   ;
      }
}

void InvMixColumns(word8 a[4][MAXBC]) {
      /* Mix the four bytes of every column in a linear way
       * This is the opposite operation of MixColumns
       */
```

```
        word8 b[4];//[MAXBC];
        word8 temp0[3], temp1[3], temp2[3], temp3[3];
        int  j;

        for(j=0; j< BC; j++)
        {

                b[0]    = mul(0xe,a[0][j]);
                temp0[0]= mul(0xb,a[1][j]);
                temp0[1]= mul(0xd,a[2][j]);
                temp0[2]= mul(0x9,a[3][j]);
                temp0[1]^=temp0[0];
                b[0]    ^=temp0[2];
                b[0]    ^=temp0[1];

                b[1]    = mul(0x9,a[0][j]);
                temp1[0]= mul(0xe,a[1][j]);
                temp1[1]= mul(0xb,a[2][j]);
                temp1[2]= mul(0xd,a[3][j]);
                temp1[1]^=temp1[0];
                b[1]    ^=temp1[2];
                b[1]    ^=temp1[1];

                b[2]    = mul(0xd,a[0][j]);
                temp2[0]= mul(0x9,a[1][j]);
                temp2[1]= mul(0xe,a[2][j]);
                temp2[2]= mul(0xb,a[3][j]);
                temp2[1]^=temp2[0];
                b[2]    ^=temp2[2];
                b[2]    ^=temp2[1];

                b[3]    = mul(0xb,a[0][j]);
                temp3[0]= mul(0xd,a[1][j]);
                temp3[1]= mul(0x9,a[2][j]);
                temp3[2]= mul(0xe,a[3][j]);
                temp3[1]^=temp3[0];
                b[3]    ^=temp3[2];
                b[3]    ^=temp3[1];

                a[0][j] = b[0]   ;
                a[1][j] = b[1]   ;
                a[2][j] = b[2]   ;
                a[3][j] = b[3]   ;

        }

}


int KeyExpansion (word8 k[4][MAXKC],
                                word8 W[MAXROUNDS+1][4][MAXBC]) {
        /* Calculate the required round keys
         */
        int i, j, t, RCpointer = 1;
        word8 tk[4][MAXKC];

        for(j=0; j< KC; j++)
                for (i=0; i<4; i++)
                        tk[i][j] = k[i][j];
                t=0;
                /* Copy values into round key array */
```

```
for (j=0; (j<KC) && (t <(ROUNDS+1) * BC ); j++, t++)
                        for ( i=0; i<4; i++) W[t / BC][i][t % BC] = tk[i][j];

                        while (t < (ROUNDS + 1)*BC) {
                               /* while not enough round key material calculated,
                                * calculate new values
                                */

                               for(i=0; i<4; i++)
                                      tk[i][0] ^= S[tk[(i+1)%4][KC-1]];
                               tk[0][0] ^= RC[RCpointer++];


                               if (KC <= 6 )
                                      for (j=1; j < KC; j++)
                                             for(i=0; i<4; i++)
                                                    tk[i][j] ^= tk[i][j-1];
                               else {
                                      for (j=1; j < 4; j++)
                                             for(i=0; i<4; i++)
                                                    tk[i][j] ^= tk[i][j-1];
                                      for(i=0; i<4; i++) tk[i][4] ^= S[tk[i][3]];
                                      for (j=5; j < KC; j++)
                                             for(i=0; i<4; i++)
                                                    tk[i][j] ^= tk[i][j-1];
                        }
                        /* copy values into round key array */
                        for (j=0; (j < KC) && (t<(ROUNDS+1)*BC); j++, t++)
                               for(i=0; i<4; i++) W[t/BC][i][t%BC] = tk[i][j];
                        }

                        return 0;
}


int Encrypt (word8 a[4][MAXBC], word8 rk[MAXROUNDS+1][4][MAXBC])
{
       /* Encryption of one blockl
        */

       int r;

       /* begin with a key addition
        */

       AddRoundKey(a, rk[0]);

       /* ROUNDS-1 ordinary rounds
        */

       for ( r=1; r < ROUNDS; r++) {
              SubBytes (a,S);
              ShiftRows(a);
              MixColumns(a);
              AddRoundKey(a,rk[r]);
       }

       /* Last round is special; there is no MixColumns
        */
       SubBytes (a,S);
       ShiftRows(a);
       AddRoundKey(a,rk[ROUNDS]);
```

```
	return 0;
}

int Decrypt (word8 a[4][MAXBC], word8 rk[MAXROUNDS+1][4][MAXBC])
{

	int r;

	/* To decrypt:
	 *  apply the inverse operations of the encrypt routine,
	 *  in opposite order
	 *
	 * - AddRoundKey is equal to its inverse)
	 * - the inverse of SubBytes with table S is

	 *              SubBytes with the inverse table of S)
	 * - the inverse of Shiftrows is Shiftrows over
	 *              a suitable distance)
	 */

	/* First the special round:
	 *   without InvMixColumns
	 *   with extra AddRoundKey
	 */

	AddRoundKey(a, rk[ROUNDS]);
	SubBytes(a, Si);
	InvShiftRows(a);

	// ROUNDS-1 ordinary rounds


	for ( r=ROUNDS-1; r > 0; r--) {
		AddRoundKey(a,rk[r]);
		InvMixColumns(a);
		SubBytes (a,Si);
		InvShiftRows(a);
	}

	// End with the extra key addition

	AddRoundKey(a,rk[0]);

	return 0;
}


int main()
{
	int i, j;
	word8 a[4][MAXBC], rk[MAXROUNDS+1][4][MAXBC], sk[4][MAXKC];

	//		AES block length allowed is 128
	//			BC=4 for 128-bit plaintext

	BC = 4;

	/*		The KC value is changed to set the key length.
				KC=4 for 128-bit key,
				KC=6 for 192-bit key
				KC=8 for 256-bit key
	*/
```

```
	KC = 4;

	{
		ROUNDS = numrounds[KC-4][BC-4];

	//		For the real system plaintext and
	//		key is given by an external application

		for ( j=0; j<BC; j++)
			for ( i=0; i < 4; i++)
				a[i][j] = 0; // plaintext
		for ( j=0; j<KC; j++)
			for ( i=0; i < 4; i++)
				sk[i][j] = 0; // key

		KeyExpansion (sk, rk);

	//		Encrpyption is performed twice and
	//		decryption is performed twice

		Encrypt(a, rk);
		printf("Encrypt 1:");

		for(j=0; j< BC; j ++)
			for ( i=0; i<4; i++)
				printf("%02X", a[i][j]);

		printf("\n");

		Encrypt(a, rk);
		printf("Encrypt 2:");

		for(j=0; j< BC; j ++)
			for ( i=0; i<4; i++)
				printf("%02X", a[i][j]);

		printf("\n");printf("\n");


		Decrypt(a, rk);
		printf("Decrypt 2:");

		for(j=0; j< BC; j ++)
			for ( i=0; i<4; i++)
				printf("%02X", a[i][j]);
		printf("\n");

		Decrypt(a, rk);
		printf("Decrypt 1:");

		for(j=0; j< BC; j ++)
			for ( i=0; i<4; i++)
				printf("%02X", a[i][j]);
		printf("\n");printf("\n");

	}


	printf("\n\n\t\t End of the program\n\n");
		return 0;
}
```

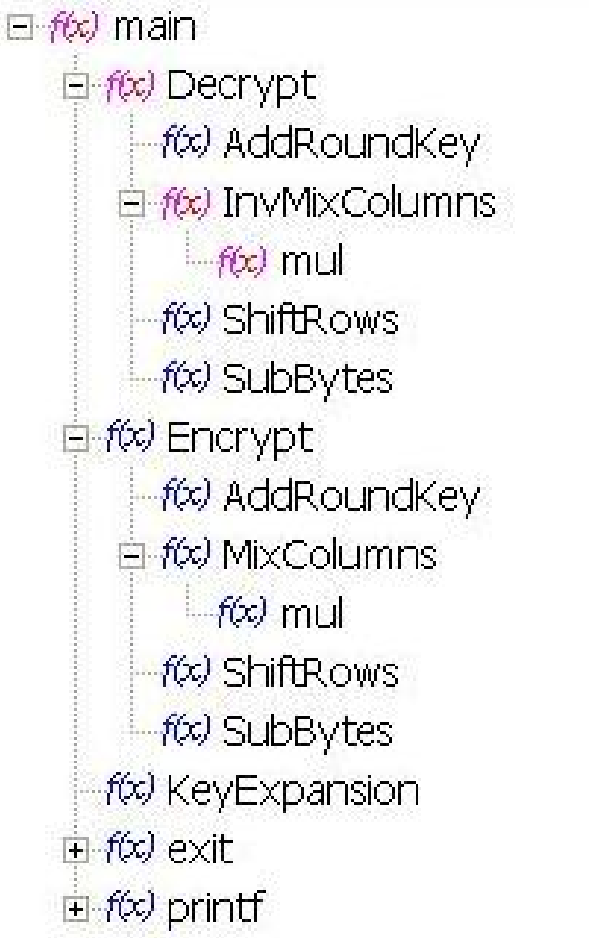

Figure A.1: Function call tree

INDEX