Implementing Security Protocols in Wireless Embedded System: A Power Performance Analysis

By

Agrawal Jayantilal Govind
(05MCE001)

Department of Computer Science and Engineering

Institute of Technology

Nirma University of Science and Technology

Ahmedabad 382481

May 2007

Implementing Security Protocols in Wireless Embedded System: A Power Performance Analysis

Major Project

submitted in partial fulfillment of the requirements

for the degree of

Master of Technology in Computer Science and Engineering

By

Agrawal Jayantilal Govind
(05MCE001)

Guide
Prof. (Dr.) S. N. Pradhan

Department of Computer Science and Engineering
Institute of Technology
Nirma University of Science and Technology
Ahmedabad 382481

May 2007

This is to certify that Dissertation entitled

Implementing Security Protocols in Wireless Embedded System: A Power Performance Analysis

Submitted by

Agrawal Jayantilal G.

has been accepted toward fulfillment of the requirement
for the degree of
Master of Technology in Computer Science & Engineering

Prof. (Dr.) S. N. Pradhan
Professor In Charge

Prof. D. J. Patel
Head of The Department

Prof. A. B. Patel
Director, Institute of Technology

CERTIFICATE

This is to certify that the Major Project entitled "Implementing Security Protocols in Wireless Embedded System" submitted by Mr. Jayantilal Govind Agrawal (05MCE001), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. (Dr.) S. N. Pradhan
Guide,
Professor,
Department of Computer Science and Engineering,
Institute of Technology,
Nirma University,
Ahmedabad.

Date: / /

ACKNOWLEDGMENT

Windy ocean and high tides can wreck any ship unless guided by a good captain. Similarly, any project can head in wrong direction by problems faced by the students. To successfully carry out any research project, the expertise, visionary goals setting, encouraging help, enthusiastic interest and more importantly discipline of project guide is invaluable.

I am therefore, grateful to my project guide, **Prof. (Dr.) S. N. Pradhan**, (P.G. Coordinator, CSE) Department of Computer Science and Engg, Nirma University of Science & Technology, Ahmedabad for his constant guidance, support and encouragement which I received so spontaneously throughout the Major Project work. Without his valuable support, suggestions and his vision I could not have envisaged the success of this project.

I would like to thank **Prof. A. B. Patel**, Director, Institute of Technology, Nirma University of Science & Technology, Ahmedabad for providing me the facilities in the Nirma campus.

I am thankful to all the faculty of Computer Science and Engg department, Nirma University of Science & Technology, Ahmedabad for providing suggestions with crucial feedback that influenced me to complete this work.

The blessings of God, my Guruji and my family members make the way for completion of major project. I am very much grateful to them.

Last but not the least, I am equally thankful to all my friends for everything.

Jayant Agrawal
(05MCE001)

ABSTRACT

As embedded devices are increasingly integrated into personal and commercial infrastructures, security becomes a paramount issue. The design of security for embedded systems differs from traditional security design because these systems are resource constrained in their capacities and easily accessible to adversaries at the physical layer.

Implementing security in wireless embedded devices arises new challenges due to the unique characteristics of battery powered embedded systems. The work is focused on an important constraint of such devices - battery life and examines how it is impacted by the use of security protocols. Software power consumption minimization is becoming more important and a very relevant issue in the design of embedded systems, in particular those dedicated to mobile devices. This motivates the need for minimizing power consumption from the point of view of software rather than at circuit and gate level, which is cumbersome.

This dissertation aims at reviewing state of the art of different cryptography protocol implementations and optimizations for reducing the power and energy consumption. It also restricts the size of lookup tables and imposes constraints on the code size where run-time memory and program ROM are scarce resources.

The protocols are implemented in C language. Using the gcc compiler with Sim–Power Analyzer as the simulation kernel protocols have been evaluated on the ARM cores. Different coding methods have been identified aiming at reducing the power consumption for the ARM processors.

The performance improvement of cryptography algorithms is demonstrated on different ARM processors such as Intel StrongARM -1110, ARM7 and ARM9TDMI, where Intel's StrongARM-1110 found to be low energy processors i.e. it consumes low energy as compared to other processors. Also AES found to be the most energy efficient symmetric algorithm and RSA among the asymmetric implemented algorithms. So, AES in combination with RSA on the StrongARM-1110 processor can be the best choice for implementing security protocol.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
CPI	Cycle Per Instructions
DES	Data Encryption Standard
FIPS	Federal Information Processing Standard
IDEA	International Data Encryption Algorithm
IP	Internet Protocol
IPSec	IP Security
LAN	Local Area Network
NIST	National Institute of Standards and Technology
RSA	Rivest-Shamir-Adleman
SET	Secure Electronic Transaction
SSL	Secure Sockets Layer
TLS	Transport Layer Security
WEP	Wired Equivalent Privacy
WTLS	Wireless Transport Layer Security

1 INTRODUCTION

1.1 GENERAL

As our lives are increasingly played out in the digital world, a variety of "sensitive" data is captured, stored, manipulated, or communicated by electronic systems. Consequently, a large and increasing number of electronic systems, and the ICs they contain, need to deal with security in one form or the other - including PCs, PDAs, wireless handsets, and smart cards, network equipment such as routers, gateways, firewalls, storage and web servers, *etc*. where security is rated as a primary concern in the adoption of new services and applications. Networked embedded systems, which account for a large portion of the electronics and semiconductor markets, makes the communication channel especially vulnerable and the need for security even more obvious.

This merging of communications and computation functionality requires data processing in real time, and embedded systems have shown to be good solutions for many applications. Examples of such applications are – PDAs, cell phones, networked sensors, and smart cards, and some electronic commerce devices, to name just a few. Security concerns in such systems range from user identification, to secure information storage, secure software execution, and secure communications. Most battery-powered systems contain wireless communication capabilities for untethered operation, introducing new security concerns due to the public nature of the physical communication medium or channel. Since many of these applications need security functionality, this dissertation focuses on cryptographic algorithms and their implementation on embedded systems.

In addition to embedded devices, the explosive growth of digital communications also brings additional security challenges. Millions of electronic transactions are completed each day, and the rapid growth of e-Commerce has made security a vital issue for many consumers. Valuable business opportunities are realized over the Internet and megabytes of sensitive data is transferred and moved over insecure communication channels around the world. Thus, it is imperative for the success of modern businesses that all these transactions be realized in a secure

manner. Secure communication across wired and wireless networks is typically achieved by employing security protocols at various layers of the network protocol stack (*e.g.*, WEP at the link layer, IPSec at the network layer, TLS/SSL and WTLS at the transport later, SET at the application layer, *etc.*). The building blocks of a security protocol are cryptographic algorithms, which are selected based on the security objectives that are to be achieved by the protocol. They include asymmetric and symmetric encryption algorithms, which are used to provide authentication and privacy, as well as hash or message digest algorithms that are used to provide message integrity.

Security protocols and the cryptographic algorithms they contain, address security considerations from a functional perspective, many embedded systems are constrained by the environments they operate in, and by the resources they possess. For such systems, there are several factors that are moving security considerations from a function centric perspective into a system architecture (hardware/software) design issue. For example,

- An ever increasing range of attack techniques for breaking security such as software, physical and side-channel attacks require that the embedded system be secure even when it can be logically or physically accessed by malicious entities. Resistance to such attacks can be ensured only if built into the system architecture and implementation.
- The processing capabilities of many embedded systems are easily overwhelmed by the computational demands of security processing, leading to undesirable tradeoffs between security and cost, or security and performance.
- Battery-driven systems and small form-factor devices such as PDA's, cell phones and networked sensors often operate under stringent resource constraints (limited battery, storage and computation capacities). These constraints only worsen when the device is subject to the demands of security.

Many of the new security protocols decouple the choice of cryptographic algorithm from the design of the protocol. Users of the protocol negotiate on the choice of algorithm to use for a particular secure session. The new devices to support these applications, then must not only support a single cryptographic

algorithm and protocol, but also must be “algorithm agile", that is, able to select from a variety of algorithms [1]. For example, IPSec (the security standard for the Internet) allows choosing out of a list of different symmetric as well asymmetric ciphers. Some of the symmetric-key algorithms are: DES, 3DES, AES, IDEA, RC4, and so on. Thus, software-based systems would seem to be a better fit because of their flexibility. However, the security engineer is faced with a difficult choice. Should he/she choose in favor of performance and security, and pay the price of inflexibility and higher costs? Or should he/she favor flexibility instead? Fortunately, many embedded processors combine the flexibility of software on general-purpose computers with the near-hardware speed and better physical security than general-purpose computers.

1.2 MOTIVATION

A self-configuring wireless sensor networks consist of hundreds or thousands of small, battery-driven nodes with a wireless modem. Security for these sensor networks is not easy since these sensors have limited processing power, storage, bandwidth, and energy. The network lifetime is an important concern: as nodes run out of power, the connectivity decreases and the network can finally be partitioned and become dysfunctional. Also, the sensor network should not leak sensor readings to neighboring networks. In many applications (e.g., key distribution) nodes communicate highly sensitive data. The standard approach for keeping sensitive data secret is to encrypt the data with a secret key that only intended receivers possess, hence achieving confidentiality.

The increasing popularity of power constrained mobile computers and embedded computing applications such as wireless sensor network drives the need for analyzing and optimizing power in all the components of a system. Software constitutes a major component of today’s systems, and its role is projected to grow even further. Thus, an ever-increasing portion of the functionality of today’s systems is in the form of instructions, as opposed to gates. This motivates the need for analyzing power consumption from the point of view of software.

Consider the following example system which motivate the need to address energy consumption issues in security protocols: a sensor node, using a Motorola "DragonBall" MC68328 processor and operating at a data rate of 10Kbps, consumes 21.5*mJ* and 14.3*mJ*, for transmitting and receiving 1024 bits of data, respectively [2]. In secure mode, when RSA encryption is used as part of a security protocol, encrypting 1024 bits of data on the node was observed to consume 42*mJ* of energy. Thus, given a typical battery capacity of 26*KJ* in sensor nodes, it can be shown that with encryption on, the battery runs out more than twice as fast as when there is no encryption. This example motivates us to investigate techniques to facilitate energy-efficient execution of security protocols. This objective can be achieved in multiple ways. For example: by making the execution of underlying cryptographic algorithms efficient through software techniques, we can improve the performance and energy requirements of security protocols. Usually, there is an overhead, in the form of more complex software, associated with these techniques.

Software constitutes a major component of systems where power is a constraint. Its presence is very visible in a mobile computer, in the form of the system software and application programs running on the main CPU. But software also plays an even greater role in general digital applications, since an ever-growing fraction of these applications are now being implemented as embedded systems. Embedded systems are characterized by the fact that their functionality is divided between hardware and a software component. The software component usually consists of application-specific software running on a dedicated processor, while the hardware component usually consists of application-specific circuits. In light of the above, there is a clear need for considering the power consumption in systems from the point of view of software. Software impacts the system power consumption at various levels of the design. At the highest level, this is determined by the way functionality is partitioned between hardware and software. The choice of the algorithm and other higher-level decisions about the design of the software component can affect system power consumption in a big way. The design of system software, the actual application source code, and the process of translation into machine instructions - all of these determine the power cost of the software component.

The challenges of energy-efficient secure communications can be better addressed if energy requirements and bottlenecks are well understood. In this work, a detailed analysis of the energy requirements of various cryptographic primitives is performed, with the intention of using this data as a foundation for devising energy-efficient security protocols. Cryptographic algorithms are known to have significant computational requirements, and studies have indicated that they stretch the processor capabilities available in many embedded systems. Comprehensive energy analysis of cryptographic protocols, such as performed in this work, will facilitate to identify energy bottlenecks and development of energy efficient security mechanisms. The ability to evaluate software in terms of power consumption makes it feasible to search for low power implementations of given programs. In addition, it can guide the development of general tools and techniques for low power software.

1.3 SCOPE OF WORK

This thesis presents a comprehensive energy measurement and analysis of the most common cryptographic algorithms used. The energy analysis in this study is performed by executing cryptographic algorithms using a power estimation and simulation tool – Sim-Panalyzer, which measures the power consumed, and calculates the energy consumed during the execution of cryptographic algorithms. Four common encryption schemes were chosen for the study ranging from symmetric block ciphers (DES, 3DES, AES) to asymmetric cipher (RSA). This choice was driven by the objective to assess encryption schemes with different overheads that provide increasing levels of protection. Most significantly, the algorithmic choice is motivated by the constraints of embedded architectures, different key and block length, execution time as well as the energy consumption, which would severely affect the lifetime of mobile devices. Measurements were obtained for three different ARM processors. The analysis takes into account features of architectures, such as processor frequency, ISA characteristics - RISC in all the processors, number of pipeline stages, and the impact of memory hierarchies for architectures with caches. Based on the analysis, energy-efficient implementations of security protocols are discussed.

1.4 ORGANIZATION OF THE PROJECT

This thesis is organized as follows:

- Chapter 2 provides the brief introduction about cryptography, previous related work for source code optimization and energy consumption analysis, the security requirements of embedded system, the different software attacks on embedded system and challenges in design of embedded system. Also, what are the components that lead to software power consumption are explained in brief.
- Chapter 3 explains the working of DES algorithm and introduces different techniques for optimizing the different permutations and substitution operations. It also gives brief idea about 3DES.
- Chapter 4 explains the working of AES algorithm and its various function. The optimization technique introduced for the MixColumn operation is explained in detail.
- Chapter 5 explains the RSA implementation in brief and the two algorithms used for efficient execution along with the efficient implementation of primality test.
- Chapter 6 explains the different optimization methodology used for writing efficient C programs for ARM.
- Chapter 7 explains Sim-Panalyzer tool used for the power and energy estimation of the protocols. It also explains how to install and run the simulator and how to calculate the power.
- Chapter 8 presents the power and energy estimation results and the outcomes are discussed.
- Chapter 9 concludes this thesis with a summary, and provides possible directions for relevant future research.

2 LITERATURE SURVEY

2.1 GENERAL

Cryptography involves the study of mathematical techniques that allow the practitioner to achieve or provide the following objectives or services [1]:

- ***Confidentiality*** is a service used to keep the content of information accessible to only those authorized to have it. This service includes both protections of all user data transmitted between two points over a period of time as well as protection of traffic flow from analysis.
- ***Integrity*** is a service that requires that computer system assets and transmitted information be capable of modification only by authorized users. Modification includes writing, changing, changing the status, deleting, creating, and the delaying or replaying of transmitted messages. It is important to point out that integrity relates to active attacks and therefore, it is concerned with detection rather than prevention. Moreover, integrity can be provided with or without recovery, the first option being the more attractive alternative.
- ***Authentication*** is a service that is concerned with assuring that the origin of a message is correctly identified. That is, information delivered over a channel should be authenticated as to the origin, date of origin, data content, time sent, etc. For these reasons this service is subdivided into two major classes: entity authentication and data origin authentication.
- ***Non-repudiation*** is a service, which prevents both the sender and the receiver of a transmission from denying previous commitments or actions.

In data and telecommunications, cryptography is necessary when communicating over any un-trusted medium, which includes just about any network, particularly the Internet.

Cryptography, then, not only protects data from theft or alteration, but can also be used for user authentication. There are, in general, three types of cryptographic schemes typically used to accomplish these goals: secret key (or symmetric) cryptography, public-key (or asymmetric) cryptography, and hash

functions. In all cases, the initial unencrypted data is referred to as *plaintext*. It is encrypted into *ciphertext*, which will in turn be decrypted into usable plaintext.

2.2 RELATED WORK

In this work, a detailed analysis of the energy requirements of various cryptographic primitives are performed, with the intention of using this data as a foundation for devising energy-efficient security protocols. The results of the experiments are used to suggest ways for making the execution of the cryptographic algorithms energy-efficient.

Security protocols and cryptographic algorithms are known to have significant computational requirements, and studies have indicated that they stretch the processor capabilities available in many embedded systems [3, 4, 5, 6]. While researchers have quantified and addressed the performance overhead of security, the energy implications are relatively less understood. Nevertheless, researchers have recently proposed interesting approaches to the design of lightweight security protocols. Low-power key management protocols have been devised for sensor nodes by analyzing the impact of security algorithms on the energy consumption of sensor nodes [2]. The work in [7] evaluated the energy consumption of selected key exchange protocols on a WINS sensor node, and proposed energy efficient ways for exchanging cryptographic keys, while custom protocols for low-power mutual authentication were proposed in [8, 9]. Energy tradeoffs in the network protocol and key management design space of sensor nodes were explored in [10]. Techniques to minimize the energy consumed by secure wireless sessions have also been proposed in [11]. The comprehensive energy analysis of cryptographic algorithms, such as the one performed in this work will facilitate identification of energy bottlenecks and development of energy efficient security mechanisms.

2.3 SECURITY REQUIREMENTS OF EMBEDDED SYSTEMS

Figure 2.1 lists the typical security requirements seen across a wide range of embedded systems, which are described as follows:

Figure 2.1: Common Security Requirements of ES

- *User identification* refers to the process of validating users before allowing them to use the system.
- *Secure network access* provides a network connection or service access only if the device is authorized.
- *Secure communications* functions include authenticating communicating peers, ensuring confidentiality and integrity of communicated data, preventing repudiation of a communication transaction, and protecting the identity of communicating entities.
- *Secure storage* mandates confidentiality and integrity of sensitive information stored in the system.
- *Content security* enforces the usage restrictions of the digital content stored or accessed by the system.
- *Availability* ensures that the system can perform its intended function and service legitimate users at all times, without being disrupted by denial of service attacks [12].

2.4 EMBEDDED SOFTWARE ATTACKS AND COUNTERMEASURES

Software in embedded systems is a major source of security vulnerabilities. Three factors, which we call the *Trinity of Trouble* —complexity, extensibility and connectivity, conspire to make managing security risks in software a major challenge [13].

- **Complexity:** Software is complicated, and will become even more complicated in the near future. More lines of code increase the likelihood of bugs and security vulnerabilities. As embedded systems converge with

the Internet and more code is added, embedded system software is clearly becoming more complex. The complexity problem is exacerbated by the use of unsafe programming languages (e.g., C or C++) that do not protect against simple kinds of attacks, such as buffer overflows. For reasons of efficiency, C and C++ are very popular languages for embedded systems. In theory, we could analyze and prove that a small program is free of problems, but this task is impossible for programs of realistic complexity today.

- **Extensibility:** Modern software systems, such as Java and .NET, are built to be extended. An extensible host accepts updates or extensions (mobile code) to incrementally evolve system functionality. Today's operating systems support extensibility through dynamically loadable device drivers and modules. Advanced embedded systems are designed to be extensible (e.g., J2ME, Java Card). Unfortunately, the very nature of extensible systems makes it hard to prevent software vulnerabilities from slipping in as an unwanted extension.
- **Connectivity:** More and more embedded systems are being connected to the Internet. The high degree of connectivity makes it possible for small failures to propagate and cause massive security breaches. Embedded systems with Internet connectivity will only make this problem grow. An attacker no longer needs physical access to a system to launch automated attacks to exploit vulnerable software. The ubiquity of networking means that there are more attacks, more embedded software systems to attack, and greater risks from poor software security practices.

2.5 DESIGN CHALLENGES

Designers of a large and increasing number of embedded systems need to support various security solutions in order to deal with one or more of the security requirements described earlier. These requirements present significant bottlenecks during the embedded system design process, which are briefly described below:

- ***Processing Gap*:** Existing embedded system architectures are not capable of keeping up with the computational demands of security processing, due

to increasing data rates and complexity of security protocols. These shortcomings are most felt in systems that need to process very high data rates or a large number of transactions (e.g., network routers, firewalls, and web servers), and in systems with modest processing and memory resources (e.g., PDAs, wireless handsets, and smartcards).

- ***Battery Gap*:** The energy consumption overheads of supporting security on battery-constrained embedded systems are very high. Slow growth rates in battery capacities (5–8% per year) are easily outpaced by the increasing energy requirements of security processing, leading to a battery gap. Various studies show that the widening battery gap would require designers to make energy-aware design choices (such as optimized security protocols, custom security hardware, and so on) for security.
- ***Flexibility*:** An embedded system is often required to execute multiple and diverse security protocols and standards in order to support, (i) multiple security objectives (e.g., secure communications, Digital Rights Management (DRM), and so on), (ii) interoperability in different environments (e.g., a handset that needs to work in both 3G cellular and wireless LAN environments), and (iii) security processing in different layers of the network protocol stack (e.g., a wireless LAN enabled PDA that needs to connect to a virtual private network, and support secure web browsing may need to execute WEP, IPSec, and SSL). Furthermore, with security protocols being constantly targeted by hackers, it is not surprising that they keep continuously evolving. It is, therefore, desirable to allow the security architecture to be flexible (programmable) enough to adapt easily to changing requirements. However, flexibility may also make it more difficult to gain assurance of a design's security.
- ***Tamper Resistance*:** Attacks due to malicious software such as viruses and Trojan horses are the most common threats to any embedded system that is capable of executing downloaded applications. These attacks can exploit vulnerabilities in the operating system (OS) or application software, procure access to system internals, and disrupt its normal functioning. Because these attacks manipulate sensitive data or processes (integrity attacks), disclose confidential information (privacy attacks), and/or deny access to system resources (availability attacks), it is necessary to develop and deploy various HW/SW countermeasures against these attacks. In

many embedded systems such as smartcards, new and sophisticated attack techniques, such as bus probing, timing analysis, fault induction, power analysis, electromagnetic analysis, and so on, have been demonstrated to be successful in easily breaking their security. Tamper resistance measures must, therefore, secure the system implementation when it is subject to various physical and side-channel attacks.

- ***Assurance Gap*:** It is well known that truly reliable systems are much more difficult to build than those that merely work most of the time. Reliable systems must be able to handle the wide range of situations that may occur by chance. Secure systems face an even greater challenge: they must continue to operate reliably despite attacks from intelligent adversaries who intentionally seek out undesirable failure modes. As systems become more complicated, there are inevitably more possible failure modes that need to be addressed. Increases in embedded system complexity are making it more and more difficult for embedded system designers to be confident that they have not overlooked a serious weakness.
- ***Cost*:** One of the fundamental factors that influence the security architecture of an embedded system is cost. To understand the implications of a security related design choice on the overall system cost, consider the decision of incorporating physical security mechanisms in a single-chip cryptographic module. The Federal Information Processing Standard (FIPS 140-2) [FIPS] specifies four increasing levels of physical (as well as other) security requirements that can be satisfied by a secure system. Security Level 1 requires minimum physical protection; Level 2 requires the addition of tamper-evident mechanisms such as a seal or enclosure, while Level 3 specifies stronger detection and response mechanisms. Finally, Level 4 mandates environmental failure protection and testing (EFP and EFT), as well as highly rigorous design processes. Thus, we can choose to provide increasing levels of security using increasingly advanced measures, albeit at higher system costs, design effort, and design time. It is the designer's responsibility to balance the security requirements of an embedded system against the cost of implementing the corresponding security measures [12].

2.6 SOFTWARE POWER CONSUMPTION

The overall power dissipation of an embedded system does not only originate from the application - specific hardware, but also from the CPU, the memory and the address and data buses when an embedded application is running on the platform's microprocessor, microcontroller or digital signal processor. This above power dissipation is referred to as software power dissipation. Obviously, the power is actually dissipated in the processor's hardware, but it is as a consequence of executing an application program. There are a number of sources of power dissipation influenced by software and contributing to the overall power dissipation of the system, which are explained below.

2.6.1 Bus Power

Busses in an embedded system consist of unidirectional address bus lines and instruction bus lines (opcodes to be executed) and bi-directional data bus lines. With these groups of interconnecting lines, the communication of the CPU with the memory, I/O circuits and peripheral modules is established. Each of the above lines can be modeled as an RC transmission line, where R and C are the line's resistance and capacitance, respectively. Activation of a line prompts for the charging or discharging of the capacitive load, depending on the previous value that this line had. For example, a transition in an 8-bit data bus between words 00101011 and 11100111 implies charging of 3 lines and discharging of 1 line. Usually, bus charging and discharging of I/O lines can occur up to 80% of the software execution time.

2.6.2 Memory Power

Power dissipated by memory read and write accesses is usually one of the dominating components (ranging from 10% - 25%) of the total software power dissipation for mobile devices and portable computers [14]. In the case of DSP applications, where a significant amount of data is processed, this contribution can be substantially higher. Memory power dissipation has a number of components, namely power dissipated in the cell array, in the decode logic and sense amplifier as well as power dissipated due to charging and discharging of

the address or data lines capacitances. The type of access is also significant in how much it contributes to the overall power dissipation. In the ARM microprocessor, which is considered in our system, CPU cycles are divided in S-cycles, N-cycles, and I-cycles.

The S-cycles refer to sequential memory accesses, where the next word is returned from the same buffer, dissipating a relatively small amount of power. Power dissipation in the address lines during sequential memory accesses is also small, due to the fact that the address word changes only in one bit. If the last memory location of a page is to be accessed though, more power has to be consumed in order to access the next page.

The N-cycles refer to non-sequential memory accesses. In this kind of access, more power is dissipated relative to sequential accesses because consecutive address words are irrelevant, causing large activity in the address bus. In N-cycles usually different pages have to be accessed, contributing to more power dissipation, as explained above.

Finally, the I-cycles refer to cycles that no memory access is involved, so no power is dissipated in the memory system. One more significant contributor to the power dissipated in the memory is the memory access patterns, affecting mainly the cache hits and misses. The cache, residing closer to the CPU than main memory, dissipates less power because the address and data lines are shorter and have less internal capacitance. Inappropriate memory access patterns lead to cache misses, consequently, power expensive main memory accesses.

2.6.3 CPU Power

When instructions are executed in the CPU, they contribute significant power to the overall power dissipation. The instructions can be divided into four broad categories:

- Load / Store instructions
- Branch instructions
- Type-1 Arithmetic instructions, such as addition, subtraction, shift etc.

- Type-2 Arithmetic instructions, such as multiplication and division.

If we assume that the average power consumption to execute one instruction is Wj , where j represents one of the categories mentioned above, and Ij is the number of times this instruction is executed, we can easily derive the CPU power dissipation as:

$$P_{CPU} \alpha \frac{\sum_{j=1}^{4} (W_j \times I_j)}{\sum_{j=1}^{4} I_j}$$

The power dissipated when an arithmetic instruction is executed depends primarily on the ALU or FPU data path that it instructed by software. Many different ways of optimization exist in this context, as for example replacing a division by power of two with corresponding right shifts.

Finally, instruction scheduling is very important, because unsuccessful scheduling can lead to pipeline stalls, which in turn consume a considerable amount of power.

2.6.4 Other Power Dissipation Sources

There is a number of additional sources of power dissipation during software execution that must be taken into account, as they contribute as an overhead to the overall power dissipation. These sources are the clock distribution and the control logic and they accompany code execution in each cycle. In [15] it was shown that short code sequences in a number of microcontrollers and DSPs always dissipated a smaller amount of power than longer sequences. The program in longer code sequences, which demanded extra execution cycles, took more time to execute so that the overhead was more than that in shorter code sequences.

Despite this problem, power management techniques nowadays tend to eliminate such overhead by cleverly dealing with power dissipation that does not have a direct contribution to the involved computational tasks.

3 DATA ENCRYPTION STANDARD (DES) ALGORITHM

3.1 DES ALGORITHM

The Data Encryption Standard (DES) has been in use since the mid-1970s, adopted by the National Bureau of Standards (NBS) [now the National Institute for Standards and Technology (NIST)] as Federal Information Processing Standard 46 (FIPS 46-3) and by the American National Standards Institute (ANSI) as X3.92. DES was the first crypto scheme commonly seen in non-governmental applications and was the catalyst for modern "public" cryptography.

DES uses the Data Encryption Algorithm (DEA), a secret key block-cipher employing a 56-bit key operating on 64-bit blocks. FIPS 81 describes four modes of DES operation: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB). Despite all of these options, ECB is the most commonly deployed mode of operation.

3.1.1 DES Encryption Operational Overview

DES encrypts and decrypts data in 64-bit blocks, using a 64-bit key (although the effective key strength is only 56 bits). It takes a 64-bit block of plaintext as input and outputs a 64-bit block of ciphertext as shown in Figure 3.1. Since it always operates on blocks of equal size and it uses both permutations and substitutions in the algorithm, DES is both a block cipher and a product cipher [16-17].

DES has 16 rounds, meaning the main algorithm is repeated 16 times to produce the ciphertext. It has been found that the number of rounds is exponentially proportional to the amount of time required to find a key using a brute-force attack. So as the number of rounds increases, the security of the algorithm increases exponentially.

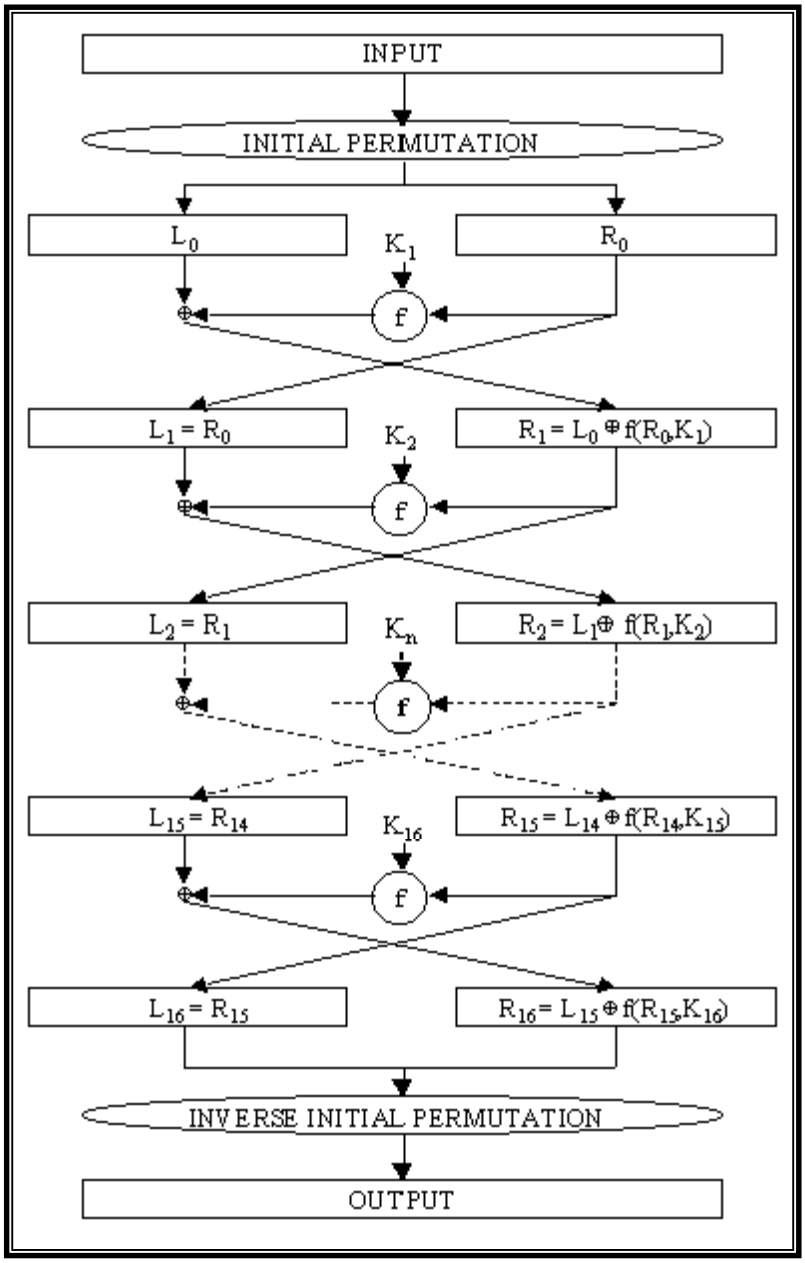


Figure 3.1: DES Encryption Cipher Algorithm

The three main phases of DES algorithm are:

3.1.1.1 Key Scheduling

Although the input key for DES is 64 bits long, the actual key used by DES is only 56 bits in length. The least significant (right-most) bit in each byte is a parity bit, and should be set so that there are always an odd number of 1s in every byte. These parity bits are ignored; so only the seven most significant bits of each byte are used, resulting in a key length of 56 bits [17].

The first step is to pass the 64-bit key through a permutation called Permuted Choice 1, or PC-1. Now that we have the 56-bit key, the next step is to use this key to generate 16 48-bit subkeys, called K[1]-K[16], which are used in the 16 rounds of DES for encryption and decryption as shown in Figure 3.2.

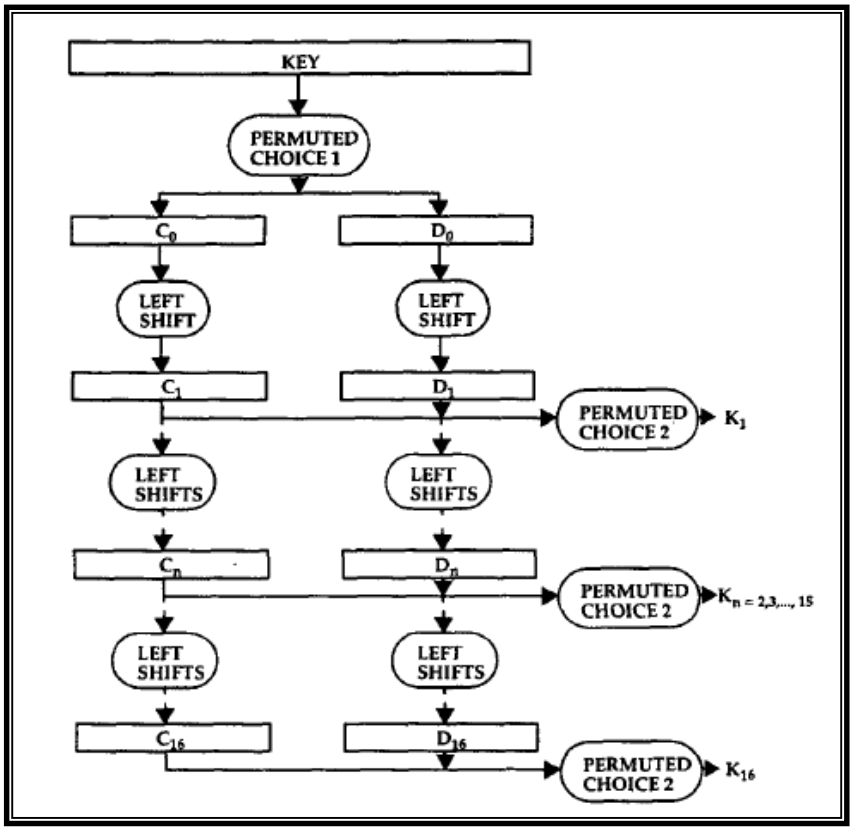


Figure 3.2: Key Scheduling

The procedure for generating the subkeys - known as key scheduling as shown in Figure 3.3:

1. Set the round number R to 1.
2. Split the current 56-bit key, K, up into two 28-bit blocks, L (the left-hand half) and R (the right-hand half).
3. Rotate L left by the number of bits specified for the current round, and rotate R left by the same number of bits as well.
4. Join L and R together to get the new K.
5. Apply Permuted Choice 2 (PC-2) to K to get the final K[R], where R is the round number we are on.
6. Increment R by 1 and repeat the procedure until we have all 16 subkeys K[1]-K[16].

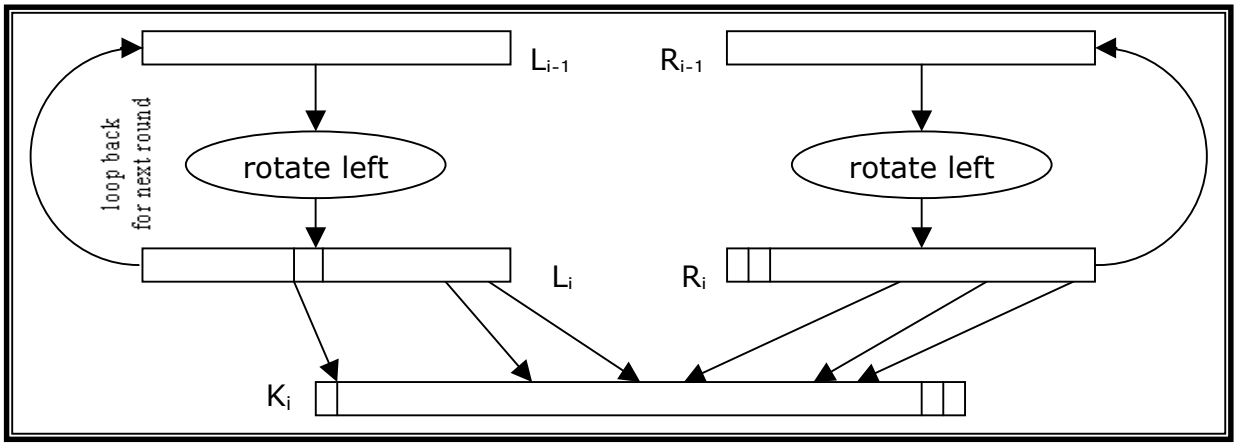


Figure 3.3: Generation of K[I] in Round I

3.1.1.2 Plaintext Preparation

Once the key scheduling has been performed, the next step is to prepare the plaintext for the actual encryption. This is done by passing the plaintext through a permutation called the Initial Permutation (IP).

3.1.1.3 DES Core

Once the key scheduling and plaintext preparation have been completed, the main DES core algorithm performs the actual encryption or decryption, as shown in Figure 3.4. The 64-bit block of input data is first split into two halves, L and R. L is the left-most 32 bits, and R is the right-most 32 bits.

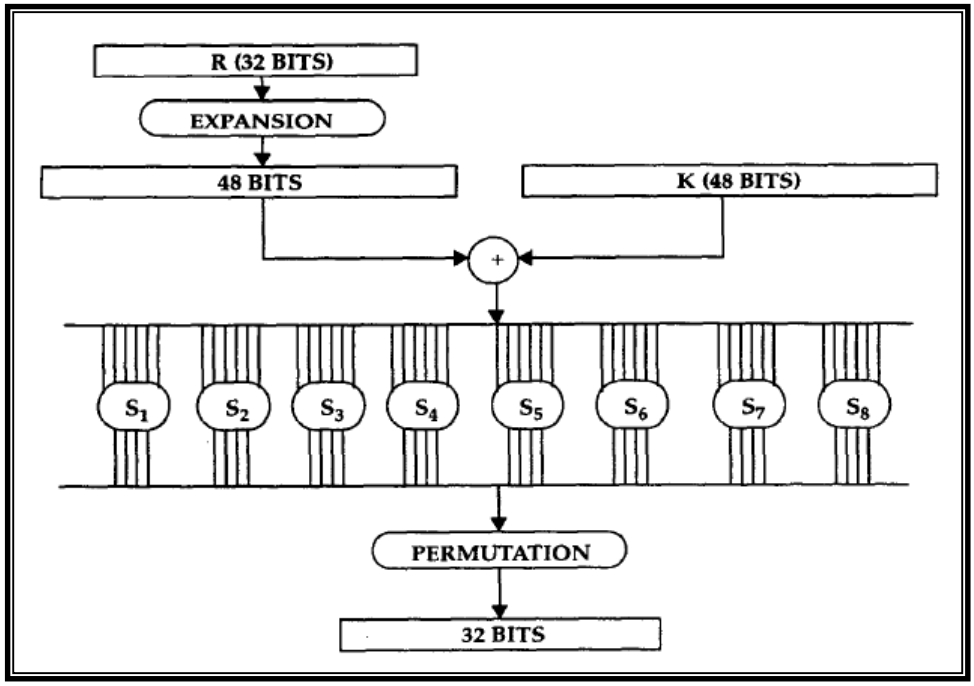


Figure 3.4: DES Core

The following process is repeated 16 times, making up the 16 rounds of standard DES [16-17]. We call the 16 sets of halves L[0]-L[15] and R[0]-R[15].

1. R[I-1] - where I is the round number, starting at 1 - is taken and fed into Expansion Permutation, except that some of the bits are used more than once. This expands the number R[I-1] from 32 to 48 bits to prepare for the next step.
2. The 48-bit R[I-1] is XORed with K[I] and stored in a temporary buffer so that R[I-1] is not modified.
3. The result from the previous step is now split into 8 segments of 6 bits each. The left-most 6 bits are B[1], and the right-most 6 bits are B[8]. These blocks form the index into the S-boxes, which are used in the next step. The Substitution boxes, known as S-boxes, are a set of 8 two-dimensional arrays, each with 4 rows and 16 columns. The numbers in the boxes are always 4 bits in length, so their values range from 0-15. The S-boxes are numbered S[1]-S[8].
4. Starting with B[1], the first and last bits of the 6-bit block are taken and used as an index into the row number of S[1], which can range from 0 to 3, and the middle four bits are used as an index into the column number, which can range from 0 to 15. The number from this position in the S-box is retrieved and stored away. This is repeated with B[2] and S[2], B[3] and S[3], and the others up to B[8] and S[8]. At this point, you now have 8 4-bit numbers, which when strung together one after the other in the order of retrieval, give a 32-bit result.
5. The result from the previous stage is now passed into the P Permutation.
6. This number is now XORed with L[I-1], and moved into R[I]. R[I-1] is moved into L[I].
7. At this point we have a new L[I] and R[I]. Here, we increment I and repeat the core function until I = 17, which means that 16 rounds have been executed and keys K[1]-K[16] have all been used.

When L[16] and R[16] have been obtained, they are joined back together in the same fashion they were split apart (L[16] is the left-hand half, R[16] is the right-hand half), then the two halves are swapped, R[16] becomes the left-most 32

bits and L[16] becomes the right-most 32 bits of the pre-output block and the resultant 64-bit number is called the pre-output.

The final step is to apply the Inverse Initial Permutation IP^(-1) to the pre-output. The result is the completely encrypted ciphertext.

3.1.2 DES Decryption

The method described above will encrypt a block of plaintext and return a block of ciphertext. In order to decrypt the ciphertext and get the original plaintext again, the procedure is simply repeated but the subkeys are applied in reverse order, from K[16]-K[1]. That is, stage 2 of the Core Function as outlined above changes from R[I-1] XOR K[I] to R[I-1] XOR K[17-I]. Other than that, decryption is performed exactly the same as encryption.

3.2 OPTIMIZATION TECHNIQUES FOR DES

Achieving small code size and fast execution time for an implementation of DES is crucial with regard to the practicalities of its use in embedded applications, in which the available processing power is often very limited. Various optimization techniques can achieve improvement in both code size and execution time. Since DES operates on blocks of data, which are only 8 bytes in length, and the size of the data that needs to be encrypted or decrypted is often quite large, the speed of encryption/decryption in a DES implementation is an important factor. Typically the DES algorithm is implemented in software for these reasons, efficient software optimizations are an essential consideration. In addition to considerations of speed, for real-time embedded applications efficient usage of limited ROM and RAM by a DES implementation is crucial.

3.2.1 Initial Permutation and Inverse Initial Permutation

The DES specified by Federal Information Processing Standard (FIPS) naive implementation uses lookup table for the Initial Permutation, which takes more memory and is less efficient. Table 3.1 shows the lookup table used for implementing Initial Permutation.

Table 3.1: Lookup Tables for Initial Permutations

Bit	0	1	2	3	4	5	6	7
1	58	50	42	34	26	18	10	2
9	60	52	44	36	28	20	12	4
17	62	54	46	38	30	22	14	6
25	64	56	48	40	32	24	16	8
33	57	49	41	33	25	17	9	1
41	59	51	43	35	27	19	11	3
49	61	53	45	37	29	21	13	5
57	63	55	47	39	31	23	15	7

Table 3.2: Lookup Tables for Inverse Initial Permutations

Bit	0	1	2	3	4	5	6	7
1	40	8	48	16	56	24	64	32
9	39	7	47	15	55	23	63	31
17	38	6	46	14	54	22	62	30
25	37	5	45	13	53	21	61	29
33	36	4	44	12	52	20	60	28
41	35	3	43	11	51	19	59	27
49	34	2	42	10	50	18	58	26
57	33	1	41	9	49	17	57	25

For example, we can use the IP table to figure out how bit 30 of the original 64-bit data transforms to a bit in the new 64-bit data. Find the number 30 in the table, and notice that it belongs to the column labeled 4 and the row labeled 17. Add up the value of the row and column to find the new position of the bit within the data. For bit 30, 17 + 4 = 21, so bit 30 becomes bit 21 of the new 64-bit data.

The SwapMove operation implements the initial permutation in a very efficient way. It swaps the bits in Right, masked by M with the bits in Left, masked by (M << N) [18]. That is,

```
SwapMove(Left, Right, M, N)
{
        Temp = (( Left >> N) XOR Right) AND M
        Right = Right XOR Temp
        Left = Left XOR (Temp << N)
}
```

The initial permutation can be implemented by 5 calls to the SwapMove operation. Here the input Left is the left half of the 64 bit input data, and the input Right is the right half. Throughout, we are assuming a 32-bit word size.

IP(Left, Right)

SwapMove(Left, Right, 0x0F0F0F0F,4)
SwapMove(Left, Right, 0x0000FFFF,16)
SwapMove(Right, Left, 0x33333333,2)
SwapMove(Right, Left, 0x00FF00FF,8)
SwapMove(Left, Right, 0x55555555,1)

The Initial Permutation takes 30 operations in all that is, 15 XOR operations, 10 Shift operations and 5 AND operations and also without using any lookup table thus saving 64 bytes each for initial and inverse initial permutation. The efficiency is also increased, as with lookup table all such operations would be performed on read-a-bit and set-a-bit basis.

Table 3.2 shows the look up table for Inverse Initial Permutation. Since the inverse initial permutation simply undoes the permutation performed by the initial permutation, the SwapMove functions are called in the reverse order to implement the inverse initial permutation.

IIP(Left, Right)

SwapMove(Left, Right, 0x55555555,1)
SwapMove(Right, Left, 0x00FF00FF,8)
SwapMove(Right, Left, 0x33333333,2)
SwapMove(Left, Right, 0x0000FFFF,16)
SwapMove(Left, Right, 0x0F0F0F0F,4)

3.2.2 Permuted Choice 1(PC-1)

Also the DES naive implementation uses lookup table for the Permuted Choice-1 operation, which results in a 56-bit key. Figure 3.6 shows the lookup table used for Permuted Choice – 1.

These tables are used just like Initial Permutation and Inverse Initial Permutation. For example, we can use the PC-1 table to figure out how bit 30 of the original 64-bit key transforms to a bit in the new 56-bit key. Find the number 30 in the table, and notice that it belongs to the column labeled 5 and the row labeled 36. Add up the value of the row and column to find the new position of the bit within the key. For bit 30, 36 + 5 = 41, so bit 30 becomes bit 41 of the new 56-bit key. Note that bits 8, 16, 24, 32, 40, 48, 56 and 64 of the original key are not in the table. These are the unused parity bits that are discarded when the final 56-bit key is created.

Table 3.3: Lookup Tables for PC-1

Bit	**0**	**1**	**2**	**3**	**4**	**5**	**6**
1	57	49	41	33	25	17	9
8	1	58	50	42	34	26	18
15	10	2	59	51	43	35	27
22	19	11	3	60	52	44	36
29	63	55	47	39	31	23	15
36	7	62	54	46	38	30	22
43	14	6	61	53	45	37	29
50	21	13	5	28	20	12	4

The HalfMove operation implements the PC1 permutation in a very efficient way. It swaps the bits in half, masked by M with the bits in half, masked by (M >> N)[18]. That is,

```
HalfMove(Half, M, N)
{
        Temp = (( Half << (16-N)) XOR Half) AND M
        Half = Half XOR Temp XOR (Temp >> (16- N))
}
```

The PC-1 permutation can be implemented by making calls to *SwapMove* and *HalfMove* functions.

PC-1

SwapMove(Right, Left, 0x0F0F0F0F,4)
HalfMove (Left,0xCCCC0000,-2)
HalfMove (Right,0xCCCC0000,-2)
SwapMove(Right, Left, 0x55555555,1)
SwapMove(Left, Right, 0x00FF00FF,8)
SwapMove(Right, Left, 0x55555555,1)
Right = ((Right AND 0x000000FF) << 16) |
(Right AND 0x0000FF00) |
((Right AND 0x00FF0000) >> 16) |
((Left AND 0xFF000000) >> 4))
Left = Left AND 0x0FFFFFFF

Some bit shifting is performed at the end because calls to the *SwapMove* and *HalfMove* operations do not quite produces the result we want. The resulting 28 bits in each half are aligned to the least significant end so that when the bits out of the key schedule are used as indices into other tables, one less bit shift will be required.

The PC-1 Permutation takes 47 operations in all that is, 18 XOR operations, 15 Shift operations, 11 AND operations and 3 OR operations and also without using any lookup table thus saving 56 bytes. The computing efficiency is increased very high.

3.2.3 The Substitution Operation

Rather than using eight two-dimensional tables, the selection function data may be represented in a single 64-element array where each element is 32 bits long. For a given 6-bit input to the 8 original selection functions, the corresponding 4 bit outputs are concatenated together to form a 32-bit element of the new array. Then, depending on which selection function output value we are interested in, we mask out unneeded bits from this 32-bit value. The total output of the selection functions after concatenation will then simply is the 32 bit outputs.

3.2.4 Permuted Choice 2(PC2)

Permuted choice 2 and the cipher function fare the two major operations in each round of DES. Optimizing the permuted choice 2 brings significant gains in performance in time. The simplest optimization is to implement this function as a lookup table. The bigger the lookup table for PC2, the faster the operation of PC2. Hence, there is a trade-off between time and code size. For real-time embedded applications where ROM size and speed are equally important, it was decided to implement PC2 as a 4-bit lookup table. This requires (24 x 8) bytes and 14 lookups each using a 4-byte mask. Had speed been a more crucial factor, the table was implemented, say, as an 8-bit lookup, which would have required 48 bytes.

As a consequence of the above optimizations, both the encryption and the decryption operations execute in fixed time. The execution time has no correlation with the given input data and the key. Thus no timing information leaked, in contrast to some cases of the naive implementation.

3.3 TRIPLE DES (3DES) ALGORITHM

When it was found that a 56-bit key of DES is not enough to guard against brute force attacks, 3DES was chosen as a simple way to enlarge the key space without a need to switch to a new algorithm. The use of three steps is essential to prevent attacks that are effective against double DES encryption.

The 3DES algorithm is a simple variant of the DES algorithm. The DES function is replaced by three rounds of that function, an encryption followed by a decryption followed by an encryption, each with independent keys, k1, k2 and k3 [19] as shown in Figure 3.5. In general 3DES with three different keys has a key length of 168 bits: three 56-bit DES keys (with parity bits 3DES has the total storage length of 192 bits). When all three keys (k1, k2 and k3) are the same, 3DES is equivalent to DES.

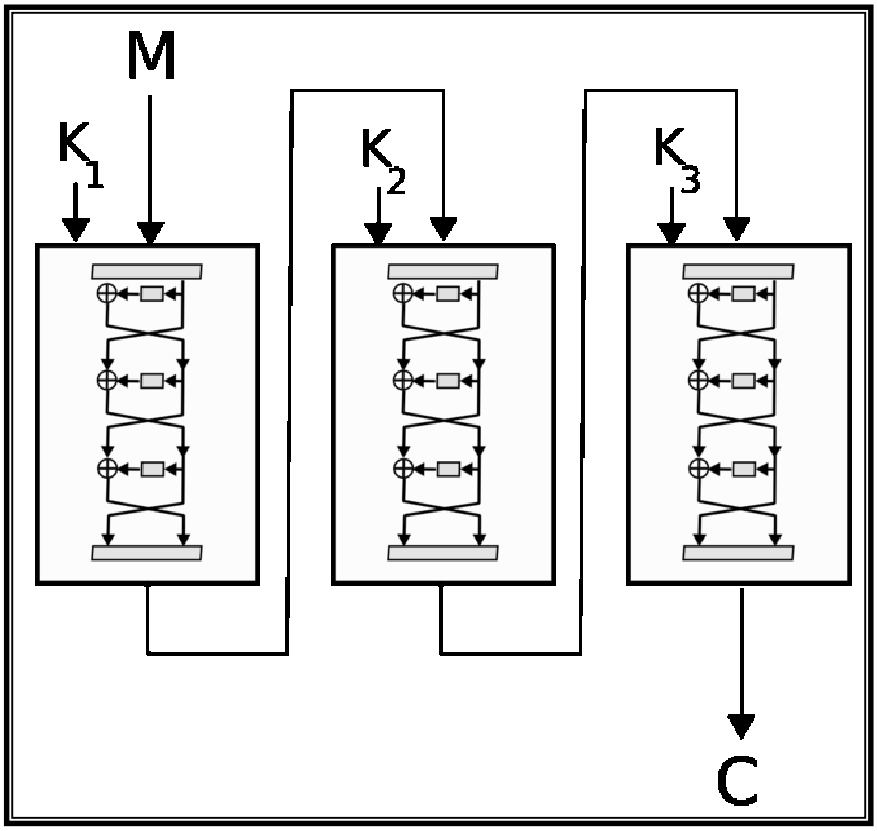


Figure 3.5: Triple DES

The simplest variant of 3DES commonly known as EDE, operates as follows:

$$\mathbf{DES}(k_3;\mathbf{DES}^{-1}(k_2;\mathbf{DES}(k_1;M)))$$

where M is the message block to be encrypted and k_1, k_2, and k_3 are DES keys.

The optimization techniques applied for the DES algorithm are also used for 3DES algorithm.

4.1 AES ALGORITHM

This standard specifies the **Rijndael** algorithm, a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. The algorithm may be used with three different key lengths indicated above, and therefore these different "flavors" may be referred to as "AES-128", "AES-192", and "AES-256".

The input and output for the AES algorithm each consist of sequences of 128 bits (digits with values of 0 or 1). These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The Cipher Key for the AES algorithm is a sequence of 128, 192 or 256 bits [20].

The basic unit for processing in the AES algorithm is a byte. All byte values in the AES algorithm will be presented as the concatenation of its individual bit.

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State. The State consists of four rows of bytes, each containing *Nb* bytes, where *Nb* is the block length divided by 32. In the State array denoted by the symbol *s*, each individual byte has two indices, with its row number *r* in the range $0 \leq r < 4$ and its column number *c* in the range $0 \leq c < Nb$. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or $s[r,c]$. For this standard, $Nb = 4$, i.e., $0 \leq c < 4$.

4.2 THE RIJNDAEL ALGORITHM

Rijndael is an iterated block cipher with a variable block length and key length. The block length and the key length can be independently set to 128, 192 or 256 bits, whereas AES restricts the block length to 128 bits only [20]. At the start of the Cipher, the input is copied to the State array. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first *Nr-1* rounds. The final State is then copied to the output. The key that is provided as input is expanded into an array of forty-four 32-bit

words, w[i]. Four distinct words (128 bits) serve as a round key for each round; these are indicated in Figure 4.1.

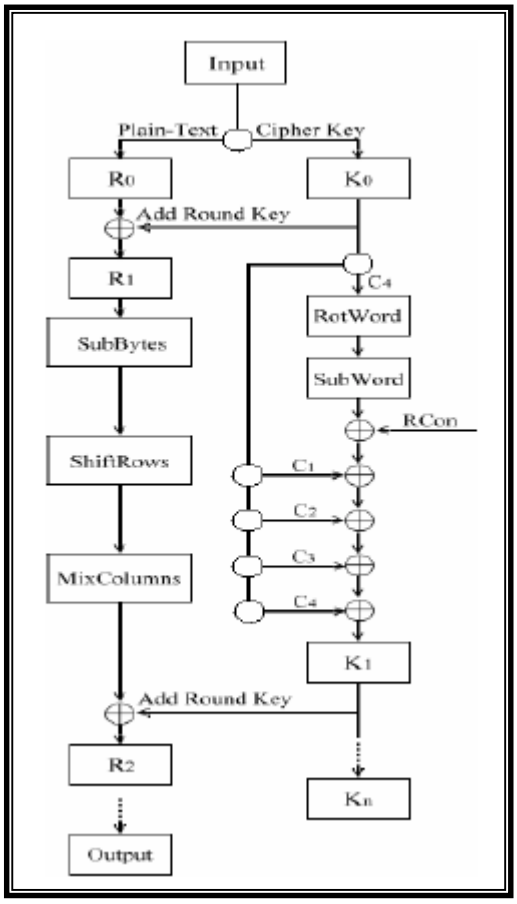


Figure 4.1: AES Encryption Procedure

Four different stages are used, one of permutation and three of substitution:

- Substitute bytes: Use an S-box to perform a byte-by-byte substitution of the block
- Shift rows: A simple permutation
- Mix Columns: A substitution that makes use of arithmetic over GF(2^8)
- Add round key: A simple bitwise XOR of the current with a portion of the expanded key

Only the Add Round Key stage makes use of the key. For this reason, the cipher begins and ends with an Add Round Key stage. The other three stages together provide confusion, diffusion, and non-linearity, but by themselves would provide no security because they do not use the key. Each stage is easily reversible. For

the Substitute Byte, Shift Row, and MixColumns stages, an inverse function is used in the decryption algorithm. For the Add Round Key stage, the inverse is achieved by XORing the same round key to the block, using the result that $A \oplus B = B$. The decryption algorithm is not identical to the encryption algorithm. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. The final round of both encryption and decryption consists of only three stages.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 4.2: Pseudo Code for the AES Cipher

The Cipher is described in the pseudo code in Figure 4.2. The individual transformations - **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey** – process the State and are described in the following subsections. In Figure 4.2, the array w[] contains the key schedule.

As shown in Figure 4.2, all *Nr* rounds are identical with the exception of the final round, which does not include the MixColumns transformation.

4.2.1 SubBytes Transformation

The SubBytes transformation is a non-linear byte substitution that operates independently on each byte of the State matrix, using a substitution table (S-box), which is invertible.

4.2.2 ShiftRows Transformation

In the ShiftRows transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$ is not shifted.

Specifically, the ShiftRows transformation proceeds as follows:

$$S'_{r,c} = S_{r,(c+\text{ shift}(r,Nb))\bmod Nb} \text{ for } 0 < r < 4 \text{ and } 0 \le c < Nb,$$

where the shift value *shift*(*r*,*Nb*) depends on the row number, *r*, as follows (recall that *Nb*= 4):

$$\text{shift}(1, 4) = 1; \text{shift}(2, 4) = 2; \text{shift}(3, 4) = 3$$

This has the effect of moving bytes to "lower" positions in the row (i.e., lower values of *c* in a given row), while the "lowest" bytes wrap around into the "top" of the row (i.e., higher values of *c* in a given row).

4.2.3 MixColumns Transformation

The MixColumns transformation operates on the State column-by-column, treating each column as a four-term polynomial. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by the expression,

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

This polynomial is co-prime to $(x^4 + 1)$, and therefore the transformation is invertible. This transformation can be written under the form of a matrix multiplication. Pose $s'_c(x) = a(x) \otimes s_c(x)$, for $0 \le c \le 3$, that is for all the 4

columns in the State matrix [21]. As a result of this multiplication, the 4 bytes in a column c are replaced by the following ones (for c = 0, 1, 2 and 3):

$$S'_{0,c} = 02 * S_{0,c} \oplus 03 * S_{1,c} \oplus S_{2,c} \oplus S_{3,c}$$
$$S'_{1,c} = S_{0,c} \oplus 02 * S_{1,c} \oplus 03 * S_{2,c} \oplus S_{3,c}$$
$$S'_{2,c} = S_{0,c} \oplus S_{1,c} \oplus 02 * S_{2,c} \oplus 03 * S_{3,c}$$
$$S'_{3,c} = 03 * S_{0,c} \oplus S_{1,c} \oplus S_{2,c} \oplus 02 * S_{3,c}$$

where the * operator stands for a multiplication in $GF(2^8)$, with:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

as irreducible generator polynomial. Performing a complete round means simply applying these 4 transformations to the State matrix, in the following order:

round = {SubBytes, ShiftRows,MixColumns, AddRoundKey}

Performing the final round means simply applying to the State matrix the following transformations, in the order:

finalround = {SubBytes, ShiftRows, AddRoundKey}

4.2.4 AddRoundKey Transformation

In the AddRoundKey transformation, a Round Key is added to the State matrix by a simple bitwise XOR operation. Each Round Key consists of *Nb* words from the key schedule. Those *Nb* words are each added into the columns of the State, such that,

$[S'_{0,c}, S'_{1,c}, S'_{2,c}, S'_{3,c}] = [S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus [W_{round*Nb+c}]$ for $0 \le c < Nb$,

where $[w_i]$ are the key schedule words, and *round* is a value in the range $0 \le round \le N_r$. In the Cipher, the initial Round Key addition occurs when *round* = 0, prior to the first application of the round function as shown in Figure 4.2. The application of the AddRoundKey transformation to the *Nr* rounds of the Cipher occurs when $1 \le round \le Nr$.

4.3 KEY EXPANSION

The AES algorithm takes the Cipher Key ***K***, and performs a Key Expansion routine to generate a key schedule. 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 $[w_i]$, with *i* in the range $0 \leq i < Nb\ (Nr + 1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo code in Figure 4.3.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Figure 4.3: Pseudo Code for Key Expansion

SubWord is a function that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word. The function RotWord takes a word [*a*0,*a*1,*a*2,*a*3] as input, performs a cyclic permutation, and returns the word [*a*1,*a*2,*a*3,*a*0]. The round constant word array, Rcon[i], contains the values

given by $[x^{i-1},\{00\},\{00\},\{00\}]$, with x^{i-1} being powers of x (x is denoted as {02}) in the field GF(2^8) (note that i starts at 1, not 0).

From Figure 4.3, it can be seen that the first *Nk* words of the expanded key are filled with the Cipher Key. Every following word, w[i], is equal to the XOR of the previous word, w[i-1], and the word *Nk* positions earlier, w[i-Nk]. For words in positions that are a multiple of *Nk*, a transformation is applied to w[i-1] prior to the XOR, followed by an XOR with a round constant, Rcon[i]. This transformation consists of a cyclic shift of the bytes in a word (RotWord), followed by the application of a table lookup to all four bytes of the word (SubWord).

It is important to note that the Key Expansion routine for 256-bit Cipher Keys (*Nk* = 8) is slightly different than for 128- and 192-bit Cipher Keys. If *Nk* = 8 and i-4 is a multiple of *Nk*, then SubWord is applied to w[i-1] prior to the XOR.

4.4 INVERSE CIPHER

The Cipher transformations in Sec. 4.1 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - **InvShiftRows**, **InvSubBytes**, **InvMixColumns**, and **AddRoundKey** [17]– process the State and are described in the following subsections.

The Inverse Cipher is described in the pseudo code in Figure 4.4. In Figure 4.4, the array w[] contains the key schedule, which was described previously.

4.4.1 InvShiftRowsTransformation

InvShiftRows is the inverse of the ShiftRows transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets).

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)
      InvSubBytes(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)
   end for

   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])

   out = state
end
```

Figure 4.4: Pseudo Code for the AES Inverse Cipher

4.4.2 InvSubBytes Transformation

InvSubBytes is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation followed by taking the multiplicative inverse in GF(2^8).

4.4.3 InvMixColumns Transformation

InvMixColumns is the inverse of the MixColumns transformation. InvMixColumns operates on the State column-by-column, treating each column as a four term polynomial. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$$

4.4.4 Inverse of the AddRoundKey Transformation

AddRoundKey, which was described in Sec. 4.2.4, is its own inverse, since it only involves an application of the XOR operation.

4.5 OPTIMIZATION TECHNIQUES FOR AES ALGORITHM

This section illustrates the optimized version of the Rijndael AES algorithm. Both the encryption and the decryption algorithms have been optimized. All the operations in the AES algorithm described above, with the exception of the MixColumn transformation, are quite straightforward to implement, therefore optimization is mainly concentrated on implementation of the MixColumn transformation and also by the fact that, compact implementations of MixColumn transformation is responsible for about 50% of the encryption and decryption time.

4.5.1 Optimize MixColumn for Encryption

In the MixColumn transformation, each column of the State is considered as a polynomial with coefficients in GF(2^8), and multiplied modulo $x^4 + 1$ with a fixed polynomial $\{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, co-prime to the modulo [21]. Assuming that the column before transformation consists of the bytes (b_0, b_1, b_2, b_3), each byte representing a polynomial in GF(2^8), the transformed column bytes (c_0, c_1, c_2, c_3) are computed as follows:

$$\left.\begin{aligned} c_0 &= \{02\} * b_0 \oplus \{03\} * b_1 \oplus \{01\} * b_2 \oplus \{01\} * b_3 \\ c_1 &= \{01\} * b_0 \oplus \{02\} * b_1 \oplus \{03\} * b_2 \oplus \{01\} * b_3 \\ c_2 &= \{01\} * b_0 \oplus \{01\} * b_1 \oplus \{02\} * b_2 \oplus \{03\} * b_3 \\ c_3 &= \{03\} * b_0 \oplus \{01\} * b_1 \oplus \{01\} * b_2 \oplus \{02\} * b_3 \end{aligned}\right\} \quad (4.1)$$

where * denotes polynomial multiplication in GF(2^8) defined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, and $\oplus$ denotes simple XOR at byte level. Multiplication by {02} in GF(2^8) can be implemented at byte level with a left shift followed by a conditional bitwise XOR with {1b}, called as doubling operation. Multiplication by larger coefficients can be implemented with repeated multiplications by {02} and XORs with previously calculated results.

The MixColumn implementation described by Gladman [26] requires 4 XORs, 3 rotate and one doubling operation, incurring 16 XORs, 12 rotates and 4 doubling operations per AES round.

The new optimized MixColumn implementation that requires 3 XORs, 3 rotations and one doubling operation, incurring 12 XORs, 12 rotations and 4 doubling operations per AES round. However, using the ARM barrel shifter, the 12 rotations can be combined with 12 XORs without any penalty, resulting in 12 XORs and 4 doubling operations effectively per round. The proposed MixColumn implementation, in addition to cutting down the number of logical operations, can support all block lengths multiples of 32-bits.

Assuming that b = (b_0, b_1, b_2, b_3) is the input column to be transformed, s as a 32-bit temporary variables, and c = (c_0, c_1, c_2, c_3) is the result, the steps of the MixColumn transformation are given as follows,

$$\left.\begin{aligned} s_0 &= b_0 \oplus b_1 \\ s_1 &= b_1 \oplus b_2 \\ s_2 &= b_2 \oplus b_3 \\ s_3 &= b_3 \oplus b_0 \end{aligned}\right\} \qquad (4.2)$$

$$\left.\begin{aligned} c_0 &= \{02\} * s_0 \oplus\ b_1 \oplus b_2 \oplus b_3 \\ c_1 &= b_0 \oplus \{02\} * s_1 \oplus b_2 \oplus b_3 \\ c_2 &= b_0 \oplus b_1 \oplus \{02\} * s_2 \oplus b_3 \\ c_3 &= b_0 \oplus b_1 \oplus b_2 \oplus \{02\} * s_3 \end{aligned}\right\} \qquad (4.3)$$

Therefore the MixColumns transformation is computed in only 3 steps: a sum step, a doubling step and a final sum step per AES rotation. The final result of Equation 4.3 is equal to Equation 4.1.

4.5.2 Optimize MixColumn for Decryption

In the case of decryption this double-and-add method is used in a more substantial way, and the steps necessary to compute the InvMixColumns transformation are 7: 4 sum steps and 3 doubling steps [21]. In the case of InvMixColumns the double-and-add method can be improved by considering the particular values of the constant coefficients. Note that there are only two coefficients containing a bit 1 in the third position, namely the coefficients 0x0e and 0x0d. These coefficients are used in combination with the operands b_0 and b_2 contained both in the first row and in the third row, and are used in combination

with the operands b_1 and b_3 contained both in the second row and in the fourth row.

To save a doubling operation we can add these two operand pairs and store the result in b_0 and b_1, respectively. Instead of calculating the sub-expression $04 * b_0 \oplus 04 * b_2$, we can calculate the sub-expression expression $04*(b_0 \oplus b_2)$, since we do not need to store separately either addend. Moreover, note that every coefficient contains a bit 1 in the fourth position. The last calculation deals with this bit. Hence we can add the previously computed values b_0 and b_1, which are $04*(b_0 \oplus b_2)$ and $04*(b_1 \oplus b_3)$, respectively, and then double them, so that the sub-expression $08*(b_0 \oplus b_1 \oplus b_2 \oplus b_3)$ is obtained, which must be accumulated to every operand c_i.

Assuming that $b = (b_0, b_1, b_2, b_3)$ is the input column to be transformed, dou8, dou4a and dou4b as a 32-bit temporary variables, and $c = (c_0, c_1, c_2, c_3)$ is the result, the steps of the InvMixColumn transformation are given as follows,

$$\left.\begin{aligned} dou8 &= \{08\} * (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \\ dou4a &= \{04\} * (b_0 \oplus b_2) \\ dou4b &= \{04\} * (b_1 \oplus b_3) \end{aligned}\right\} \quad (4.4)$$

$$\left.\begin{aligned} c_0 &= dou8 \oplus dou4a \oplus \{02\} * (b_0 \oplus b_1) \oplus b_1 \oplus b_2 \oplus b_3 \\ c_1 &= dou8 \oplus dou4b \oplus \{02\} * (b_1 \oplus b_2) \oplus b_0 \oplus b_2 \oplus b_3 \\ c_2 &= dou8 \oplus dou4a \oplus \{02\} * (b_2 \oplus b_3) \oplus b_0 \oplus b_1 \oplus b_3 \\ c_3 &= dou8 \oplus dou4b \oplus \{02\} * (b_0 \oplus b_3) \oplus b_0 \oplus b_1 \oplus b_2 \end{aligned}\right\} \quad (4.5)$$

These optimizations can eliminate a number of shifts and rotates operations on the ARM architectures. Also, it enhances the performance of AES on all ARM cores. The new implementation is the most compact implementation, preserves the flexibility of Rijndael and exhibits the highest encryption performance on ARM processors, compared with similar implementations claiming low memory use. It exploits the ISA features of ARM by efficiently rearranging some operations and reducing their number.

5.1 RSA ALGORITHM

The RSA scheme is a block cipher in which the plaintext and ciphertext are integers between 0 and n-1 for some n. The scheme developed by Rivest, Shamir, and Adleman makes use of an expression with exponentials. Plaintext is encrypted in blocks, with each block having a binary value less than some number n. That is, block size must be less than or equal to $\log_2(n)$; generally, the block size is k bits, where $2^k < n \leq 2^{k+1}$ [17]. Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

$$C = M^e \bmod n$$

$$M = C^d \bmod n$$

RSA gets its security from the difficulty of factoring large numbers. The public and private keys are functions of a pair of large (100 to 200 digits or even larger) prime numbers. Recovering the plaintext from the public key and the ciphertext is conjectured to be equivalent to factoring the product of the two primes.

To generate the two keys, choose two random large prime numbers, *p* and *q* and two integers, n and m, such that $0 < m < n$. For maximum security, choose *p* and *q* of equal length. Compute the product:

$$n = pq$$

For p, q prime, $\phi(pq) = (p-1)(q-1)$, where $\phi(n)$ is the Euler totient function, which is the number of positive integers less than n and relatively prime to n. Finally, use the extended Euclidean algorithm to compute the decryption key, *d*, such that,

$$ed \equiv 1 \bmod \phi(n)$$

$$d \equiv e^{-1} \bmod \phi(n)$$

That is, e and d are multiplicative inverses mod $\phi(n)$. According to the rules of modular arithmetic, this is true only if d (and therefore e) is relatively prime to $\phi(n)$. Equivalently, $\gcd(\phi(n),d) = 1$.

Note that *d* and *n* are also relatively prime. The numbers *e* and *n* is the public key and the number *d* and *n* is the private key. The two primes, *p* and *q*, are no longer needed. They should be discarded, but never revealed.

To encrypt a message *M*, first divide it into numerical blocks smaller than *n* (with binary data, choose the largest power of 2 less than *n*). That is, if both *p* and *q* are 100-digit primes, then *n* will have just under 200 digits and each message block, M_i, should be just under 200 digits long. (If a fixed number of blocks are to be encrypted, one can pad them with a few zeros on the left to ensure that they will always be less than *n*.) The encrypted message C_i, will be made up of similarly sized message blocks.

To decrypt a message, take each encrypted block c_i and compute

$$M_i = C_i^d \bmod n$$

the formula recovers the message. This is summarized as follows,

Public Key:

n product of two primes, *p* and *q* (*p* and *q* must remain secret)

e relatively prime to $(p - 1)(q - 1)$

Private Key:

$d = e^{-1} \bmod ((p - 1)(q - 1))$

Encrypting:

$c = m^e \bmod n$

Decrypting:

$m = c^d \bmod n$

The message could just be easily encrypted with *d* and decrypted with *e*; the choice is arbitrary.

5.2 EFFICIENT ALGORITHMS FOR RSA IMPLEMENTATION

5.2.1 Efficient Method for Primality Test

In implementing RSA algorithm, the primality test is carried by introducing a very fast and efficient method instead of using the probabilistic test. The Miller-Rabin Test will merely determine that a given integer is probably prime and also which is a tedious procedure. The procedure for testing whether a given integer *n* is prime is to perform some calculations that involves *n* and *a* randomly chosen integer *a.* If *n* "fails" the test, then n is not a prime. If *n* "passes" the test, then *n* may be prime or nonprime.

In the approach introduced here, it tests the prime number on the actual calculations performed. The number of test required to test the primality of a given number n is SquareRoot(n). Actually, because all the even numbers can be immediately rejected, the correct figure is SquareRoot(n)/2. Consider the following example,

Suppose, n = 105, then according to Miller-Rabin Test it will take ln(n)/ 2 = 3 test. While in this approach, it will take SquareRoot(n)/2 = 5 test, but will surely determine whether the given number n is prime or nonprime.

5.2.2 Extended Euclid Algorithm

Euclid's algorithm can be extended so that, in addition to finding gcd (a, b), if the gcd is 1, the algorithm returns the multiplicative inverse of b [17].

Given a and b, the extended Euclid algorithm computes g, u, and v such that

$$g = \gcd(a, b) = u \cdot a + v \cdot b$$

This algorithm is used to compute the modular inverse. If g = 1, then

$$1 = u \cdot a + v \cdot b$$

implies that,

$$1 = u \cdot a \pmod{b}$$

$$1 = v \cdot b \pmod{a}$$

and therefore,

$$u = a^{-1} \pmod{b}$$

$$v = b^{-1} \pmod{a}$$

EXTENDED EUCLID ALGORITHM (a, b)

```
begin
    (g0, g1) = (a, b)
    (u0, u1) = (1, 0)
    (v0, v1) = (0, 1)
    while g1 ≠ 0 do
    begin
        q = g0 div g1
        (g0, g1) = (g1, g0 - g1 · q)
        (u0, u1) = (u1, u0 - u1 · q)
        (v0, v1) = (v1, v0 - v1 · q)
    end
    g = g0 ; u = u0 ; v = v0
end
```

5.2.3 Left to Right Binary Exponentiation

One of the most important arithmetic operations for public-key cryptography is exponentiation.

The most naive way to compute g^e is to do e - 1 multiplications in the group. But for cryptographic applications like RSA, the order of the group g typically exceeds 2^{160} elements, and may exceed 2^{1024}. Most choices of e are large enough that it would be infeasible to compute g^e using e - 1 successive multiplications by g. There are two ways to reduce the time required to do exponentiation. One way is to decrease the time to multiply two elements in the group and the other is to

reduce the number of multiplications used to compute g^e. Ideally, one would do both.

There are basically three types of exponentiation algorithms:

1. *Basic techniques for exponentiation:* Arbitrary choices of the base g and exponent e are allowed.
2. *Fixed-exponent exponentiation algorithms:* The exponent e is fixed and arbitrary choices of the base g are allowed. RSA encryption and decryption schemes benefit from such algorithms.
3. *Fixed-base exponentiation algorithms:* The base g is fixed and arbitrary choices of the exponent e are allowed. ElGamal encryption and signatures schemes and Diffie-Hellman key agreement protocols benefit from such algorithms.

The RSA implementation in this work uses the Left to Right Binary Exponentiation Algorithm for computing the exponentiation, as it reduces the number of multiplication for the computation of exponentiation [17]. The following pseudo-code explains the Left to Right Binary Method for the computation of exponent in the RSA implementation.

LEFT TO RIGHT BINARY METHOD

Input: M, e, n.
Output: C := M^e mod n.

begin

Step 1. if e_{k-1} = 1 then C := M else C := 1
Step 2. for i = k - 2 downto 0
2a. C := C · C (mod n)
2b. if e_i = 1 then C := C ·M (mod n)
Step 3. return C

The binary method mentioned above requires, k – 1 squaring operation at step 2a, the number of Multiplications operations at step 2b is equal to the number of 1s in the binary expansion of e, excluding the MSB.

Therefore, the total number of multiplications required for left to right binary exponentiation method is,

Maximum: (k - 1) + (k - 1) = 2(k - 1)
Minimum : (k - 1) + 0 = k - 1
Average : (k - 1)+ 1/2(k - 1) = 1.5(k - 1)

RSA encryption goes much faster if smart value of *e* is chosen. The three most common choices are 3, 17, and 65537 (216 + 1). Since, the binary representation of 65537 has only two ones, so it takes only 17 multiplications for exponentiation. There are no security problems with using any of these three values for, even if a whole group of users uses the same value for *e*.

6 EFFICIENT C PROGRAMMING FOR ARM

6.1 INTRODUCTION

The ARM is mature and the ANSI C compiler which is capable of producing high quality machine code. However, when writing source code, it is always worthwhile to use programming techniques, which work well on RISC processors such as ARM. This chapter describes some of the techniques that can be useful. It also explains how to use the C language efficiently. These techniques and knowledge will enable programmers to increase execution speed and/or lower code density.

6.2 BASIC C VARIABLE TYPES

ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is RISC load/store architecture. In other words you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly. The C compilers support the basic types *char*, *short*, *int* and *long long* (signed and unsigned), *float* and *double*. Using the most appropriate type for variables is important, as it can reduce code and/or data size and increase performance considerably.

6.2.1 Local Variables Type

Most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit datatype, *int* or *long*, for local variables wherever possible. Avoid using *char* and *short* as local variable types, even if you are manipulating an 8-bit or 16-bit value. For the types *char* and *short* the compiler needs to reduce the size of the local variable to 8 or 16 bits after each assignment. This is called sign-extending for signed variables and zero-extending for unsigned variables [23]. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned char takes one instruction).

These shifts can be avoided by using *int* and *unsigned int* for local variables. This is particularly important for calculations, which first load data into local variables and then process the data inside the local variables. Even if data is input and output as 8- or 16-bit quantities, it is worth considering processing them as 32-bit quantities.

For example, the following code checksums a data packet containing 64 words.

```
int checksum(int *data)
{
        char i;
        int sum=0;
        for( i=0; i<64; i++)
        {
                sum + = data[i];
        }
        return sum;
}
```

At first sight it looks as though declaring i as a *char* variable, is efficient, as *char* uses less register space or less space on the ARM stack than an *int*. But these assumptions are wrong for ARM, as ARM registers are 32-bit and all stack entries and at least 32-bit [22].

The compiler output for this function is as follows,

```
checksum
        MOV         r2, r0                  ; r2 = data
        MOV         r0, #0                  ; sum = 0
        MOV         r1, #0                  ; i = 0
checksum_loop
        LDR         r3, [r2, r1, LSL, #2]   ; r3 = data[i]
        ADD         r1, r1, #1              ; r1 = i + 1
        AND         r1, r1, #0xff           ; i = (char) r1
        CMP         r1, #0x40               ; compare i, 64
```

```
ADD     r0, r3, r0          ; sum + = r3
BCC     checksum_loop       ; if (i < 64) loop
MOV     pc, r14             ; return sum
```

As the compiler inserts an extra AND instruction to reduce a *char* variable to the range 0 to 255 before the comparison with 64. Thus converting local variables from types *char* or *short* to type *int* increases performance and reduces code size.

6.2.2 Space Occupied by Global Data

When declaring global variables in source code to be compiled for ARM, three things are affected by the way the code is structured [24]:

- How much space the variables occupy at run time. This determines the size of RAM required for a program to run. The ARM compilers may insert padding bytes between variables, to ensure that they are properly aligned.
- How much space the variables occupy in the image. This is one of the factors determining the size of ROM needed to hold a program. Some global variables, which are not explicitly initialized in your program, may nevertheless have their initial value stored in the image.
- The size of the code needed to access the variables. Some data organizations require more code to access the data. As an extreme example, the smallest data size would be achieved if all variables were stored in suitably sized bitfields, but the code required to access them would be much larger.

Table 6.1: Required Alignment Table

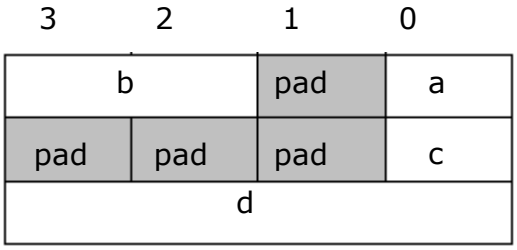
Type	Required Alignment
char, signed char, unsigned char	1
short, unsigned short	2
int, unsigned int, long, unsigned long	4
float	4
double	4
long long	4

As with most RISC processors, ARM code can access data objects most efficiently if they are properly aligned for their type (on their natural size boundary). For simple types, Table 6.1 above determines the required alignment.

To minimize padding, data use group variables of the same type together. This is the best way to ensure that as little padding data as possible is added by the compiler. For example,

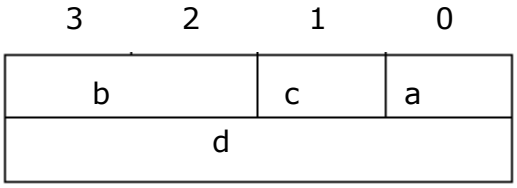
```
char a;
short b;
char c;
int d;
```

occupies 12 bytes, with 4 bytes of padding:



To improve the memory usage, the elements should be reorder as shown below, where it occupies only 8 bytes, without any padding

```
char a;
char c;
short b;
int d;
```



6.2.3 Function Argument Types

The *char* or *short* type function arguments and return values introduce extra casts. These increase code size and decrease performance. It is more efficient to use the *int* type for function arguments and return values, even if you are only passing an 8-bit value [22].

6.3 C LOOPING STRUCTURES

Loops are a common construct in most programs; a significant amount of the execution time is often spent in loops. It is therefore worthwhile to optimize time-critical loops.

6.3.1 Loops With Fixed and Variable Number of Iterations

Let's consider the same checksum example studied in Section 6.2.1, which shows how the compiler treats a loop with incrementing count i++.

It takes three instructions to implement for loop using up counter [22]:

- An ADD to increment the counter i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if i < 64

This is not efficient method. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result
- A conditional branch instruction

The following example shows the improvement of using decrementing loop rather than incrementing loop.

```
 int checksum(int *data)
{
        int i;
        int sum=0;

        for( i=64; i!=0; i--)
        {
                sum + = data[i];
        }
        return sum;
}
```

This compiles to,

```
checksum
        MOV     r2, r0                  ; r2 = data
        MOV     r0, #0                  ; sum = 0
        MOV     r1, #0                  ; i = 0
checksum_loop
        LDR     r3, [r2, r1, LSL, #2]   ; r3 = data[i]
        SUBS    r1, r1, #1              ; r1 = i - 1
        ADD     r0, r3, r0              ; sum + = r3
        BNE     checksum_loop           ; if (i != 0) goto loop
        MOV     pc, r14                 ; return sum
```

The SUBS and BNE instructions implement the loop.

Use do-while loops rather than for loops when you know the loop will iterate at least once. This saves the compiler checking to see if the loop count is zero.

6.3.2 Loop Unrolling

Small loops can be unrolled for higher performance, with the disadvantage of increased code size. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears. The ARM compilers currently do not unroll loops automatically, so any unrolling should be done in the source code [22].

The following code unrolls the packet checksum loop by four times, studied in Section 6.2.1. The number of words in the packet N is assumed to be multiple of four.

```
int checksum(int *data, unsigned int N)
{
        int sum=0;
```

```
        do
        {
                sum + = *(data++);
                sum + = *(data++);
                sum + = *(data++);
                sum + = *(data++);
                N -= 4;
        } while (N!=0);
        return sum;
}
```

This unrolling reduced the loop overhead from 4N cycles to (4N)/4 = N cycles. Only unroll important loops to reduce the loop overhead. Do not overunroll. If the loop overhead is small as a proportion of the total, then unrolling will increase code size and hurt the performance of the cache. Also, try to arrange that the numbers of elements in arrays are multiples of four and eight. You can then unroll loops easily by two, four, or eight times without worrying about the leftover array elements.

6.4 REGISTER ALLOCATION

The most important optimization supported by the ARM compilers is called register allocation. This is a process where the compiler allocates variables to ARM registers, rather than to memory. This has the advantage that those variables can be accessed quickly whenever needed, without needing instructions to transfer them from/to memory. As a result of register allocation, most variables are kept in registers, resulting in dramatic improvement in code size and performance.

When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled or swapped out variables since they are written out to memory [22]. Spilled variables are slow to access compared to variables allocated to registers. If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use. To implement a function efficiently, you

need to minimize the number of spilled variables and ensure that the most important and frequently accessed variables are stored in registers.

The C compiler can assign 14 variables to registers without spillage. Some compilers use a fixed register such as r12 for intermediate scratch working and do not assign variables to this register. Therefore, try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.

6.5 FUNCTION CALLS

The first four integer arguments are passed in the first four ARM registers: r0, r1, r2 and r3. Subsequent integer arguments are placed on the full descending stack, ascending in memory as shown in Figure 6.1. Function return integer values are passed in r0 [22].

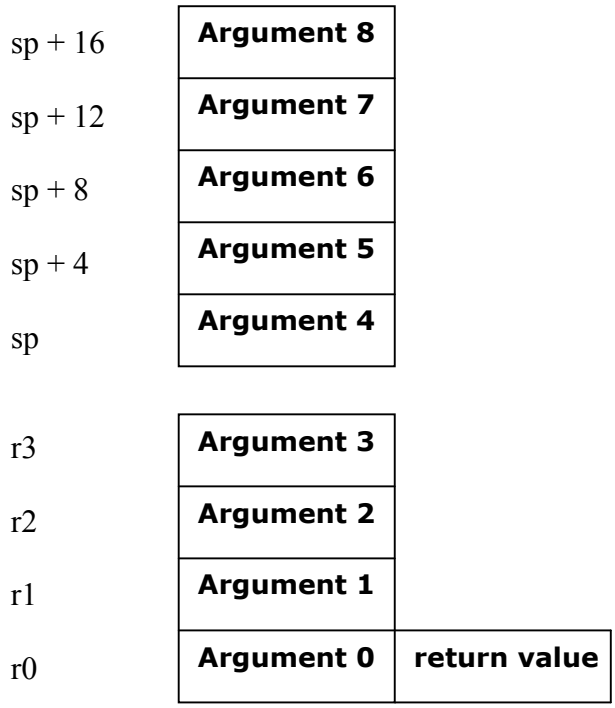


Figure 6.1: ARM Procedure Call Standard argument Passing

Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions

with more arguments, both the caller and callee must access the stack for some arguments.

Also, define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function.

6.6 DIVISION

The ARM does not have a divide instruction in hardware. Instead the compiler implements divisions by calling software routines in the C library. Depending on the numerator and denominator, a 32-bit division takes 20-140 cycles. The division function takes a constant time plus a time for each bit to divide [23]:

Time(numerator / denominator)

$= C0 + C1 * \log_2 (\text{numerator} / \text{denominator}) =$

$= C0 + C1 * (\log_2 (\text{numerator}) - \log_2(\text{denominator}))$.

As division is an expensive operation, it is desirable to avoid it where possible. Sometimes expressions can be rewritten so that a division becomes a multiplication. For example, (x / y) > z can be rewritten as x > (z * y) if it is known that y is positive and y * z fits in an integer.

If you can't avoid a division, then try to arrange that the numerator and denominator as unsigned integers. Signed division routines are slower since they take the absolute values of the numerator and denominator and then call the unsigned division routine.

Also, if the divisor in a division operation is a power of two, the compiler uses a shift to perform the division. Therefore try to always arrange, if possible, for scaling factors to be powers of two (for example, 128 rather than 100).

Therefore, this chapter explains different optimization techniques that would be useful for writing efficient C programming for ARM processors.

7 SOFTWARE POWER ESTIMATION

In order to estimate the power dissipated by the software executed on the ARM processor, Sim-Panalyzer [4] is used, an augmentation to the SimpleScalar performance simulator [A.1]. ARM binaries were produced using an ARM-Linux cross-compiler [A.2, A.3].

7.1 SIMPLESCALAR TOOL SET

The SimpleScalar tool set is a system software infrastructure used for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification. Through SimpleScalar, applications can be built and simulated on a range of processors. The type of simulator varies from fast functional simulators to detailed processor model simulators that are able to simulate caches, branch predictors, and many other features of modern processors. Since SimpleScalar can emulate the ARM instruction set, and because its reliability is in very high levels (in 2000 more than one third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs), it appears ideal for emulating the ARM processor in our case.

7.2 SIM-PANALYZER TOOL

The Sim-Panalyzer tool on the other hand is an infrastructure for micro-architectural power simulation. It is broken out into several components that model distinct parts of a computer: cache power models; datapath and execution unit power models; clock tree power models; and I/O power models. These power models can be configured into an augmented SimpleScalar simulator that will then produce power consumption figures. It is positioned above the "sim-outorder" component of the SimpleScalar simulator. The Sim-Panalyzer program contains components that model specific parts of the ARM processor. Sim-Panalyzer focuses efficiently on basic micro-architectural blocks and provides power information over a wide range of power dissipation sources, such as caches, clock trees, external I/O, on-chip memories and datapath and execution

blocks. For micro-architectural blocks, the basic method to calculate the power dissipation is by multiplying the appropriate switching capacitance by the number of micro-architectural accesses. For external I/O accesses, a transaction model counts the I/O pin switches in a cycle accurate way. Moreover, support for more sophisticated and accurate power models is provided through libraries, containing basic building blocks for the embedded logic simulator and ability to extract switching capacitance for CMOS gates. The logic simulator accumulates the switching capacitance of internal nodes into a switching capacitance estimation of each functional block's node [25].

7.3 ARM-LINUX CROSS-COMPILER

Since the target architecture for our experiments is the ARM architecture, the inputs to the Sim-Panalyzer tool for program emulation and power dissipation calculation must be ARM binaries. In this case, a cross-compiler kit targeting the ARM should be built on a Linux platform in order to acquire an ARM executable from C code. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the cross compiler runs.

The Crosstool [A.3] is a collection of scripts to build and test several versions of gcc and glibc for most architecture supported by glibc.

7.4 COMPILATION OF SIM-PANALYZER

Untar "sim-panalyzer-2.0.3.tar.gz" into your install directory. The source code can be downloaded from the website *http://www.eecs.umich.edu/~panalyzer.* Sim-Panalyzer has currently been compiled using gcc 3.2. Other gcc versions have not been tested thoroughly.

'make sim-panalyzer' generates a binary for the simulator. Go to the root directory for each version './Implementations/targetmachine/sim-panalyzer2.0.3' and execute this command. This should generate the executable file 'sim-panalyzer'.

7.5 HOW TO RUN THE SIMULATOR

Sim-Panalyzer has created a separate script file that parses the *cmd* file. The format for a *cmd* file is similar to a Microsoft Windows *ini* file. The configuration variables are divided into sections and are parse through these sections to generate an appropriate configuration for our simulator. An example of a *cmd* file is shown in Figure 7.1. Power configurations can be given as follows below.

```
[Component]
AIO
DIO
IL1 Cache
DL1 Cache
IL2 Cache
DL2 Cache
ITLB
DTLB
IRF
FPRF
Random Logic
Clock

[Global]
supply_voltage=1.8
frequency=200

[AIO]
frequency=200
IO_voltage=3.3
numberofbufferstages=5
microstrip length=10
external load=1

[DIO]
frequency=200
IO_voltage=3.3
numberofbufferstages=5
microstrip length=10
external load=1
```

Figure 7.1: Example of *cmd* file

The [Component] section that is shown in the beginning of Figure 7.1 represents the components that are intended for power analysis. Currently the components that are supported are Caches, Branch Target Buffers, Branch Predictors, Register files, Clock Trees & Random Logic. Based on the chosen components in

the [Component] section, the configuration variables are defined in the following subsections. For example, the [AIO], which configures the address IO pads, has the parameters "frequency" for the bus frequency, "IO_voltage" to describe the supply voltage for the IO pad, "Buffer ratio" for buffer sizing, "microstrip length" for modeling the PCB, and finally "external load" to model the load that is connected to this IO. "test_arm.cmd" is located in the source code, is the template command files that can be used as a reference.

It is important to note that in the *cmd* file, we assume capacitance to be in pF, time unit to be in ps, frequency to be in MHz, and voltage to be in V.

The power configurations are then integrated with the architectural configurations and create a single configuration file. Power configuration templates are also provided for these microprocessors in the "./cmd_files/" directory. The typical method for executing Sim-Panalyzer would be executing the gen_cfg_<target_machine>.pl script and then using the generated output file as the configuration file for Sim-Panalyzer.

$ gen_cfg_<target machine>.pl <architectural config filename> <PA cmd filename>
$ sim-panalyzer -config <configuration filename> <executing program> <program parameters>

7.6 ESTIMATION PROCEDURE

This Section explains the estimation procedure for a Sample Source Code. After building the cross compiler, the command line argument required in order to compile a C application (for example hello.c) for the ARM is the following:

$ arm-unknown-linux-gnu-g++ -static -o hello hello.c

After building the Sim-Panalyzer tool, the following command is outputting the power dissipation report of the hello.cpp application:

$ sim-panalyzer -config hello.cfg hello

The -config defines the name of the configuration file that contains architecture specific information for the ARM processor, such as the operating frequency, supply voltage etc. This configuration file is generated from a script provided by the Sim-Panalyzer tool, that parses a command file. The command file that we use in order to generate the configuration file is the default command file provided by Sim-Panalyzer.

8 RESULTS

8.1 EXPERIMENTAL RESULTS

In this chapter, the results based on the so far discussed theory and methodology of Chapters 3, 4, 5 and 6 are introduced. The power dissipation values for the software implementation are presented. These values were derived using the estimation procedure described in Chapter 7.

The average power consumed by a microprocessor while running a certain program is given by: $P = I \times V_{cc}$, where P is the average power, I is the average current and Vcc is the supply voltage. Since power is the rate at which energy is consumed, the energy consumed by a program is given by: $E = P \times T$, where T is the execution time of the program. This in turn is given by: $T = N \times \Upsilon$, where N is the number of clock cycles taken by the program and Υ is the clock period. In common usage, the terms power consumption and energy consumption are often interchanged. However, it is important to distinguish between the two when we talk of either of these in the context of programs running on wireless systems. Since wireless systems run on the limited energy available in a battery. Therefore, the energy consumed by the system or by the software running on it, determines the length of the battery life. Energy consumption is thus the focus of attention.

The Sim-Panalyzer tool output gives the average power dissipation for the components listed in the *cmd* file. The components for which power dissipation is given as follows:

Address Input-Output (AIO): AIO which configures the address IO pads, has the parameters "frequency" for the bus frequency, "IO_voltage" to describe the supply voltage for the IO pad, "Buffer ratio" for buffer sizing, "microstrip length" for modeling the PCB, and finally "external load" to model the load that is connected to this IO.

Data Input-Output (DIO): DIO, similar to AIO configures the data IO pads, has the parameters similar to AIO i.e. "frequency", "IO_voltage", "Buffer ratio", "microstrip length" and "external load".

Instruction Register File (IRF): IRF configures the instruction register bank where the parameter "Capacitance" is used.

Instruction Level 1 (IL1) Cache: IL1 cache configures the instruction level 1 cache that has the parameters "numberofbitlines" for the bit lines, "numberofwordlines" for the word lines in the cache and "Capacitance" that consumes energy in IL1 cache.

Data Level 1 (DL1) Cache: DL1 cache configures for data level 1 cache has parameters similar to IL1 cache i.e. "numberofbitlines", "numberofwordlines" and "Capacitance".

Instruction Table Look-aside Buffer (ITLB): ITLB configures parameters **"**numberofbitlines", "numberofwordlines" and "Capacitance".

Data Table Look-aside Buffer (DTLB): DTLB also configures parameters **"**numberofbitlines", "numberofwordlines" and "Capacitance".

Following tables gives the power and energy consumed by different protocols for different ARM processors. For each processor and each protocol, the table gives the average power consumed by the protocol across different components of the processor, using Sim-Panalyzer tool and also gives the number of instruction executed, the cycle per instruction, the clock frequency of the processor and the energy consumed by the processor for that protocol. The following three Sections gives the power and energy consumed by all the four protocols implemented on different processors.

8.2 POWER AND ENERGY CONSUMPTION ON STRONGARM-1110

Table 8.1: Average Power and Energy Consumption for DES on SA-1110

Components		Encryption		Decryption	
		Unoptimized	Optimized	Unoptimized	Optimized
Average Power Dissipation in	AIO	0.2519	0.2496	0.2583	0.2483
	DIO	1.0018	0.9926	0.9837	0.9980
	IRF	0.0322	0.0319	0.0322	0.0319
	IL1 Cache	0.3661	0.3723	0.3652	0.3715
	DL1 Cache	0.2153	0.2138	0.2150	0.2137
	ITLB	0.1487	0.1513	0.1484	0.1510
	DTLB	0.0874	0.0867	0.0873	0.0867
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0002	0.0002	0.0002	0.0002
	Micro-arch	0.6651	0.6624	0.6682	0.6676
Average Power (W)		**2.9962**	**2.9883**	**2.9860**	**2.9964**
Instruction Committed		171395	157737	171593	157833
Instruction Executed		283640	263840	284006	263800
Cycles Per Instruction		1.6711	1.7307	1.6506	1.7000
Clock Frequency (MHz)		206	206	206	206
Energy (mJ)		**6.89**	**6.62**	**6.79**	**6.52**

Table 8.1 gives the power and energy consumed by the DES protocol on Intel's StrongARM-1110 processor. By using the optimization techniques discussed in Section 3.2 for DES, gives the one version of DES implementation. Again by using the methodology explained in Chapter 6 a small reduction in the power consumption is achieved, but a significant change is achieved in the number of instruction executed. The use of decrementing loop, reduction in number of local variables, and properly assigning the data types to the variable helped to reduce this large number of instruction.

In both encryption and decryption, the power consumed at IL1 cache and ITLB power is increased. Also, in decryption power consumption is increased at DIO, while in all the other component power is decreased. The increase in power consumption in these components is due to repeated use of the some variables for data manipulation. With the use of optimization technique discussed the over all power and consumed by the protocol on this processor is reduced.

Table 8.2 gives the power and energy consumed by the 3DES protocol on SA-1110 processor. It uses the same methodology and optimization techniques used for DES algorithm.

In 3DES encryption, the power consumed is increased because the address and data input-output busses consumes more power, due to constant load and store instructions. However, there is a significant reduction in the number of instruction executed. This bottlenecks the ITLB performance and consumes more power. All over, the energy consumption is reduced which is the main focus.

Table 8.2: Average Power and Energy Consumption for 3DES on SA-1110

Components		Encryption		Decryption	
		Unoptimized	Optimized	Unoptimized	Optimized
Average Power Dissipation in	AIO	0.2431	0.2502	0.2427	0.2427
	DIO	0.9930	1.0292	0.9958	1.0372
	IRF	0.0325	0.0320	0.0325	0.0321
	IL1 Cache	0.3600	0.3743	0.3598	0.3744
	DL1 Cache	0.2226	0.2192	0.2224	0.2193
	ITLB	0.1460	0.1518	0.1459	0.1518
	DTLB	0.0902	0.0888	0.0901	0.0888
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0003	0.0003	0.0003	0.0003
	Micro-arch	0.7239	0.7201	0.7240	0.7203
Average Power (W)		**3.0391**	**3.0934**	**3.0410**	**3.0944**
Instruction Committed		462685	426999	463066	427276
Instruction Executed		774028	713464	774779	714164
Cycles Per Instruction		1.3959	1.4184	1.3958	1.4178
Clock Frequency (MHz)		206	206	206	206
Energy (mJ)		**15.94**	**15.19**	**15.96**	**15.20**

The power and energy consumed by the AES protocol is given in Table 8.3. There is a significant reduction in power and energy consumed by the algorithm. There is a little increase in the DL1 cache and DTLB power consumption but the overall power consumption is reduced. It uses the optimization techniques explained in Sec. 4.4 and the methodology of Chapter 6. There is also, a significant reduction in the number of instruction executed due to the optimization of MixColumn function.

As compared to DES and 3DES implementation on StrongARM-1110 processor, the AES takes less energy.

Table 8.3: Average Power and Energy Consumption for AES on SA-1110

Components		Encryption		Decryption	
		Unoptimized	Optimized	Unoptimized	Optimized
Average Power Dissipation in	AIO	0.2511	0.2514	0.2476	0.2438
	DIO	0.9790	0.9551	1.0368	0.9530
	IRF	0.0315	0.0317	0.0313	0.0313
	IL1 Cache	0.3906	0.3849	0.3932	0.3883
	DL1 Cache	0.2053	0.2069	0.2065	0.2070
	ITLB	0.1588	0.1567	0.1459	0.1580
	DTLB	0.0834	0.0840	0.0838	0.0841
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0002	0.0002	0.0002	0.0002
	Micro-arch	0.7195	0.6671	0.7673	0.6969
Average Power (W)		**3.0469**	**2.9655**	**3.1401**	**2.9901**
Instruction Committed		129659	92984	201698	112885
Instruction Executed		232507	167555	363983	203048
Cycles Per Instruction		1.7932	2.0595	1.6036	1.9188
Clock Frequency (MHz)		206	206	206	206
Energy (mJ)		**6.16**	**4.96**	**8.89**	**5.65**

Table 8.4: Average Power and Energy Consumption for RSA on SA-1110

Components		Encryption		Decryption	
		Unoptimized	Optimized	Unoptimized	Optimized
Average Power Dissipation in	AIO	0.2573	0.2473	0.2572	0.2495
	DIO	1.1045	0.9690	1.1044	0.9809
	IRF	0.0307	0.0313	0.0307	0.0311
	IL1 Cache	0.3743	0.3816	0.3742	0.3787
	DL1 Cache	0.2079	0.1993	0.2080	0.2023
	ITLB	0.1517	0.1558	0.1517	0.1543
	DTLB	0.0842	0.0811	0.0842	0.0822
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0004	0.0002	0.0004	0.0002
	Micro-arch	0.7740	0.5800	0.7736	0.6069
Average Power (W)		**3.2125**	**2.8731**	**3.2119**	**2.9136**
Instruction Committed		3824728	54038	3849896	77966
Instruction Executed		4572342	78236	4603642	106744
Cycles Per Instruction		0.9792	2.4261	0.9802	1.9726
Clock Frequency (MHz)		206	206	206	206
Energy (mJ)		**69.82**	**2.64**	**70.35**	**2.97**

Table 8.4 gives the power and energy consumption of RSA algorithm on StrongARM-1110 processor. There is a significant reduction in power and energy of the algorithm due to the efficient implementation of primality test algorithm explained in Sec. 5.2.1. This algorithm reduces the number of test to find the prime number; in effect it reduces a large number of instructions. Also, the methodology explained in Chapter 6 helped to reduce the power. Thus, RSA takes the minimum energy consumption on SA-1110, but with less security.

8.3 POWER AND ENERGY CONSUMPTION ON ARM7

Table 8.5 shows the power and energy consumption for DES on ARM7 processor. In DES encryption and decryption the power is reduced but the number of instruction executed for this processor is large as compared to SA-1110 because ARM7 is 3-stage pipeline architecture. So, the power consumption is less but the number of instructions to be executed is more and the clock frequency is less as compared to SA-1110 making it less attractive as compared to SA-1110. It also uses the optimization techniques explained in Sec. 3.2 and techniques of writing efficient C code, as discussed in Chapter 6.

Table 8.5: Average Power and Energy Consumption for DES on ARM7

Components		Encryption		Decryption	
		Unoptimized	Optimized	Unoptimized	Optimized
Average Power Dissipation in	AIO	0.2520	0.2523	0.2467	0.2439
	DIO	0.9792	0.9799	0.9855	0.9852
	IRF	0.0322	0.0318	0.0323	0.0318
	IL1 Cache	0.3727	0.3743	0.3727	0.3740
	DL1 Cache	0.2167	0.2138	0.2167	0.2142
	ITLB	0.1514	0.1521	0.1514	0.1520
	DTLB	0.0879	0.0867	0.0879	0.0869
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0003	0.0002	0.0003	0.0003
	Micro-arch	0.6891	0.6793	0.6894	0.6831
Average Power (W)		**3.0090**	**2.9979**	**3.0104**	**2.9989**
Instruction Committed		195348	182185	195733	182281
Instruction Executed		312790	288559	313549	288885
Cycles Per Instruction		1.5527	1.5968	1.5512	1.5751
Clock Frequency (MHz)		133	133	133	133
Energy (mJ)		**10.98**	**10.38**	**11.00**	**10.25**

3DES algorithm increases the security by increasing the key length but with this it also increases the power and energy consumption, which is undesirable for embedded devices. Table 8.6 shows the power and energy consumption of 3DES algorithm on the ARM7 processor. In this algorithm, it takes three keys to encrypt or decrypt the data, which increases the power consumption of the DIO component. Also, constant load and store instructions are executed that leads to more power consumption.

Table 8.6: Average Power and Energy Consumption for 3DES on ARM7

Components		Encryption		Decryption	
		Unoptimized	**Optimized**	**Unoptimized**	**Optimized**
Average Power Dissipation in	**AIO**	0.2436	0.2498	0.2465	0.2424
	DIO	0.9929	1.0098	0.9922	1.0044
	IRF	0.0324	0.0320	0.0324	0.0320
	IL1 Cache	0.3706	0.3723	0.3705	0.3722
	DL1 Cache	0.2221	0.2184	0.2220	0.2184
	ITLB	0.1503	0.1510	0.1502	0.1510
	DTLB	0.0900	0.0885	0.0900	0.0885
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0003	0.0003	0.0003	0.0003
	Micro-arch	0.7408	0.7364	0.7409	0.7363
Average Power (W)		**3.0705**	**3.0860**	**3.0725**	**3.0730**
Instruction Committed		541819	500300	542200	500579
Instruction Executed		862494	788882	863234	789583
Cycles Per Instruction		1.3107	1.3223	1.3104	1.3228
Clock Frequency (MHz)		133	133	133	133
Energy (mJ)		**26.09**	**24.20**	**26.13**	**24.13**

AES implementation gives the significant reduction in power and energy consumption as shown in Table 8.7. It uses the optimization techniques for MixColumn operation explained in Sec. 4.4 and also the software methodologies discussed in Chapter 6. There is significant reduction in power in DIO and the other micro-architecture components. Overall this reduces the power consumption and thus the energy consumption of AES protocol.

Table 8.7: Average Power and Energy Consumption for AES on ARM7

Components		Encryption		Decryption	
		Unoptimized	**Optimized**	**Unoptimized**	**Optimized**
Average Power Dissipation in	**AIO**	0.2498	0.2526	0.2462	0.2464
	DIO	0.9841	0.9669	1.0361	0.9590
	IRF	0.0315	0.0317	0.0313	0.0313
	IL1 Cache	0.3916	0.3858	0.3954	0.3896
	DL1 Cache	0.2045	0.2056	0.2054	0.2056
	ITLB	0.1592	0.1570	0.1605	0.1585
	DTLB	0.0830	0.0837	0.0833	0.0836
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0002	0.0002	0.0002	0.0002
	Micro-arch	0.7205	0.6681	0.7696	0.6983
Average Power (W)		**3.0519**	**2.9791**	**3.1555**	**3.0000**
Instruction Committed		130044	93070	202371	113079
Instruction Executed		234815	167393	364070	202886
Cycles Per Instruction		1.7880	2.0518	1.5984	1.9108
Clock Frequency (MHz)		133	133	133	133
Energy (mJ)		**9.63**	**7.69**	**13.80**	**8.74**

Table 8.8: Average Power and Energy Consumption for RSA on ARM7

Components		Encryption		Decryption	
		Unoptimized	Optimized	Unoptimized	Optimized
Average Power Dissipation in	AIO	0.2492	0.2550	0.2491	0.254
	DIO	1.0649	0.9517	1.0651	0.9718
	IRF	0.0307	0.0311	0.0308	0.0312
	IL1 Cache	0.3743	0.3805	0.3742	0.3781
	DL1 Cache	0.2079	0.2000	0.208	0.2027
	ITLB	0.1517	0.1554	0.1517	0.1541
	DTLB	0.0842	0.0811	0.0843	0.0822
	Clock	0.2275	0.2275	0.2275	0.2275
	ALU	0.0004	0.0002	0.0004	0.0002
	Micro-arch	0.7740	0.5761	0.7736	0.6078
Average Power (W)		**3.1648**	**2.8586**	**3.1647**	**2.9096**
Instruction Committed		3825188	52733	3850831	78892
Instruction Executed		4573292	76033	4606241	109608
Cycles Per Instruction		0.9792	2.4380	0.9804	1.9695
Clock Frequency (MHz)		133	133	133	133
Energy (mJ)		**106.56**	**3.98**	**107.45**	**4.72**

Table 8.8 gives the power and energy consumed during execution of a 32-bit RSA algorithm on ARM7 processor. The optimization technique explained in Sec. 5.2 reduces the power and energy consumption significantly on the processor. Further the use of methodology discussed in Chapter 6 helps to reduce energy. RSA takes the minimum energy on the ARM7 processor among the other implemented algorithms.

8.4 Power and Energy Consumption on ARM9TDMI

Table 8.9: Average Power and Energy Consumption for DES on ARM9TDMI

Components		Encryption				Decryption			
		Unoptimized		Optimized		Unoptimized		Optimized	
Average Power Dissipation in	AIO	0.2484		0.2423		0.2486		0.2404	
	DIO	0.9729		0.9793		0.9769		0.9750	
	IRF	0.0312		0.0308		0.0312		0.0308	
	IL1 Cache	0.3645		0.3740		0.3646		0.3736	
	DL1 Cache	0.2133		0.2102		0.2135		0.2105	
	ITLB	0.1482		0.1521		0.1482		0.1519	
	DTLB	0.0861		0.0851		0.0862		0.0852	
	Clock	0.2275		0.2275		0.2275		0.2275	
	ALU	0.0002		0.0002		0.0002		0.0002	
	Micro-arch	0.6674		0.6538		0.6678		0.6617	
Average Power (W)		**2.9597**		**2.9553**		**2.9647**		**2.9568**	
Instruction Committed		168303		156989		168696		157084	
Instruction Executed		285099		265065		285886		265428	
Cycles Per Instruction		1.7084		1.7960		1.7058		1.7476	
Clock Frequency (MHz)		150	200	150	200	150	200	150	200
Energy (mJ)		**9.61**	**7.20**	**9.37**	**7.03**	**9.63**	**7.22**	**9.14**	**6.85**

This Section discuss the outcome of power and energy consumption on ARM9TDMI. This processor is having a 5-stage pipeline architecture, which reduces the average Cycle Per Instruction on the processor. Two ARM9TDMI processors with different clock frequency are used for performance evaluation.

Table 8.9 and 8.10 gives the power and energy consumption on ARM9TDMI processor for DES and 3DES algorithm respectively. Both the implementation uses the same optimization technique explained in Sec. 3.2. It also uses the same coding methodology explains in the Chapter 6. In both the encryption the DIO, IL1 Cache and the ITLB component power is increased while in the both the decryption implementation IL1 Cache and ITLB component power is increased. This is due to the constant load and store instructions executed which leads to the bottleneck of the ITLB. But the important factor i.e. energy is reduced in both the cases.

Table 8.10: Average Power and Energy Consumption for 3DES on ARM9TDMI

Components		**Encryption**				**Decryption**			
		Unoptimized		**Optimized**		**Unoptimized**		**Optimized**	
Average Power Dissipation in	**AIO**	0.2511		0.2464		0.2538		0.2415	
	DIO	0.9833		0.9964		0.9589		0.9798	
	IRF	0.0320		0.0315		0.0320		0.0315	
	IL1 Cache	0.3586		0.3698		0.3583		0.3696	
	DL1 Cache	0.2204		0.2140		0.2205		0.2141	
	ITLB	0.1455		0.1501		0.1454		0.1501	
	DTLB	0.0890		0.0866		0.0890		0.0867	
	Clock	0.2275		0.2275		0.2275		0.2275	
	ALU	0.0003		0.0003		0.0003		0.0003	
	Micro-arch	0.7164		0.7131		0.7164		0.7131	
Average Power (W)		**3.0241**		**3.0357**		**3.0021**		**3.0142**	
Instruction Committed		461744		426017		462133		426300	
Instruction Executed		775444		714445		776219		715177	
Cycles Per Instruction		1.4223		1.4445		1.4222		1.4447	
Clock Frequency (MHz)		150	200	150	200	150	200	150	200
Energy (mJ)		**22.23**	**16.67**	**20.88**	**15.66**	**22.09**	**16.57**	**20.76**	**15.57**

Table 8.11 gives the power and energy consumption on the AES protocol on the ARM9TDMI processor. Here the power and energy are reduced significantly due to the MixColumn optimization and the C coding methodology for ARM as explained in Chapter 6. Since the number of instruction executed and the CPI is same for the two processors with different clock frequency, the one with low frequency consumes more energy as compared to other.

Table 8.11: Average Power and Energy Consumption for AES on ARM9TDMI

Components		Encryption				Decryption			
		Unoptimized		Optimized		Unoptimized		Optimized	
Average Power Dissipation in	AIO	0.2531		0.2531		0.2533		0.2466	
	DIO	0.9778		0.9922		1.0005		0.9791	
	IRF	0.0304		0.0303		0.0305		0.0302	
	IL1 Cache	0.3924		0.3871		0.3951		0.39	
	DL1 Cache	0.2027		0.2027		0.2044		0.2032	
	ITLB	0.1596		0.1574		0.1605		0.1586	
	DTLB	0.0819		0.082		0.0827		0.0822	
	Clock	0.2275		0.2275		0.2275		0.2275	
	ALU	0.0002		0.0002		0.0002		0.0002	
	Micro-arch	0.7159		0.6646		0.7663		0.6962	
Average Power (W)		**3.0415**		**2.9971**		**3.1210**		**3.0138**	
Instruction Committed		129353		92658		201008		112082	
Instruction Executed		238320		170594		366768		205306	
Cycles Per Instruction		1.8403		2.1325		1.6298		1.9705	
Clock Frequency (MHz)		150	200	150	200	150	200	150	200
Energy (mJ)		**8.89**	**6.66**	**7.26**	**5.45**	**12.43**	**9.32**	**8.12**	**6.09**

Table 8.12: Average Power and Energy Consumption for RSA on ARM9TDMI

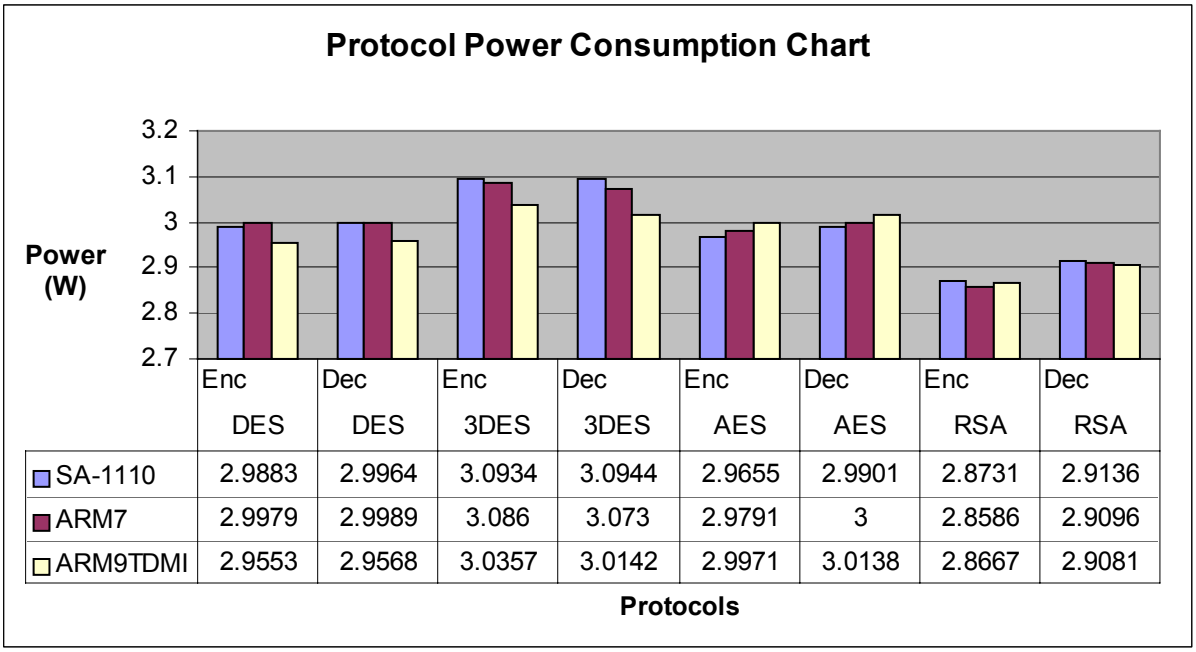
Components		Encryption				Decryption			
		Unoptimized		Optimized		Unoptimized		Optimized	
Average Power Dissipation in	AIO	0.2607		0.2549		0.2605		0.252	
	DIO	1.0168		0.9641		1.0169		0.9756	
	IRF	0.0307		0.0291		0.0307		0.0295	
	IL1 Cache	0.3744		0.3848		0.3744		0.3823	
	DL1 Cache	0.2077		0.1940		0.2078		0.1978	
	ITLB	0.1517		0.1571		0.1517		0.1559	
	DTLB	0.0841		0.0785		0.0842		0.08	
	Clock	0.2275		0.2275		0.2275		0.2275	
	ALU	0.0004		0.0002		0.0004		0.0002	
	Micro-arch	0.7737		0.5765		0.7731		0.6073	
Average Power (W)		**3.1277**		**2.8667**		**3.1272**		**2.9081**	
Instruction Committed		3824168		51773		3849274		77402	
Instruction Executed		4574627		77355		4605980		109254	
Cycles Per Instruction		0.9807		2.5649		0.9822		2.0470	
Clock Frequency (MHz)		150	200	150	200	150	200	150	200
Energy (mJ)		**93.54**	**70.15**	**3.79**	**2.84**	**94.31**	**70.73**	**4.33**	**3.25**

The RSA algorithm implementation gives significant results for power and energy consumption as shown in Table 8.12. The algorithm used for primality test introduced in Sec. 5.2.1 reduces significantly large number of instructions, which also helps to reduce power and energy consumption.

The RSA implementation takes the minimum energy among all other algorithm implemented on all the processor, but it has less security as compared to all other implementation.

9 CONCLUSION

The basic concepts, characteristics, and goals of various cryptographic algorithms are introduced in the thesis. The work have shown how embedded systems are essential parts of most communications systems and how this makes them especially attractive as a potential platform to implement cryptographic algorithms. In this work, a framework for analyzing the energy consumption of security protocols is presented. Asymmetric algorithms i.e. RSA have the lowest energy cost, and then come the symmetric algorithms. The energy cost of asymmetric algorithms is very much dependent on the key size, while that of symmetric algorithms is not affected to the same extent by the key size. The cost of symmetric algorithms mainly depends on key expansion and encryption/decryption cost. Here the RSA implemented is only 32-bit and therefore consumes less energy. There is a wide variation in the energy costs within the same family of cryptographic algorithms.

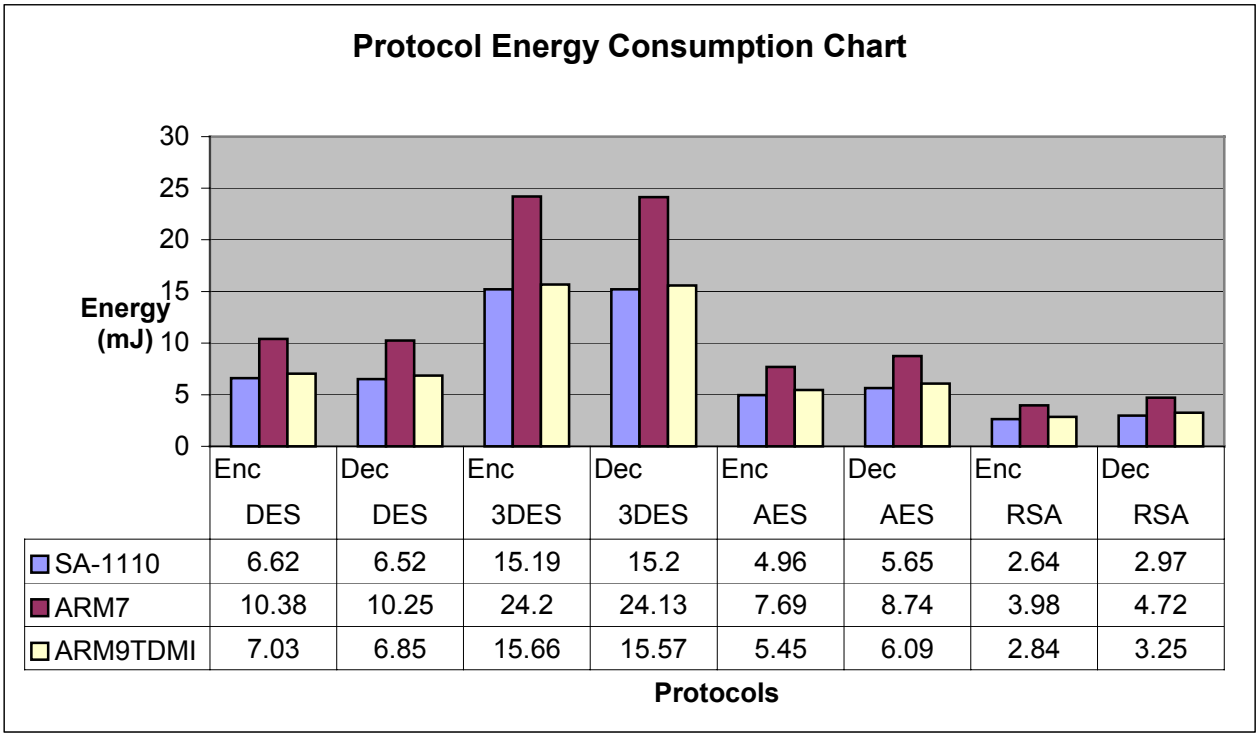


	Enc DES	Dec DES	Enc 3DES	Dec 3DES	Enc AES	Dec AES	Enc RSA	Dec RSA
SA-1110	2.9883	2.9964	3.0934	3.0944	2.9655	2.9901	2.8731	2.9136
ARM7	2.9979	2.9989	3.086	3.073	2.9791	3	2.8586	2.9096
ARM9TDMI	2.9553	2.9568	3.0357	3.0142	2.9971	3.0138	2.8667	2.9081

Figure 9.1: Power Consumption By Different Protocols on Different Processors

Figure 9.1 shows the power consumed by different protocols on different processors. It shows, that if only power consumption is focused than the RSA implementation with the ARM7 processor would be the best-suited choice for security protocol implementation, as it has the lowest power consumption. But the RSA implementation is only 32-bit as compared to 64-bit DES, 192-bit 3DES

and the 128-bit AES implementation, which weakens the security if implemented with RSA. But, the focus is on the energy consumption as the target machine is wireless device.



	Enc DES	Dec DES	Enc 3DES	Dec 3DES	Enc AES	Dec AES	Enc RSA	Dec RSA
SA-1110	6.62	6.52	15.19	15.2	4.96	5.65	2.64	2.97
ARM7	10.38	10.25	24.2	24.13	7.69	8.74	3.98	4.72
ARM9TDMI	7.03	6.85	15.66	15.57	5.45	6.09	2.84	3.25

Figure 9.2: Energy Consumption By Different Protocols on Different Processors

Figure 9.2 shows the energy consumed by different protocols on different processors. The figure shows that RSA consumes very less energy among all the protocols implemented. The energy cost of StrongARM-1110 processor is very low with RSA protocol. But the security, algorithm cannot be implemented with RSA algorithm, as it is only 32—bit implementation which is easily vulnerable by the attackers. Also, energy is consumed in key management i.e. transferring the public keys between the communicating devices. So the best choice for implementing security protocol in wireless embedded device is AES algorithm as it has 128-bit key length, which is a strong key and also process 128-bit of data. RSA can be used in combination with AES to implement security protocol where RSA can be used to transfer the symmetric keys between the devices. Future research needs to be done so that these protocol make be more optimized that can help in low energy cost.

REFERENCES

1. Thomas Wollinger, Jorge Guajardo, and Christof Paar, "Cryptography in Embedded Systems: An Overview," *Proceedings of the Embedded World 2003 Exhibition and Conference*, pp. 735-744, Design & Elektronik, Nuernberg, Germany, February 18-20, 2003.

2. D. W. Carman, P. S. Kruus, and B. J. Matt, "*Constraints and Approaches for Distributed Sensor Security*," Network Associates Labs Tech. Rep. 00-010, 2000.

3. W. Freeman and E. Miller, "An experimental analysis of cryptographic overhead in performance-critical systems," in *Proceedings International Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 348–357, Oct. 1999.

4. S. K. Miller, "Facing the challenges of wireless security," *IEEE Computer*, pp. 46–48, July 2001.

5. D. S. Wong, H. H. Fuentes, and A. H. Chan, "The performance measurement of cryptographic primitives on Palm devices," in *Proceedings Annual Computer Security Applications Conference*, pp. 92–101, Dec. 2001.

6. S. Ravi, A. Raghunathan, and N. Potlapally, "Securing wireless data: System architecture challenges," in *Proceedings International Symp. System Synthesis*, pp. 195–200, Oct 2002.

7. A. Hodjat and I. Verbauwhede, "The Energy cost of secrets in ad-hoc networks," *in Proceedings IEEE CAS Workshop. Wireless Communication and Networking*, Sept. 2002.

8. M. Jakobsson and D. Pointcheval, "Mutual authentication for low-power mobile devices," in *Proc. Financial Cryptography*, pp. 178–195, Feb. 2001.

9. D. S. Wong and A. H. Chan, "Mutual authentication and key exchange for low power wireless communications," in *Proc. IEEE MILCOM Conf*., pp. 39–43, Oct. 2001.

10. Y. W. Law, S. Dulman, S. Etalle, and P. J. M. Havinga, "*Assessing Security-Critical Energy-Efficient Sensor Networks,*" Univ. of Twente, The Netherlands, Tech. Rep. TR-CTIT-02-18, July 2002.

11. R. Karri and P. Mishra, "Minimizing energy consumption of secure wireless session with QoS constraints," in *Proc. Int. Conf. Communications*, pp. 2053–2057, May 2002.

12. S. Ravi, A. Raghunathan, P. Kocher and S. Hattangady, "Security in Embedded Systems: Design Challenges" in *ACM Transactions on Embedded Computing Systems: Special Issue on Embedded Systems and Security,* 2004.

13. P. Kocher, R. Lee, G. McGraw, A. Raghunathan, S. Ravi, Security as a New Dimension in Embedded System Design, *DAC 2004*, June 2004.

14. E. P. Harris, S. W. Depp, W. E. Pence, S. Kirkpatrick, M. Sri-Jayantha, and R. R.Troutman, "Technology directions for portable computers," in *Proc. IEEE*, vol. 83, Apr. 1995, pp. 636–657.

15. V. Tiwari, S. Malik, A. Wolfe, and M. T. C. Lee, "Instruction level power analysis and optimization of software," *Journal of VLSI Signal Processing*, 1996.

16. Data Encryption Standard, *National Institute of Standard and Technology (U.S.), FIPS PUB 46-3*, 25 Oct 1999.

17. Bruce Schneier, *Applied Cryptography*, Second Edition, John Wiley & Sons, 1996.

18. Sung-ho Jee and Paul Montague, "A secure DES implementation for the real time embedded application," *1999 Third International Conference on Knowledge-Based Intelligent Information Engineering Systems*, Dec 1999 pp. 496 – 500.

19. William C. Barker, Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher, *National Institute of Standard and Technology (U.S.)*, May 2004.

20. Advanced Encryption Standard (AES)*, National Institute of Standard and Technology (U.S.),* FIPS PUB 197, November 26, 2001.

21. Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin, "Efficient Software Implementation of AES on 32-Bit Platforms," *4th International Workshop on Cryptographic Hardware and Embedded Systems*, 2002, pp. 159 - 171

22. Andrew N. Sloss, Dominic Symes, and Chris Wright, "*ARM System Developer's Guide: Designing and Optimizing System Software,"* Elsevier, 2004.

23. Writing Efficient C for ARM, *ARM DAI 0034A*, January 1998.

24. Using C Global Data, ARM DAI 0036B, January 1998.

25. Sim-Panalyzer, The SimpleScalar-Arm Power Modeling Project. [Online]. Available:http://www.eecs.umich.edu/˜panalyzer/

26. B. Gladman, "*A Specification for Rijndael, the AES Algorithm*", Available at http://fp.gladman.plus.com, May 2002.

Appendix A LIST OF USEFUL WEBSITES

A.1. SimpleScalar Tool Set. [Online].
Available:http://www.simplescalar.com.

A.2. Cross Compiler ToolChain [Online].
Available:http://www.amidasimputer.com/developer/downloads/

A.3. CrossTool-0.43 [Online].
Available:http://kegel.com/crosstool/#download