Implementation of security algorithms for wireless sensor networks using reconfigurable devices

by

Konstantinos Papadopoulos, B.Sc.

THESIS

Presented to the Faculty of the Graduate School of

The Technical University of Crete at Chania

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

THE TECHNICAL UNIVERSITY OF CRETE AT CHANIA

October 2009

Implementation of security algorithms for wireless sensor networks using reconfigurable devices

APPROVED BY

SUPERVISING COMMITTEE:

Ioannis Papaefstathiou, Supervisor

Apostolos Dollas

Dionisios Pnevmatikatos

Implementation of security algorithms for wireless sensor networks using reconfigurable devices

Konstantinos Papadopoulos, M.Sc.
The Technical University of Crete at Chania, 2009

Supervisor: Ioannis Papaefstathiou

Wireless sensor networks (WSNs) are quickly becoming a vital part of our infrastructure. Since their integrated sensor nodes are very compact and wireless, they are highly energy constrained. In general, energy-efficiency is a key concern in WSNs. The large number of sensor nodes involved in such networks and the need to operate over a long period of time require careful management of the energy resources. At the same time, security is a critical factor in numerous ultra low-power WSN applications. However, these tiny, pervasive computing devices have extremely limited resources and computational capabilities. Thus, security engineers face the seemingly contradictory challenge of providing lightweight algorithms for strong encryption and other cryptographic services that can perform on a speck of dust.
Nowadays, using contemporary low-cost reprogrammble field-programmable gate array (FPGA) technology enables us to improve the performance of WSN nodes. FPGAs are semiconductor devices that can be configured by the customer or designer after manufacturing and they perform certain CPU intensive tasks more efficiently than the general-purpose CPUs. Furthermore, FPGA manufacturers have constructed small Complex Programmable Logic Devices (CPLDs) which are programmable logic devices with architectural features of Programmable Logic Arrays (PLAs) and FPGAs. The main characteristics of the CPLDs are their very low energy consumption and the relative high frequency rate when executing certain data manipulation tasks.
The purpose of this thesis is to use FPGAs and CPLDs in different kinds of nodes (i.e., sensor nodes or base stations) within a WSN environment in order to improve

their performance. Especially, a CPLD is an ideal addition in a sensor node platform due to its characteristics resulting a decrease of its overall energy consumption. Moreover, the base station of our platform is expanded using a more powerful FPGA for increasing the nodes processing power (thus enabling the implementation of more complex functions). Different ciphers are implemented on such nodes in order to provide the high level of security expected. Skipjack and Blowfish algorithms were selected to secure sensor nodes while a stronger cipher such AES-128 is the algorithm implemented on our base station. Finally, base stations are usually a common target of attackers while many of the aforementioned cryptographic schemes are vulnerable to sophisticated attacks such differential power analysis. Consequently, we develop a hardware implementation of the AES-128 algorithm following a specific methology which leads us to a well-protected base station system from such attacks.

Acknowledgments

First of all, I would like to thank my supervisor, Assistant Professor Papaefstathiou, for the excellent working atmosphere and the trust and freedom he granted me for my research. Furthermore, I am grateful to Professor Pnevmatikatos and Professor Dollas who agreed to evaluate this thesis.
I would also like to mention all my colleagues in Microprocessor and Hardware Laboratory (MHL) who have greatly helped me to accomplish this work. I am eternally grateful to Ph.D. student Dimitrios Meintanis for his great help in learning the tools needed for carrying out the specific measurements and Georgios-Grigorios Mplemenos for his help during the working period of this work.
I would like to dedicate this work to my mother, who left us too early, and my family for their ever-lasting support of my work and ideas and for getting me to the stage where I could attempt it. Finally, I would also like to express my gratitude to Magda for putting up with me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This initial chapter provides some introductory information about the wireless sensor networks and the motivation for conducting research in security in WSNs, summarizes the scientific contribution of the work and describes the structure of this thesis.

1.1 Motivation

Wireless sensor networks (WSNs) are quickly becoming a vital part of our infrastructure; applications of massively distributed sensor networks include seismic, acoustic, medical and intelligence data gathering as well as climate, equipment monitoring etc. Since these integrated sensor nodes are very compact and wireless, they are highly energy constrained.

In general, energy-efficiency is a key concern in WSNs. The large number of sensor nodes involved in such networks and the need to operate over a long period of time require careful management of the energy resources. In addition, wireless communication is a major source of power consumption. Furthermore, replacing batteries on thousands of WSN nodes may well become infeasible. Hence, it is well accepted that one of the key challenges in unlocking the potential of such data gathering sensor networks is conserving energy so as to maximize their post-deployment active sensing lifetime [1].

At the same time, security is a critical factor in numerous ultra low-power WSN applications. However, these tiny, pervasive computing devices have extremely limited resources and computational capabilities. Thus, security engineers face the seemingly contradictory challenge of providing lightweight algorithms for strong encryption and other cryptographic services that can perform on a speck of dust.

Moving to a different sector, Field-Programmable Gate Arrays (FPGAs) are semiconductor devices that can be configured by the customer or designer after manufacturing and they perform certain CPU intensive tasks more efficiently than the

general-purpose CPUs. FPGAs contain programmable logic components called *logic blocks*, and a hierarchy of reconfigurable interconnections that allow the blocks to be *wired together*. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complex memory structures [2] [3] [4].
Furthermore, Field Programmable Gate Array (FPGA) manufacturers have constructed small Complex Programmable Logic Devices (CPLDs) which are programmable logic devices with architectural features of Programmable Logic Arrays (PLAs) and FPGAs. The main characteristics of the CPLDs are a) their very low energy consumption, b) the relative high frequency rate when executing certain data manipulation tasks and c) their low cost (less than $10). On the other hand, the main disadvantage of those devices is their small number of resources allowing them to execute only relatively small, yet very CPU intensive, tasks.

1.2 Scientific contribution

As this thesis demonstrates, we can use FPGAs and CPLDs in different kinds of nodes (i.e., sensor nodes or base stations) within a WSN environment in order to improve their performance. Especially, a CPLD is an ideal addition in a sensor node platform due to its main characteristics resulting a decrease of its overall energy consumption. Moreover, the base station of our platform is expanded using a more powerful FPGA for increasing the nodes processing power (thus enabling the implementation of more complex functions).
As regards the high level of security that must be provided by such nodes, we choose to implement different cipher encryption schemes in these reconfigurable devices. Especially, a mini version of Skipjack and Blowfish are selected to be implemented in sensor node CPLD taking into account its lower capabilities and needs of security compared to the base station in the FPGA of which a stronger scheme such AES-128 is implemented.
Furthermore, base stations are usually a common target of attackers while many of the aforementioned cryptographic schemes are vulnerable to sophisticated attacks such *differential power analysis*. Consequently, we develop a hardware implementation of the AES-128 algorithm following a specific methology which leads us to a well-protected base station system from such attacks.

1.3 Remainder

The remainder of this thesis is organized as follows:

- Chapter 2 presents the theoretical background of this work including the possible threats that may be used from an attacker in order to establish potential vulnerabilities in different networks, the existing cipher schemes which can give solutions to several of the previously referred attacks and the *dual-rail* method that provides a way to avoid sophisticated attacks such as *differential power analysis*.
- Chapter 3 describes in detail the three different cipher encryption schemes that are selected to be implemented in this work.
- Chapter 4 illustrates the specific architectures which lead us to low energy consumption and high performance cipher encryption and their hardware implementations provide us with the desired results.
- Finally, chapters 5, 6 and 7 point out timing, energy and power consumption results derived from the measurements in our real-world experiments, conclusions and future work on cipher encryption in WSN nodes with reconfigurable devices.

Chapter 2

Theoretical background

Embedded Peer-to-Peer (EP2P) systems introduce a new challenge in the development of distributed systems. These systems have brought about an important revolution in distributed computing paradigms, now that the roles of client and server, which are the basis of the most widely used distributed computation models, are disappearing. The new scenario consists of systems in which all the elements of the network are symmetrical and in most cases, the mechanisms of communication are not based on pre-existing infrastructures, but rather on dynamic ad-hoc networks among peers. At the same time, the recent technological advances in short distance wireless communications have opened up new areas of application which represent important technological challenges. In addition, these systems are extremely vulnerable against internal and external attacks due to resource constraints, lack of tamper-resistant packaging and the nature of open and public communication channels.

The most typical representative of EP2P systems is Wireless Sensor Network (WSN) as it features the most important characteristics of EP2P systems, i.e., heterogeneity, resource constraints of devices and P2P network communication. Therefore, in order to present a coherent view on the EP2P threats, we focus on the context of wireless sensor networks.

WSNs are quickly gaining popularity due to the fact that they are potentially low cost solutions to a variety of real-world challenges [5]. This provides a means to deploy large sensor arrays in a variety of conditions capable of performing both military and civilian tasks. However, security in WSNs poses different challenges than traditional network/computer security due to inherent constraints of resources (computing, communication and storage).

2.1 Threats

In this section, the according threats, most of them revolving around WSN, are presented in order to establish the potential vulnerabilities of EP2P systems. These threats are categorized as follows: denial of service attack (DoS), node compromise, attack on data aggression, impersonation attack, side-channel analysis and other common attacks.

2.1.1 Denial of service attack

The denial-of-service (DoS) attacks have been regarded as serious security threats against the Internet in general [6]. The EP2P system could not be an exception.
Normally, the denial-of-service (DoS) attacks [7] involve three parameters: users, a shared service or resource and a maximum waiting time. During a denial-of-service attack, a user is made to wait longer than the predefined maximum waiting time by a malicious user for the use of shared service(s) or resource(s). This is a very general and straight forward definition of DoS attack. A more specific definition of DoS for WSN is as follows; the result of any action that prevents any part of WSN from functioning correctly or in a timely manner.
Some of the existing DoS attacks are described in the following subsections.

2.1.1.1 Spoofed or altered routing information

By spoofing the routing information exchanged among nodes, the adversaries want to cripple the sensor network by creating routing loops, partitioning the network, increasing end-to-end latency, etc.

2.1.1.2 Selective forwarding

In a WSN, the motes double up as a router and forward the messages faithfully. A black hole is created when a malicious node refuses to forward any messages it received. However, it is quite easy to detect this simple attack and neighboring nodes exclude the malicious node from the routing path. A more refined form of this attack is selective forwarding attack, where a malicious node selectively forwards messages depending on some criteria. This diminishes the probability of the

attack detection by neighboring nodes.

2.1.1.3 Sinkhole attack

Any network, where the predominant communication pattern is many-to-one, is susceptible to sinkhole attacks. In a sinkhole attack, the adversary attracts all the traffic of a particular area through a malicious node with false routing information and, then, tampers with the messages passing through it. Mounting a sinkhole attack is particularly easy due to the nature of communication in WSN where most of the packets share the same final destination, namely the base station. Thus, a compromised node only needs to declare a high quality link to the base station, which may lure a large number of nodes to send packets through the compromised node.

2.1.1.4 Sybil attack

In a Sybil attack [8] [9], a single node takes on multiple identities to deceive other nodes. It can reduce effectiveness of a fault-tolerant system which deploys resources redundantly. It can also affect the functioning of geographic routing protocols [10] [11].

2.1.1.5 Wormhole attack

In the wormhole attack presented in [12], an attacker captures message bits at one location and replays them in another location. A typical wormhole attack involves two distant compromised nodes that falsely understate their distance using some high bandwidth channel available to them only. Well placed wormholes can significantly alter the routing paths to their advantage. Most of the wormhole attacks are mounted in combination with selective forwarding, Sybil attack or eavesdropping.

2.1.1.6 Hello flood attack

This is a very simple attack which can cripple an entire sensor network. Nodes send HELLO packets to their neighbors, which are specifically susceptible to this attack, during network setup or neighborhood discovery. Nodes which receive

these HELLO packets consider the other node to be their neighbor being within the normal radio range. However, this can be false as a laptop-class attacker with strong transmission power can send HELLO packets to the entire network and advertise a very good link to the base station. Many motes may consider the adversary as their neighbor but nodes far away from the attacker may actually send their packets into oblivion. Thus, the network performance will degrade considerably.

2.1.1.7 Acknowledgment spoofing

It is a technique used by an adversary to mount attacks on those networks where routing schemes use link-layer acknowledgments to decide the link reliability. Adversary may be able to convince nodes that a weak link is strong or reinforce a dead link by acknowledgment spoofing. Consequently, messages sent via these routes are lost. This attack can also be used to mount several other attacks.

2.1.2 Node compromise

An embedded device is considered being compromised when an attacker, through various means, gains control or access to the device (node) itself after it is being deployed [13]. Using the compromised nodes, an attacker can easily manipulate the nodes to create denial of service (DoS) attacks on the EP2P system or to inject false data into the network. When a node is compromised, the attacker is able to retrieve critical information, such as the security keys used for securing the communication or information pertaining to the routing protocols. Using the retrieved information, the attacker will be able to eavesdrop on the communication data or launch other malicious attacks on the EP2P system.
There are several attacks from compromised nodes, some of which are presented below.

2.1.2.1 Eavesdropping

Since an attacker is able to retrieve the security keys from the node, he is able to eavesdrop on the on-going traffic and sniff out important information that is sent across the network. Depending on the type of security protocol, the key is used in a specific part of the network could become subject to eavesdropping. For example, if the compromised key is the network key, the entire network will not be

secured since the attacker will be able to decrypt all the encrypted messages passing through the network.

2.1.2.2 Node replication

With the capturing of the node, the attacker is able to replicate the node at different locations since the attacker has the nodes program flash, EEPROM and SRAM images. The attacker is able to confuse the EP2P system by deploying replicated nodes throughout the network. These replicated nodes can also be used to create DoS attacks or confuse the network protocols that perform data aggregation, voting, routing and so on, by injecting false data into the network. Using more compromised nodes in the network enables the attacker to obtain greater control over the EP2P system.

2.1.2.3 Node masquerading

Using the compromised security keys and other information, an attacker can masquerade as the legitimate node using a more powerful device such as a laptop. This increase in the capability of the embedded device allows the attacker to launch attacks that are more computational intensive. Such computational intensive attacks are not possible to be launched from the embedded device itself due to its limited resources, such as its processing power. Using a more powerful embedded device, the attacker can also disrupt the routing protocol by sending out false routing information. For example, the embedded device can now reach the control station within 1 hop, instead of 4 hops, through the use of a more powerful antenna. This will result in the neighboring nodes directing their traffic towards the malicious embedded device due to its shorter route to the control station. Hence, the attacker can perform attacks such as selective forwarding and modification of the data to inject false data into the network.

2.1.2.4 False injection of data

Since the attacker has the control of the compromised nodes, he is able to modify its code to perform false data injection to the EP2P system. These false data will mislead other nodes and also create false alarms, which will result in the wastage of the embedded devices valuable resources in reaction to the false data.

Examples of the different types of false data injection include the embedded devices reading, aggregation result, routing routes and reporting of bad neighbor nodes.

2.1.3 Attack on data aggregation

In the general network setting, there is a number of nodes communicating with one or more base stations. This setting can be viewed as an event-based system where the base stations act as sinks, which subscribe to specific data streams by expressing interest and queries, and the nodes act as sources to report environmental events to the sink. The sink is often assumed to be powerful enough to perform computationally intensive cryptographic operations while the nodes have constrained resources in terms of computation, memory and battery power. The networking among nodes is usually assumed to be highly ad-hoc in a sense that the network topology may change rapidly and unpredictably.
Among the sensor nodes, there can be some special nodes called aggregators that conduct some computational operations on the data from their children nodes, for example, taking the sum, average, maximum or minimum of the data [14]. The resulting data from the aggregators are forwarded to the sinks.
There are two main security threats to the secure data aggregation. One is *eavesdropping* which an attacker uses to obtain information on the transmitted data between sensing nodes (non-aggregators), between aggregators, and between aggregators and sink. This causes a great damage especially when the data aggregated are of critical importance.
The other threat is *forging* which makes it possible to an attacker to alter the aggregated data or other related information in such a way that invalid aggregated data are accepted as correct or vice versa.

2.1.4 Impersonation attack

In the EP2P system, *impersonation attack* means that a malicious embedded device impersonates a legitimate one. The attacks described in this section can be divided into impersonation attacks and attacks that result from impersonation attacks. Several classes of attacks such node replication and the Sybil attack can be categorized as impersonation attacks. Some of the existing attacks conducted by impersonated nodes are described to the next subsections.

2.1.4.1 Sybil attack

In a Sybil attack, a single node takes on multiple identities to deceive other nodes. A node that wishes to conduct the Sybil attack can adopt a new identity by creating a new identity or stealing the identity of an existing node. It is difficult by a node to create a new identity as there are several schemes in place that will detect unknown identities, but it is possible by a Sybil node to masquerade as a single node directly communicating with other nodes or an aggregator node pretending to represent a number of identities that do not actually exist. Sybil attacks can disrupt several of the functions that may be conducted on a network including data aggregation, voting, routing and fair resource allocation.

2.1.4.2 Node replication

Node or identity replication is the simple duplication of nodes. As nodes tend to be physically unprotected, it is feasible by an attacker to capture, replicate and insert duplicate nodes back into selected regions of the network. Node replication is different from a Sybil attack, because the multiple nodes are duplicates and basically have the same identities. A Sybil attack is more sophisticated, because new identities must be adopted. Node replication attacks should not be ignored because of their simple nature as large networks cannot easily verify every identity. If identities remain unchecked a node replication attack can achieve the same effects as the Sybil attack disrupting data aggregation and threshold voting schemes.

2.1.4.3 Eavesdropping

Eavesdropping is an attack in which the malicious attacker uses the impersonated node to obtain data. Attackers who impersonate cluster heads or intermediate nodes are able to gain access to data generated by sections of the network. If an attacker is able to impersonate a base station or other privileged node, the attacker can query the database for all information obtained by the network. Eavesdropping is an insidious attack as it does not alter the network in respect to its original purpose. Eavesdropping can be mitigated by the use of encryption to prevent the unauthorized access of data. However, if an attacker is able to obtain authentication keys, it will be likely that encryption keys will have also been compromised.

2.1.4.4 Message injection

The message injection attack can be effectively used to mislead the data aggregation algorithm. A typical scenario is that of an attacker impersonating a number of nodes in a particular part of the network. All impersonated nodes can be made to send false data reports about an event to the original network. This may mislead the data aggregation algorithm in the original network if the number of impersonated nodes is larger than the number of valid nodes. If an attacker is able to impersonate a cluster head, aggregator or forwarder in a hierarchical network, only one node will be needed to be impersonated. Message authentication mitigates this attack as the original network is able to detect the messages generated by impersonated nodes.

2.1.4.5 Wormhole attack

In the original wormhole attack, an attacker records messages from one part of the network and replays them in another one. Attackers use a low-latency link or out-of-bound channel to move the messages from one part of the network to the other so that the copied message could appear at an aggregator node at the same time. The original message can be encrypted and the attacker may not be aware about the content of the message. An impersonation attack on the remote node before conducting the wormhole attack will help the receiver believe that the wormhole data is legitimate. Finally, wormhole attacks can be used to conduct sinkhole attacks.

2.1.5 Side-channel analysis

In order to mitigate the threats to the network, cryptographic algorithms are implemented on the devices. While the algorithms themselves can be deemed secure from a mathematical standpoint, so-called *implementation attacks* allow to extract secret keys from the devices, which would negate any gained security. A very powerful class of implementation attacks are *side-channel analysis* attacks. In such an attack, the adversary monitors certain physical properties (the side-channel, which can be, e.g., execution time, power consumption or electromagnetic emanation) of a device while it performs some cryptographic operation. If the recorded physical values are influenced in some way by the processed secret key, the attacker

can extract information about this key or even reveal it completely.
In the last decade, a considerable number of side-channel analysis (SCA) attacks have been published. In principle, any kind of information leaking from a cryptographic device can be exploited by an attacker.

2.1.5.1 Timing attacks

Timing attacks exploit variations in execution time, which are dependent on the secret key. Their first publication was by Kocher in 1996 [15] and they were among the first side-channel attacks to be proposed. Such attacks have even been shown to be feasible on Internet servers which execute cryptographic algorithms [16]. In the last years, there have been interesting proposals for timing attacks on software implementations of cryptographic algorithms on processors with cache [17] [18] [19]. Processors with advanced micro-architectural features like hyper-threading, have been shown to be especially vulnerable [20].
The execution time of cryptographic algorithms often shows slight differences dependent on the input of the algorithm. This data-dependent variation is due to performance optimization, conditional statements, handling of special cases, cache misses and a variety of other causes. Since an adversary can easily measure the execution time of a tamper-proof device like a smart card with high accuracy, the dependence between public inputs, secret data hidden in the device and changes in execution time can be used to derive valuable information about the secret data [15] [21] [22] [23] [24].

2.1.5.2 Power-analysis attacks

Power-analysis attacks work by measuring the power consumption of a device while it performs a cryptographic operation. Measurement is usually conducted with a digital oscilloscope connected to a sensing resistor or current probe on the device power lines. The recorded course of the voltage (which is proportional to the current and thus to the power consumption) during a single cryptographic operation is commonly denoted as *power trace*.
The original publication by Kocher et al. [25] already pointed out different methods for exploiting the obtained power traces: *Simple power analysis (SPA)* and the much more powerful *differential power analysis (DPA)*. SPA [26] exploits the fact that different operations and different data values processed by a device also have

different power consumption characteristics. In an SPA attack, single power traces are used to look for such distinguishing features. With some knowledge of the structure and operation of the device, it can be possible to find out information about the processed secret key. In the past, SPA attacks have been successful mainly on unprotected implementations of public-key algorithms on smart cards.
DPA attacks [25] [27] use larger numbers of power traces and normally require less knowledge about the cryptographic device than SPA attacks. For each cryptographic operation corresponding to a power trace, the respective plaintext or ciphertext needs to be known. Then, a small portion of the used key is guessed (key hypothesis) and an intermediate value of the cryptographic algorithm, which depends both on this portion of the key and the known plaintext or ciphertext calculated. A suited power model (often based on the Hamming weight or Hamming distance of the calculated intermediate values) is used to map the intermediate values to a hypothetical power consumption value. With the help of powerful statistical methods (e.g., correlation between measured and hypothetical power consumption) the most likely key hypothesis is identified. If the power model resembles the actual power consumption characteristics, the correct key hypothesis will always be the most likely one and the analysis procedure will deliver the correct key value. The analysis procedure can then be repeated targeting another portion of the key until the whole key is determined or a brute-force attack becomes feasible.

2.1.5.3 EM attacks

In general, electromagnetic (EM) attacks [28] [29] [30] are similar to power-analysis attacks. Power traces are collected during the operation of the cryptographic device with over an EM probe. The spatial positioning of the probe can potentially be used to improve information extraction from the device. Analysis of the traces is carried out similarly as power analysis with simple and differential methods, which are sometimes denotes as simple EM analysis (SEMA) and differential EM analysis (DEMA), respectively.
The link between electromagnetic emanations and power consumption is given by the fact that any movement of electric charges is accompanied by an electromagnetic field.

2.1.5.4 Combined attacks

The power of side-channel analysis attacks can be increased further by combining the information of several side channels (e.g., power consumption and EM) or several attack techniques (e.g., template-based DPA [25]). Many of the possible combinations still remain to be explored.

2.1.6 Other attacks

2.1.6.1 Message modification

Message modification is the alteration of messages or data, usually the collection of sensors readings, with the aim of causing confusion to the network. An adversary can simply intercept and modify the unsecured packets content meant by the base station or intermediate nodes to disrupt the sensors value of a particular sensing region. For example, a burglar may attempt to modify a motion sensors signal to avoid alerting the base station of an intrusion. In this way, the intrusion by the burglar will not be detected by the central control system since no alert message was received.

2.1.6.2 Message replay

A message replay or replay attack is an attack where the adversary reuses valid transaction messages or packets content with malicious intent. The adversary performs a replay attack by, first, intercepting a valid critical transaction data packet and, then, re-transmitting at a later time. This critical transaction data can be, for example, a proof of identity in a form of a response to a challenge sent by a verifier. Hence, by re-transmitting the correct response that has been captured earlier to the same challenge, issued by the verifier, an adversary can fool the verifier to believe that the adversary is the valid party in response to the challenge that was sent out.

2.2 Ciphers

One solution in order to avoid many of the threats referred to the previous section is the use of cipher encryption. In cryptography, a *cipher* (or *cypher*) [31] [32] is an algorithm for performing encryption and decryption, a series of well-defined steps that can be followed as a procedure. An alternative term is encipherment. In

non-technical usage, a *cipher* is the same thing as a *code*; however, the concepts are distinct in cryptography. In classical cryptography, ciphers were distinguished from codes. Codes operated by substituting according to a large codebook which linked a random string of characters or numbers to a word or phrase. For example, "UQJHSE" could be the code for "Proceed to the following coordinates". When using a cipher, the original information is known as plaintext and the encrypted form as ciphertext. The ciphertext message contains all the information of the plaintext message, but it is not in a format readable by a human or computer without the proper mechanism to decrypt it; it should resemble random gibberish to those not intended to read it.
The operation of a cipher usually depends on a piece of auxiliary information, called a *key* or, in traditional NSA parlance, a *cryptovariable*. The encrypting procedure is varied depending on the key, which changes the detailed operation of the algorithm. A key must be selected before using a cipher to encrypt a message. Without knowledge of the key, it should be difficult, if not nearly impossible, to decrypt the resulting cipher into readable plaintext.
Most modern ciphers can be categorized in several ways:

- By whether they work on blocks of symbols usually of a fixed size (*block ciphers*) or a continuous stream of symbols (*stream ciphers*).
- By whether the same key is used for both encryption and decryption (*symmetric-key algorithms*) or a different key is used for each (*asymmetric-key algorithms*). If the algorithm is symmetric, the key must be known to the recipient and no one else. If the algorithm is an asymmetric one, the enciphering key is different from, but closely related to, the deciphering key. If one key cannot be deduced from the other, the asymmetric key algorithm has the public/private key property and one of the keys may be made public without loss of confidentiality. The *Feistel cipher* [32] uses a combination of substitution and transposition techniques. Most block cipher algorithms are based on this structure.

2.2.1 Block ciphers

Block ciphers [33] are symmetric-key ciphers which operate on fixed-length groups of bits, termed blocks, with unvarying transformations. When encrypting, a block cipher might take, for example, a 128-bit block of plaintext as input and

output a corresponding 128-bit block of ciphertext. The exact transformation is controlled using a second input, the secret key. Decryption is similar; it takes, in this example, a 128-bit block of ciphertext together with the secret key and yields the original 128-bit block of plaintext.
To encrypt messages longer than the block size (128 bits in the above example), a mode of operation is used.
Block ciphers can be contrasted with stream ciphers; a stream cipher operates on individual digits once at a time and the transformation varies during the encryption. The distinction between these two types is not always clear-cut; a block cipher, when used in certain modes of operation, acts effectively as a stream cipher.
An early and highly influential block cipher design was the Data Encryption Standard (DES) [34] [35] developed by IBM and published as a standard in 1977. A successor to DES, the Advanced Encryption Standard (AES) [36], was adopted in 2001.

2.2.2 Stream ciphers

Stream ciphers [37] are symmetric key ciphers where plaintext bits are combined with a pseudo-random cipher bitstream (keystream), typically by an exclusive-or (XOR) operation. In a stream cipher, the plaintext digits are encrypted once at a time and the transformation of successive digits varies during the encryption. An alternative name is a *state cipher*, as the encryption of each digit is dependent on the current state. In practice, the digits are typically single bits or bytes.
Stream ciphers represent a different approach to symmetric encryption from block ciphers. Block ciphers operate on large blocks of digits with a fixed, unvarying transformation. This distinction is not always clear-cut; in some modes of operation, a block cipher primitive is used in such a way that it acts effectively as a stream cipher. Stream ciphers typically execute at a higher speed than block ciphers and have lower hardware complexity. However, stream ciphers can be susceptible to serious security problems if used incorrectly; see stream cipher attacks in particular, the same starting state must never be used twice.
Stream ciphers are often used in applications where plaintext comes in quantities of unknowable length, for example, a secure wireless connection. If a block cipher were to be used in this type of application, the designer would need to choose either transmission efficiency or implementation complexity, since block ciphers cannot directly

work on blocks shorter than their block size. For example, if a 128-bit block cipher received separate 32-bit bursts of plaintext, three quarters of the data transmitted would be padding. Block ciphers must be used in ciphertext stealing or residual block termination mode to avoid padding, while stream ciphers eliminate this issue by naturally operating on the smallest unit that can be transmitted (usually bytes). Another advantage of stream ciphers in military cryptography is that the cipher stream can be generated in a separate box that is subject to strict security measures and fed to other devices, e.g., a radio set, which will perform the XOR operation as part of their function. The latter device can then be designed and used in less stringent environments.
RC4 [38] is the most widely used stream cipher in software; others include: A5/1 [39], FISH [40], Phelix [41], ISAAC [42], MUGI [43] [44] [45], Panama [46], SEAL [47] [48] and SOBER-128 [49] [50].

2.2.3 Symmetric-key algorithms

Symmetric-key algorithms [51] are a class of cryptographic algorithms that use trivially related, often identical, cryptographic keys for both decryption and encryption.
The encryption key is trivially related to the decryption key; they may be identical or there is a simple transform to go between the two keys. The keys, in practice, represent a shared secret between two or more parties that can be used to maintain a private information link.
Other terms for symmetric-key encryption are *secret-key*, *single-key*, *shared-key*, *one-key* and eventually *private-key* encryption. Use of the latter term does conflict with the term private key in public-key cryptography.
Some examples of popular and well-respected symmetric algorithms include Twofish [52], Serpent [53], AES (a.k.a. Rijndael) [36], Blowfish [54], CAST5 [55], RC4 [38], TDES [56] and IDEA [57].

2.2.4 Asymmetric-key algorithms

Public-key cryptography [32] is a method for secret communication between two parties without requiring an initial exchange of secret keys. It can also be used to create digital signatures. Public-key cryptography is a fundamental and widely used technology around the world and enables secure transmission of information

on the Internet.
It is also known as asymmetric key cryptography because the key used to encrypt a message differs from the key used to decrypt it. In public-key cryptography, a user has a pair of cryptographic keys, a public key and a private key. The private key is kept secret, while the public key may be widely distributed. Messages are encrypted with the recipient's public key and can only be decrypted with the corresponding private key. The keys are related mathematically, but the private key cannot be feasibly (i.e., in actual or projected practice) derived from the public key.
Symmetric cryptography uses a single secret key for both encryption and decryption. To use a symmetric encryption scheme, the sender and receiver must share a key in advance. Because symmetric encryption is less computationally intensive and requires less bandwidth, it is common to exchange a key using a key-exchange algorithm and transmit data using an enciphering scheme.

2.3 Dual-rail

As previously mentioned, cryptographic algorithms are implemented on the specific devices in order to mitigate the threats to the network, but these solutions make them vulnerable to side channel analysis such timing and power consumption analysis. Several methods have been proposed for avoiding this threat.
Dual-rail encoding proposed by Sokolov et al. in [58] provides a method to enhance the security properties of a system, making DPA more difficult.

2.3.1 Single spacer dual-rail

Dual-rail code uses two rails with only two valid signal combinations, {01} and {10}, which encode values {0} and {1}, respectively. Dual-rail code is widely used to represent data in self-timed circuits [59] [60], where a specific protocol of switching helps to avoid hazards. The protocol allows only transitions from *all-zeros* {00}, which is a non-code word, to a *code word* and back to *all-zeros*, as shown in Figure 2.1; this means the switching is monotonic. The *all-zeros* state is used to indicate the absence of data, which separates one *code word* from another. Such a state is often called a *spacer*.
An approach for automatically converting single-rail circuits to dual-rail using the above signaling protocol that is easy to incorporate in the standard RTL-based de-

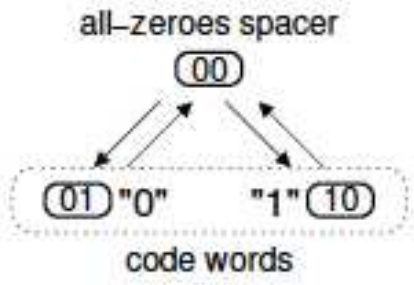

Figure 2.1: Single spacer dual-rail protocol.

sign flow has been described in [61]. Within this approach, called Null-Convention Logic [62], one of two major implementation strategies for logic can be followed; the first one is with full completion detection through the dual-rail signals (NCL-D) and the other one with separate completion detection (NCL-X). The former implementation strategy is more conservative with respect to delay dependence, while the latter one is less delay-insensitive, but more area and speed efficient. NCL methods of circuit construction exploit the fact that the negation operation in dual-rail corresponds to swapping the rails. Such dual-rail circuits do not have negative gates (internal negative gates, for example, in XOR elements, are also converted into positive gates), hence they are race-free under any single transition.

If the design objective is only power balancing (as in our case), one can abandon the completion detection channels relying on timing assumptions as in standard synchronous designs, thus saving a considerable amount of area and power. This approach was followed in [62] considering the circuit in a clocked environment, where such timing assumptions were deemed quite reasonable to avoid any hazards in the combinational logic. Hence, in the clocked environment, the dual-rail logic of an AND gate is simply a pair of AND and OR gates.

The above implementation techniques certainly help to balance switching activity at the level of dual-rail nodes. Assuming that the power consumed by one rail in a pair is the same as in the other rail, the overall power consumption is invariant to the data bits propagating through the dual-rail circuit. However, the physical realization of the rails at the gate level is not symmetric and experiments with these

dual-rail implementations show that power source current leaks the data values. While there could be ways of balancing power consumption between individual gates in dual-rail pairs by means of modifications at the transistor level, adjusting loads, changing transistor sizes, e.t.c., all such measures are costly. The standard logic library requires finding a more economic solution. Randomization techniques can be also applied independently and possibly in conjunction with the above method. Synchronous flip-flops are built to be power efficient, so if they switch to the same value (data input remains the same within several clocks), nothing will change at the output. The absence of the output transition saves power, but, at the same time, it makes the power consumption data dependent. In order to avoid this, flip-flops are made in order to operate in the return-to-spacer protocol. This solution uses the master-slave scheme, writing to the master is controlled by the positive edge of the clock and writing to the slave is controlled by the negative edge. At the same time, the high value of the clock enforces slave outputs into zero and the low clock value enforces master outputs into one (a similar spacer for the logic with active zero).
In this circuit explained before, both the master and slave latches have their respective reset and enable inputs (active zero for the master). The delay between removing the reset signal and disabling writing for each latch (hold time) is formed by the couple of buffers in the clock circuit. Buffers between master and slave are needed to insert a delay. The advantage of this implementation is the use of a single cross-coupled latch in each stage for a couple of input data signals.

2.3.2 Dual spacer dual-rail

In order to balance the power signature, the use of two spacers [63] is proposed (i.e., two spacer states, {00} for *all-zeros spacer* and {11} for *all-ones spacer*), resulting in a dual spacer protocol as shown in Figure 2.2. It defines the switching as follows: *spacer* $\longrightarrow$ *code word* $\longrightarrow$ *spacer* $\longrightarrow$ *code word*. The polarity of the spacer can be arbitrary and possibly random, as in Figure 2.2a. A possible refinement for this protocol is the *alternating spacer protocol* shown in Figure 2.2b. The advantage of the latter is that all bits are switched in each cycle of operation, thus opening a possibility for perfect energy balancing between cycles of operation.
As opposed to single spacer dual-rail, where, in each cycle, a particular rail is switched up and down (i.e., the same gate always switches), in the *alternating spacer protocol*, both rails are switched from *all-zeros spacer* to *all-ones spacer* and back.

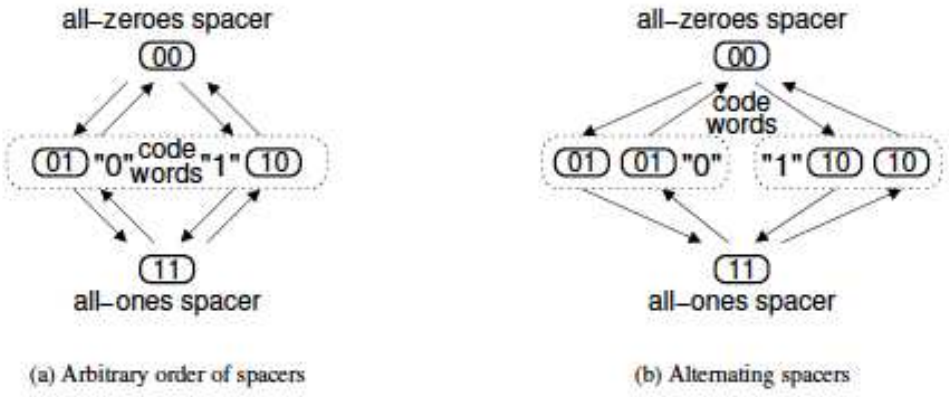

Figure 2.2: Dual spacer dual-rail protocol.

The intermediate states in this switching are *code words*. In the scope of the entire logic circuit, this means that, for every computation cycle, all gates are always fired forming the dual-rail pairs. This makes the circuit more resistant to DPA.

The new alternating spacer discipline cannot be directly applied to the implementation techniques described in the previous subsection. Those, both in the logic rails as well as in completion detection, assume the fact that, for each pair of rails, the {11} combination never occurs. In fact, the use of *all-ones spacer* would upset the speed-independent implementation in NCL-D because the outputs of the second layer elements would not be acknowledged during *code word* $\longrightarrow$ *all-ones spacer* transition. The completion detection for those gates can, of course, be ensured by using an additional three-input C-element, but this extra overhead would make this implementation technique much less elegant because of the additional acknowledgment signal channel. In the single spacer structure, due to the principle of orthogonality (one-hot) between min-terms $a_0 \cdot b_0$, $a_1 \cdot b_0$ and $a_0 \cdot b_1$, only one C-element in the rail c_0 fires per cycle.

If some parts of a dual-rail circuit operate using the single spacer and other parts the alternating spacer protocol, then spacer converters should be used. The alternating-to-single spacer converter is transparent to *code words* and enforces *all-zeros spacer* on the output if the input is all-ones or all-zeros.

The implementation of a single-to-alternating spacer converter uses a toggle to de-

cide which spacer to inject all-ones or all-zeros. The toggle can be constructed out of two latches and operates in the following way:

$$x+ \longrightarrow x1+ \longrightarrow x- \longrightarrow x2+ \longrightarrow x+ \longrightarrow x1- \longrightarrow x- \longrightarrow x2-,$$

i.e., $x1$ changes on the positive edge of x and $x2$ switches on its negative edge. The frequency of $x1$ and $x2$ is half the frequency of x.

The alternation of spacers in time is enforced by flip-flops. The alternating spacer flip-flop can be built by combining a single spacer dual-rail flip-flop with a single spacer to alternating spacer converter. The power consumption of the single spacer dual-rail flip-flop is data independent due to the symmetry of its rails. The rails of the spacer converter are also symmetric, which makes the power consumption of the resultant alternating spacer flip-flop data independent. This implementation uses the *clk2* signal to decide which spacer to inject on the positive phase of *clk*. The signal *clk2* changes on the negative edge of the clock and is formed by a toggle (one for the whole circuit) whose input is *clk*. The timing assumption for *clk2* is that it changes after the output of single spacer flip-flop. Both the slave latch of the single spacer flip-flop and the toggle which generates the *clk2* signal are triggered by the negative edge of *clk*. The depth of logic in the toggle is greater than in the slave latch of the flip-flop. At the same time, *clk2* goes to all flip-flops of the circuit and requires buffering, which also delays it.

It should be mentioned that the inputs of the dual-rail circuit must also support the alternating spacer protocol. Moreover, the same spacer should appear each cycle on the inputs of a dual-rail gate. That means the spacer protocol on the circuit inputs and flip-flop outputs must be synchronized in the reset phase.

Chapter 3

Cipher encryption schemes

3.1 Skipjack

Skipjack [64] is a block cipher developed by the U.S. National Security Agency (NSA). Initially classified, it was originally intended for use in the controversial *Clipper chip* [65]. Subsequently, the algorithm was declassified and now provides a unique insight into the cipher designs of a government intelligence agency. Skipjack was proposed as the encryption algorithm in a U.S. government-sponsored scheme of *key escrow* and the cipher provided for use in the *Clipper chip* and implemented in tamper-proof hardware. Skipjack is used only for encryption; the key escrow is achieved through the use of a separate mechanism known as the Law Enforcement Access Field (LEAF) [66]. The design was initially secret and was regarded with considerable suspicion by many in the public cryptography community for that reason. It was declassified on 24 June 1998. To ensure public confidence in the algorithm, several academic researchers from outside the government were called in to evaluate the algorithm [67]. The researchers found no problems with either the algorithm itself or the evaluation process.
Skipjack uses an 80-bit key to encrypt or decrypt 64-bit data blocks. It is an unbalanced Feistel network [32] with 32 rounds. It was specially designed to replace the Data Encryption Standard (DES) [34].

3.1.1 Basic Structure

Skipjack encrypts 4-word (i.e., 8-byte) data blocks by alternating between the two stepping rules (A and B) shown in Figure 3.1 and 3.2. A step of rule A does the following:

1. G permutes w_1,
2. the new w_1 is the XOR of the G output, the counter and w_4,

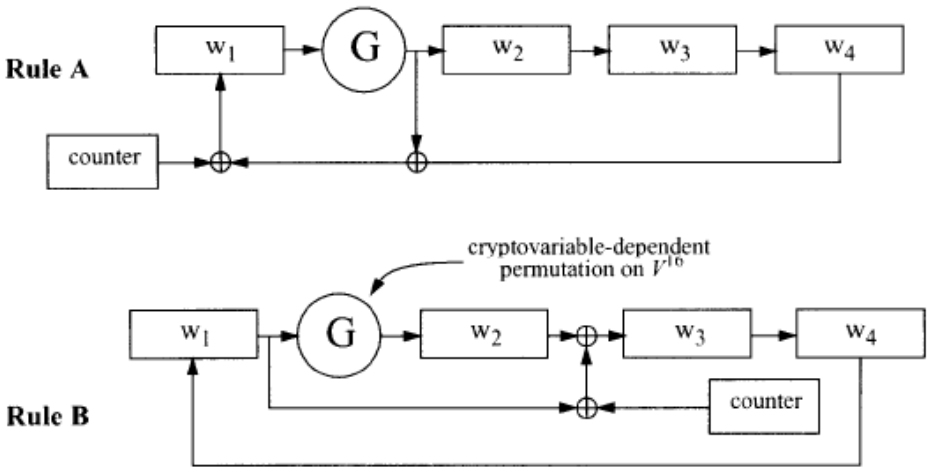

Figure 3.1: Skipjack stepping rules

3. words w_2 and w_3 shift one register to the right; i.e., become w_3 and w_4 respectively,

4. the new w_2 is the G output and

5. the counter is incremented by one.

Rule B works similarly.
The algorithm requires a total of 32 steps. As referred to the encryption procedure, the input is w_i^0, $1 \leq \mathrm{i} \leq 4$, (i.e., k = 0 for the beginning step). The counter is initialized to start at 1. It steps 8 times according to Rule A, then switch to Rule B and steps 8 more times. Return to Rule A for the next 8 steps and, then, complete the encryption with 8 steps in Rule B. The counter increments by one after each step. The output is w_i^{32}, $1 \leq \mathrm{i} \leq 4$. In contrast with the encryption procedure, the decryption one has as input w_i^{32}, $1 \leq \mathrm{i} \leq 4$, (i.e., k = 32 for the beginning step). The counter starts at 32, steps according to Rule B^{-1} for 8 times, then switch to Rule A^{-1} and perform 8 more steps. Then, it returns to Rule B^{-1} for the next 8 steps and, finally, it completes the decryption with 8 steps based on Rule A^{-1}. The counter is decremented by one after each step. The output is w_i^0, $1 \leq \mathrm{i} \leq 4$.

ENCRYPT

Rule A	Rule B
$w_1^{k+1} = G^k(w_1^k) \oplus w_4^k \oplus counter^k$	$w_1^{k+1} = w_4^k$
$w_2^{k+1} = G^k(w_1^k)$	$w_2^{k+1} = G^k(w_1^k)$
$w_3^{k+1} = w_2^k$	$w_3^{k+1} = w_1^k \oplus w_2^k \oplus counter^k$
$w_4^{k+1} = w_3^k$	$w_4^{k+1} = w_3^k$

DECRYPT

Rule A^{-1}	Rule B^{-1}
$w_1^{k-1} = [G^{k-1}]^{-1}(w_2^k)$	$w_1^{k-1} = [G^{k-1}]^{-1}(w_2^k)$
$w_2^{k-1} = w_3^k$	$w_2^{k-1} = [G^{k-1}]^{-1}(w_2^k) \oplus w_3^k \oplus counter^{k-1}$
$w_3^{k-1} = w_4^k$	$w_3^{k-1} = w_4^k$
$w_4^{k-1} = w_1^k \oplus w_2^k \oplus counter^{k-1}$	$w_4^{k-1} = w_1^k$

Figure 3.2: Stepping rules equations

3.1.2 G-permutation

The cryptovariable-dependent permutation G on a subword (16 bits) is a four-round Feistel structure. The round function is implemented by a fixed byte-substitution table (permutation on a byte), which is called the F-table. Each round of G also incorporates a byte cryptovariable. We give two characterizations of the function below:

1. Recursively (mathematically): $G^k(g_1||g_2) = g_5||g_6$, where
   $g_i = F(g_{i-1} \bigoplus cv_{4k+i-3}) \bigoplus g_{i-2}$, k is the step number (the first step is 0), F is the substitution table and cv_{4k+i-3} is the $(4k+i-3)^{th}$ byte in cryptovariable schedule. Thus,

$$g_3 = F(g_2 \bigoplus cv_{4k}) \bigoplus g_1$$

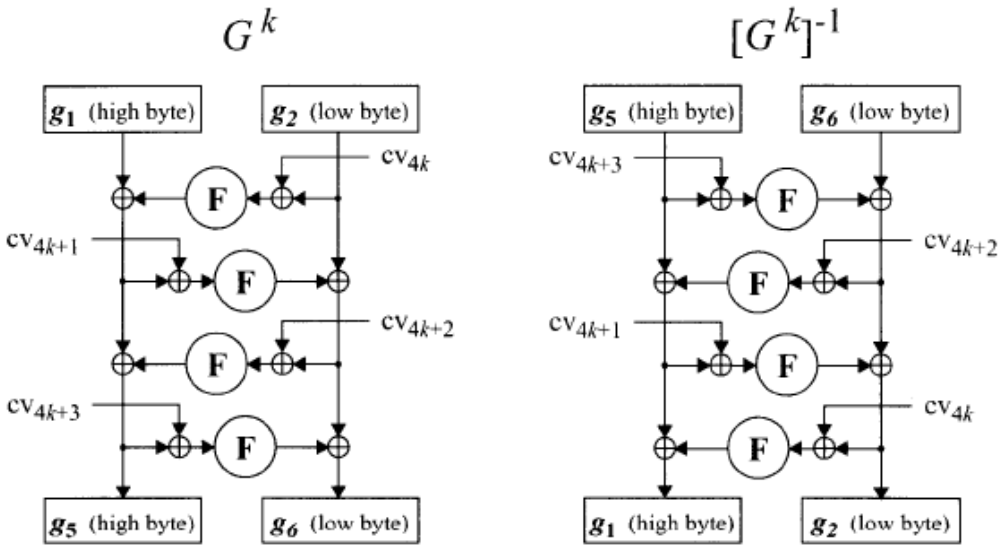

Figure 3.3: G-permutation diagram

$$g_4 = F(g_3 \bigoplus cv_{4k+1}) \bigoplus g_2$$
$$g_5 = F(g_4 \bigoplus cv_{4k+2}) \bigoplus g_3$$
$$g_6 = F(g_5 \bigoplus cv_{4k+3}) \bigoplus g_4.$$

Similarly, for the inverse, $[G^k]^-1(w = g_5||g_6) = g_1||g_2$, where

$$g_{i-2} = F(g_{i-1} \oplus cv_{4k+i-3}) \oplus g_i.$$

2. Figure 3.3 presents a schematic description of G-permutation.

The cryptovariable is 10 bytes long (labelled 0 through 9) and used in its natural order. So the schedule subscripts given in the definition of the G-permutation are to be interpreted in a modulo-10 manner.
The Skipjack F-table is given in Figure 3.4 in hexadecimal notation. The high order 4 bits of the input index the row and the low order 4 bits index the column.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	a3	d7	09	83	f8	48	f6	f4	b3	21	15	78	99	b1	af	f9
1x	e7	2d	4d	8a	ce	4c	ca	2e	52	95	d9	1e	4e	38	44	28
2x	0a	df	02	a0	17	f1	60	68	12	b7	7a	c3	e9	fa	3d	53
3x	96	84	6b	ba	f2	63	9a	19	7c	ae	e5	f5	f7	16	6a	a2
4x	39	b6	7b	0f	c1	93	81	1b	ee	b4	1a	ea	d0	91	2f	b8
5x	55	b9	da	85	3f	41	bf	e0	5a	58	80	5f	66	0b	d8	90
6x	35	d5	c0	a7	33	06	65	69	45	00	94	56	6d	98	9b	76
7x	97	fc	b2	c2	b0	fe	db	20	e1	eb	d6	e4	dd	47	4a	1d
8x	42	ed	9e	6e	49	3c	cd	43	27	d2	07	d4	de	c7	67	18
9x	89	cb	30	1f	8d	c6	8f	aa	c8	74	dc	c9	5d	5c	31	a4
Ax	70	88	61	2c	9f	0d	2b	87	50	82	54	64	26	7d	03	40
Bx	34	4b	1c	73	d1	c4	fd	3b	cc	fb	7f	ab	e6	3e	5b	a5
Cx	ad	04	23	9c	14	51	22	f0	29	79	71	7e	ff	8c	0e	e2
Dx	0c	ef	bc	72	75	6f	37	a1	ec	d3	8e	62	8b	86	10	e8
Ex	08	77	11	be	92	4f	24	c5	32	36	9d	cf	f3	a6	bb	ac
Fx	5e	6c	a9	13	57	25	b5	e3	bd	a8	3a	01	05	59	2a	46

Figure 3.4: Skipjack F-table

3.2 Blowfish

Blowfish [54] is a secret-key block cipher that can be used as a drop-in replacement of DES [34] or IDEA [57]. It takes a variable-length key, from 32 bits to 448 bits, making it ideal for both domestic and exportable use. Blowfish was designed in 1993 by Bruce Schneier as a fast, free alternative to existing encryption algorithms. Since then it has been considerably analyzed and it is slowly gaining acceptance as a strong encryption algorithm.

Blowfish is a Feistel network, employing a simple encryption function 16 times. Its block size is 64 bits and its key can have any length up to 448 bits. Although there is a complex initialization phase required before any encryption takes place, the actual encryption of data is very efficient [54].

Blowfish is a variable-length key, 64-bit block cipher. The algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key of at most 448 bits into several sub-key arrays totaling 4168 bytes.
Data encryption occurs via a 16-round Feistel network. Each round consists of a key-dependent permutation and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round.

3.2.1 Sub-keys

Blowfish uses a large number of sub-keys. These keys must be precomputed before any data encryption or decryption.

- The P-array consists of 18 32-bit sub-keys:

  P1, P2,..., P18.

- There are four 32-bit S-boxes with 256 entries each:

  S1,0, S1,1,..., S1,255;
  S2,0, S2,1,..,, S2,255;
  S3,0, S3,1,..., S3,255;
  S4,0, S4,1,..,, S4,255.

3.2.2 Encryption

As mentioned above, the encryption process of Blowfish consists of 16 rounds (see Figure 3.5) and its input is a 64-bit data element, X.

The data encryption algorithm is shown below:

Divide X into two 32-bit halves: X_L, X_R
For i = 1 to 16:
$X_L = X_L \oplus P_i$
$X_R = F(X_L) \oplus X_R$
Swap X_L and X_R
Next i

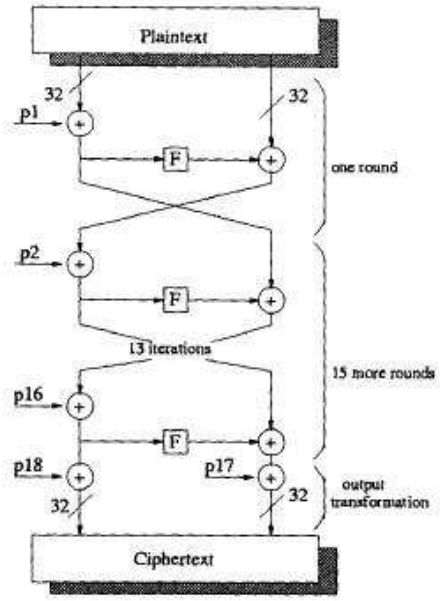

Figure 3.5: Data flow graph of Blowfish block cipher

Swap X_L and X_R (Undo the last swap)
$X_R = X_R \oplus P_{17}$
$X_L = X_L \oplus P_{18}$
Recombine X_L and X_R

3.2.3 Function F

Function F, the non-reversible function, gives Blowfish the best possible avalanche effect for a Feistel network: every text bit on the left half of the round affects every text bit on the right half. Additionally, since every sub-key bit is affected by every key bit, the function also has a perfect avalanche effect between the key and the right half of the text after every round. Hence, the algorithm exhibits a perfect avalanche effect after three rounds as well as every two rounds after that.
The Function F implemented in our systems is an 8x8 S-box, which has been derived from the method developed by Yi et al. [68] and is presented in Figure 3.6.

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

Figure 3.6: Blowfish function F

3.3 Advanced encryption standard

Advanced Encryption Standard (AES) [36] is an encryption standard adopted by the U.S. government. The standard comprises three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as Rijndael [69] [70]. Each AES cipher has a 128-bit block size, with key sizes of 128, 192 and 256 bits, respectively. The AES ciphers have been extensively analyzed and are now used worldwide, as was the case with its predecessor, the Data Encryption Standard (DES) [34].

The rest of this section specifies the Rijndael algorithm, a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with length of 128 bits. Rijndael was designed to handle additional block sizes and key lengths, however

they are not adopted in this work. Throughout the remainder of this section, the algorithm specified herein will be referred to as "the AES algorithm".
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 bits.
Internally, the AES algorithm operations are performed on a two-dimensional array of bytes called the *State*. The *State* consists of four rows of bytes, each containing four bytes. 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 $0 \leq c < 4$. This allows an individual byte of the *State* to be referred to as either $s_{r,c}$ or $s[r, c]$.
At the start of the *Cipher*, the input - the array of bytes $in_0, in_1, ...in_{15}$ - is copied into the *State array* as illustrated in Figure 3.7. The *Cipher* operation is then conducted on this *State array*. After the completion of this operation, its final value is copied to the output - the array of bytes $out_0, out_1, ...out_{15}$.
Hence, at the beginning of the *Cipher*, the input array, *in*, is copied to the *State array* according to the scheme:

$$s[r, c] = in[r + 4c], \text{ for } 0 \leq r < 4 \text{ and } 0 \leq c < 4,$$

and at the end of the *Cipher*, the *State* is copied to the output array, *out*, as follows:

$$out[r + 4c] = s[r, c], \text{ for } 0 \leq r < 4 \text{ and } 0 \leq c < 4.$$

For its *Cipher*, the AES algorithm uses a round function that is composed of four different byte-oriented transformations:

1. byte substitution using a substitution table (*S-box*),
2. shifting rows of the *State array* by different offsets,
3. mixing the data within each column of the *State array* and
4. adding a *Round Key* to the *State*.

These transformations are described in the following subsections.

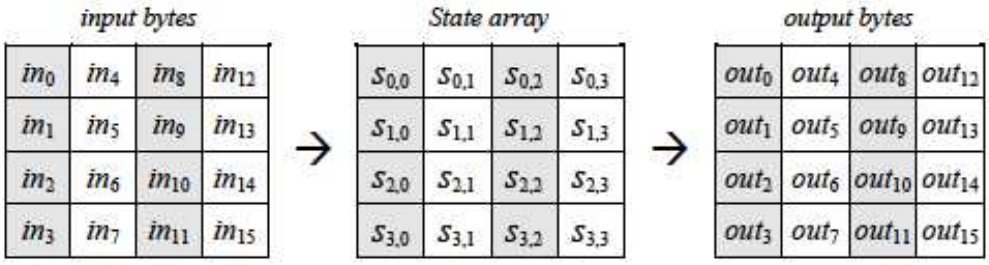

Figure 3.7: State array input and output.

3.3.1 Cipher

At the start of the *Cipher*, the input is copied to the *State array* as described above. After an initial *Round Key* addition, the *State array* is transformed by implementing a round function 10 times, with the final round differing slightly from the first 9 ones. The final *State* is then copied to the output as described above.
The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived by the *Key Expansion* routine.
The *Cipher* is described in the pseudocode in Figure 3.8. The individual transformations - *SubBytes()*, *ShiftRows()*, *MixColumns()* and *AddRoundKey()* process the *State* and are described in the following subsections. In Figure 3.8, the array *w[]* contains the key schedule.
As shown in Figure 3.8, all rounds are identical with the exception of the final round, which does not include the *MixColumns()* transformation.

3.3.1.1 SubBytes() transformation

The *SubBytes()* transformation is a non-linear byte substitution that operates independently on each byte of the *State* using a substitution table (*S-box*). This S-box, which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field $GF(2^8)$; the element {00} is mapped to itself.

2. Apply the following affine transformation (over $GF(2)$):

```
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])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      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 3.8: Pseudocode for the AES cipher.

$$b'_i = b_i \oplus b_{(i+4) mod 8} \oplus b_{(i+5) mod 8} \oplus b_{(i+6) mod 8} \oplus b_{(i+7) mod 8} \oplus c_i$$

for $0 \leq i < 8$, where b_i is the i^{th} bit of the byte, and c_i is the i^{th} bit of a byte c with the value $\{63\}$ or $\{01100011\}$. Here and elsewhere, a prime on a variable (e.g., b') indicates that the variable is to be updated with the value on the right.

In matrix form, the affine transformation element of the *S-box* can be expressed as:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Figure 3.9 illustrates the effect of the *SubBytes()* transformation on the *State*.

The *S-box* used in the *SubBytes()* transformation is presented in hexadecimal form

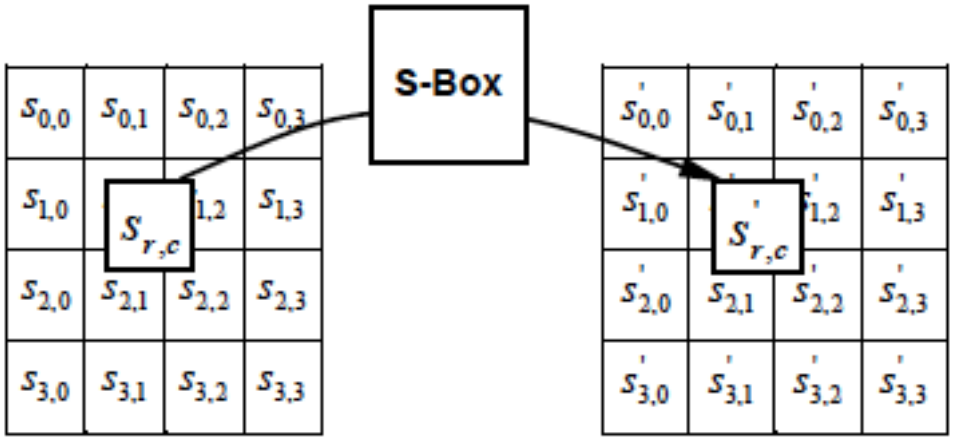

Figure 3.9: SubBytes() applies the S-box to each byte of the State.

in Figure 3.10. For example, if $s_{1,1} = \{53\}$, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Figure 3.10. This would result in $s'_{1,1}$ having a value of $\{ed\}$.

3.3.1.2 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:

$$s'_{r,c} = s_{r,(c+shift(r,4))mod4}, \text{ for } 0 \leq r < 4 \text{ and } 0 \leq c \leq 4,$$

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

$$shift(1,4) = 1;\ shift(2,4) = 2;\ shift(3,4) = 3.$$

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).
Figure 3.11 illustrates the *ShiftRows()* transformation.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.10: S-box: substitution values for the byte xy (in hexadecimal format).

3.3.1.3 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

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

The above equation can be written as a matrix multiplication. Let $s'(x) = a(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix},$$

for $0 \leq c < 4$.
As a result of this multiplication, the four bytes in a column are replaced by the following:

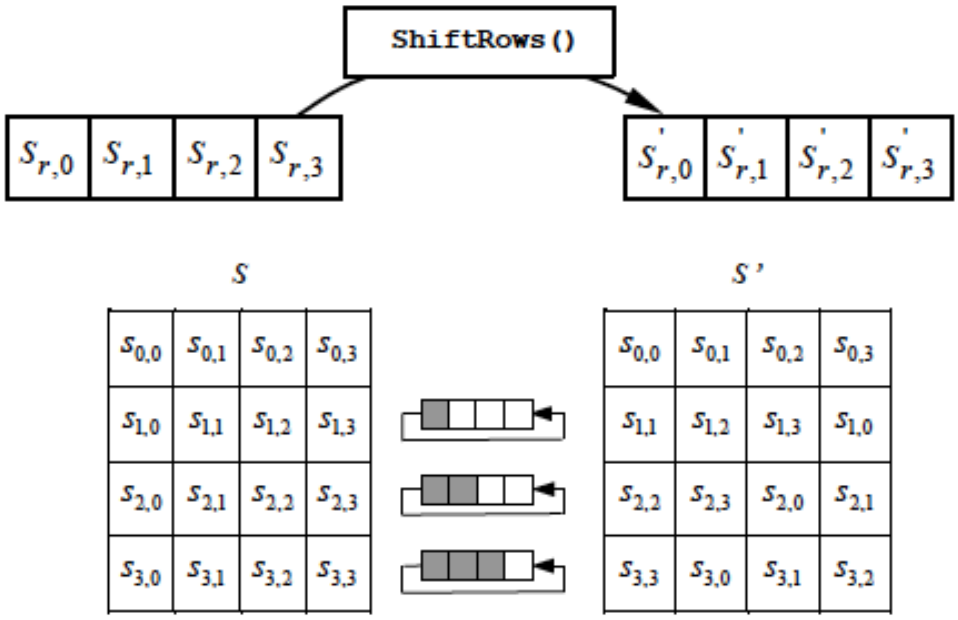

Figure 3.11: ShiftRows() cyclically shifts the last three rows in the State.

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \\
s'_{3,c} &= (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c})
\end{aligned}
$$

Figure 3.12 illustrates the *MixColumns()* transformation.

3.3.1.4 AddRoundKey() transformation

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

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round \cdot 4+c}], \text{ for } 0 \leq c < 4,$$

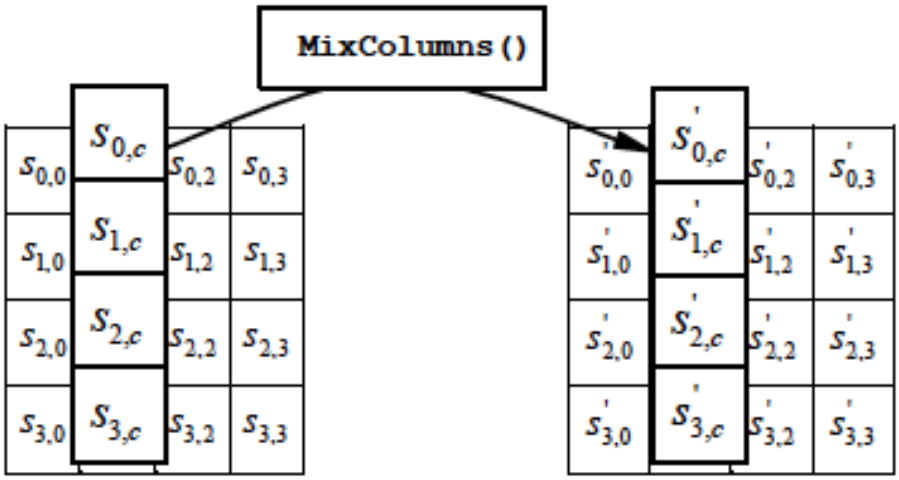

Figure 3.12: MixColumns() operates on the State column-by-column.

where $[w_i]$ are the key schedule words and *round* is a value in the range $0 \leq round \leq 10$. In the *Cipher*, the initial *Round Key* addition occurs when $round = 0$, prior to the first application of the round function. The application of the *AddRoundKey()* transformation to the 10 rounds of the *Cipher* occurs when $1 \leq round \leq 10$.
The action of this transformation is illustrated in Figure 3.13, where $l = round \cdot 4$.

3.3.2 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 44 words; the algorithm requires an initial set of 4 words and each of the 10 rounds requires 4 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 $0 \leq i < 44$.
The expansion of the input key into the key schedule proceeds according to the pseudocode in Figure 3.14.
SubWord() is a function that takes a four-byte input word and applies the *S-box* (Figure 3.10) 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

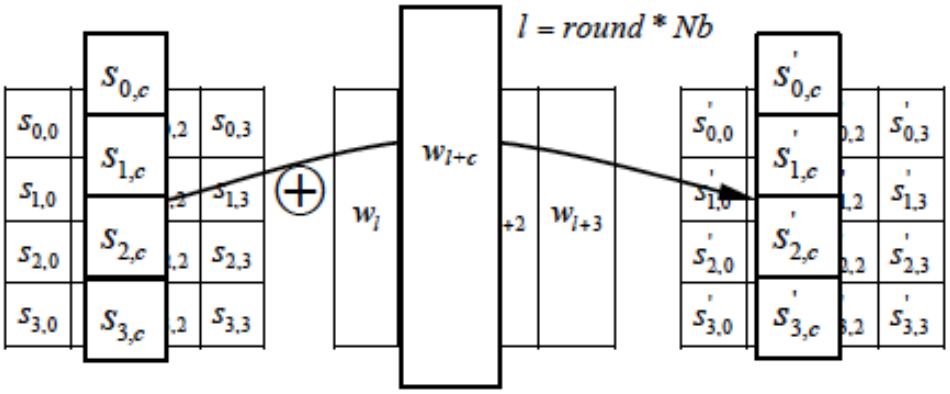

Figure 3.13: AddRoundKey() XORs each column of the State with a word from the key schedule.

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 3.14, it can be seen that the first 4 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 4 positions earlier, *w[i-4]*. For words in positions that are a multiple of 4, 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()*).

```
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 3.14: Pseudocode for Key Expansion.

Chapter 4

Implementation

This chapter describes both the sensor node and base station architecture and implementation in detail including the hardware and the software used for these nodes integration. As this work demonstrates, we can use FPGAs and CPLDs in different kinds of nodes (i.e. sensor nodes or base stations) within a WSN environment in order to increase the nodes processing power (thus enabling the implementation of more complex functions) and decrease their overall energy consumption which is a crucial factor of wireless nodes functionality because it affects their battery lifetime. A general scheme of the platform that is going to be described is illustrated in Figure 4.1. This figure shows all the hardware modules that are used for implementing this sophisticated platform consisting of sensor node and base station including both processor and reconfigurable devices. Moreover, all the platform interconnections between these devices are presented.

4.1 Sensor node

4.1.1 Sensor node architecture

Sensor nodes of a wireless sensor network are low requirements nodes that sense the environment and send this data to a base station via a wireless link. In such nodes, we propose to embed a CPLD, a small reconfigurable device, in order to decrease their energy consumption and, consequently, increase their battery lifetime. The main characteristics of the CPLDs are: (a) their very low energy consumption, (b) their relatively high bandwidth when executing certain data manipulation tasks and (c) their low cost (less than $10). On the other hand, the main disadvantage of those devices is their small number of resources allowing them to execute only small, yet very CPU intensive, tasks.

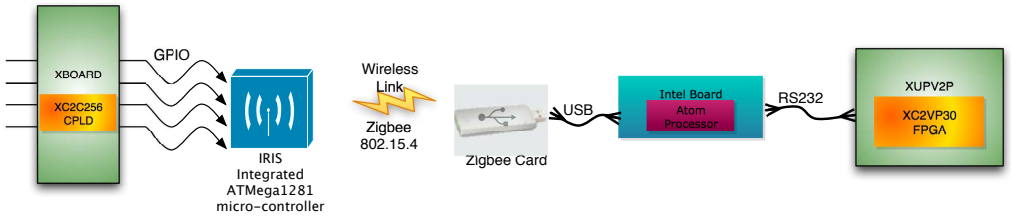

Figure 4.1: Wireless platform general scheme

4.1.1.1 Skipjack

Our first thought for providing security in such nodes is to use the TinySec security protocol. TinySec [10] is the first fully-implemented link layer security architecture for wireless sensor networks developed by Karlof et al. in 2004. TinySec addresses these extreme resource constraints with careful design exploring the trade-offs among different cryptographic primitives and using the inherent sensor network limitations to advantage when choosing parameters to find a sweet spot for security, packet overhead and resource requirements. It is portable to a variety of hardware and radio platforms.

Skipjack is the default cipher encryption scheme of TinySec because it is found to be most appropriate for software implementation on embedded micro-controllers. Taking into account the limited resources of our given CPLD and the low requirements of wireless sensor nodes in security, a mini version of this cipher (with 32-bit plaintext/ ciphertext) is implemented on hardware for taking advantage of the CPLD low energy consumption. This hardware implementation of Skipjack plus Cipher Block Chaining (CBC) [71] provides a relatively high level of security according to sensor nodes requirements.

Moreover, the Skipjack encipher has been implemented on the CPLD since (a) it is a very CPU intensive task, (b) the encryption task is executed much more frequent than the decryption one in the WSN motes (i.e. each node encrypts the collected data, whereas they are decrypted only in their final destination which is most often the base station) and (c) the block cipher encryption tasks consume the majority of the overall *sending message* procedure execution time based on our measurements. In the proposed architecture, we use four entry pipeline registers (*W1*, *W2*, *W3* and *W4*) and four 8-bit 2-to-1 multiplexers which give us the opportunity to choose the plaintext or the result derived from a single step of the algorithm. In particular for

the first step of the algorithm, we use the plaintext and for the subsequent ones we select the previous result as input of the four pipeline registers. Furthermore, we have designed a module named *G-permutation* implementing the specified permutation and its architecture is described below.
Except for the main components of the architecture referred above, there are also some secondary ones used for several purposes. Firstly, a circuit consisting of two XOR gates and a counter increases the grade of randomness of the encryption. The first XOR gate gets as inputs the output of the last register in row (*W4* register) and the output of *G-permutation*. The result of the previous gate is XORed with the value of the counter, which corresponds to the number of the algorithm step. The result of the second XOR gate is the final result of the current step. This process is repeated 32 times equal to the number of algorithm steps. After these 32 steps, the pipeline registers are disabled and their outputs are concatenated in order to form the final result of the encryption procedure.
Figure 4.2 presents the block diagram of the presented Skipjack encryption implementation.
The *G-permutation* box is implemented as shown in Figure 4.3. This device implements the specified permutation previously referred. It consists of eight XOR gates and four 4x4 S-boxes representing the F function. These main components of this module enable us to substitute the value of input byte in a completely different value which lead us to a relatively high level of security. At the beginning of the *G-permutation* procedure, the input byte is divided in two 4-bit quantities, q_1 and q_2. These two quantities are separately processed and, after a total of transformations, concatenated in order to form the final result of the permutation (the output byte) using a *concatenation circuit*. Apart from this input byte, this module has also four other inputs that are four 4-bit long parts of the cryptovariable *cv[4k]*, *cv[4k+1]*, *cv[4k+2]* and *cv[4k+3]*, where k is the number of the current step and $0 \leq k \leq 31$. These cryptovariable parts are used for performing several transformations by XOR them with some intermediate values (g_2, g_3, g_4, g_5) of the two 4-bit quantities as shown in Figure 4.3.

4.1.1.2 Blowfish

Another approach for securing data communication among the sensor nodes of a wireless network is to use Blowfish cipher in combination with CBC. There are

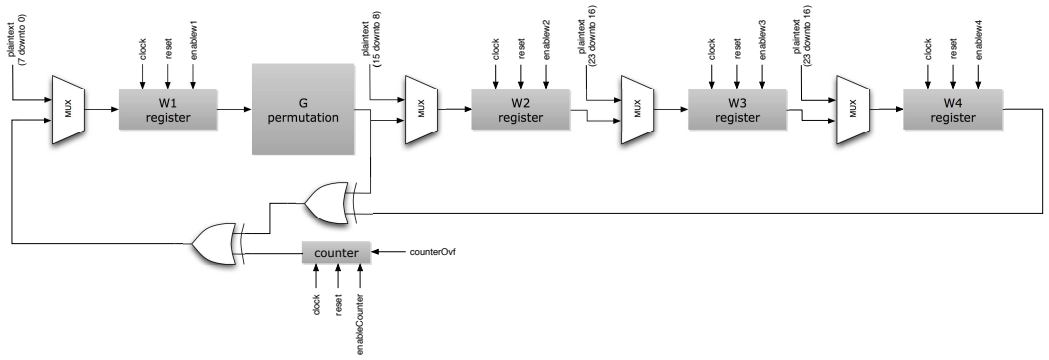

Figure 4.2: Skipjack encryption block diagram

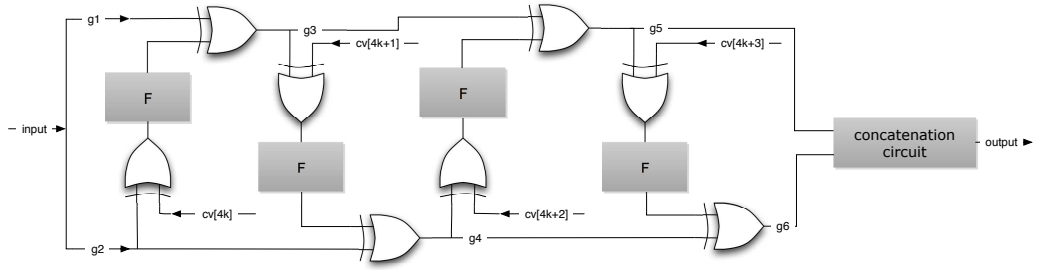

Figure 4.3: Skipjack G-permutation block diagram

also some mini versions of Blowfish, such as Blowfish-16 which has a 16-bit block size. Its small block size makes this version of Blowfish perfectly suitable for wireless sensor network platforms and, moreover, taking into account the limited resources of a CPLD, Blowfish-16 seems a good algorithm to offer security in such small devices. We implement both encipher and decipher process of the Blowfish algorithm in order to have a complete view of cipher performance.

In the proposed architecture, a 16-bit 2-to-1 multiplexer is used. This multiplexer give us the opportunity to choose the plaintext or the result derived from a single round of the algorithm. In particular for the first round of the algorithm, we use the plaintext and for the subsequent ones we select the previous result. Another multiplexer is also used for choosing the specific byte of the crypto-variable that is going to be used to each round. Furthermore, we have designed a module named *Round* implementing a specified round of the algorithm; the architecture of this round is described below.

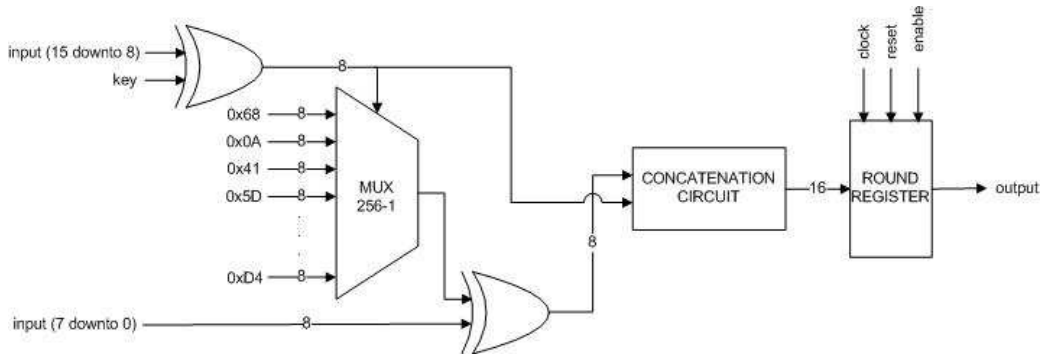

Figure 4.4: Blowfish round block diagram

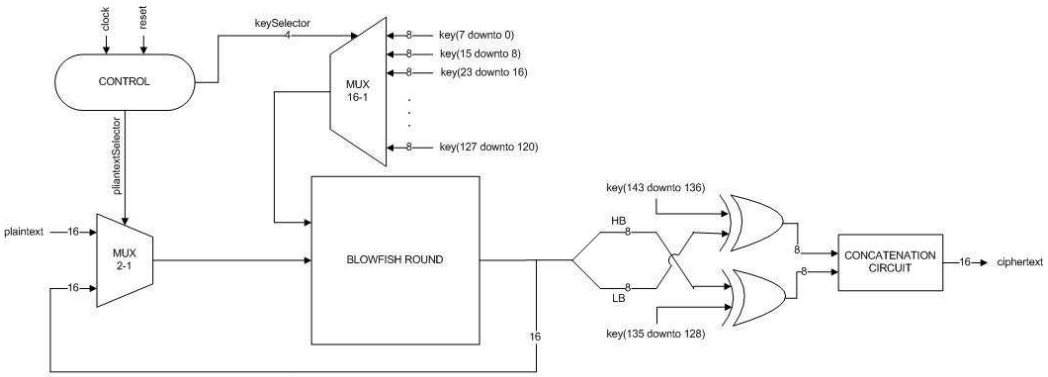

Figure 4.5: Blowfish encryption block diagram

The *Round* box is implemented as shown in Figure 4.4. This system implements a single round of the Blowfish encryption. It consists of a 256-to-1 multiplexer which represents the S-box, two XOR gates, a swap circuit which change the order of the high and low bytes of the result and a concatenation circuit that creates the final result of each round.

The high byte of the input is XORed with the specific byte of the crypto-variable corresponding to the current algorithm round in order to calculate, the new value of the high byte. The result is passed through the S-box and this new value is XORed with the input low byte in order to calculate the new low byte value. Finally, these two bytes are swapped and concatenated in order to create the final result of the current round that is stored in a 16-bit register.

Apart from the main components of the architecture referred above, there are also some secondary ones used for several purposes. Firstly, a swap circuit is used in order to undo the swap of the last algorithm round (round 16). Furthermore, two XOR gates are used; the swapped result of the last round of Blowfish is XORed with the two last bytes of the crypto-variable. Finally, the ciphertext, the output of the whole encryption procedure, is created by using a concatenation circuit which concatenates the results of the two XOR gates.
Figure 4.5 presents the block diagram of the presented Blowfish encryption implementation.
As mentioned above, the presented blocks execute only one of the rounds of the encryption algorithm. So the use of a finite state machine (FSM) is necessary for securing the right operation of the system.
First of all, during the first cycle, the round register is enabled and the plaintext is selected as the input for the first round of the algorithm. Regarding the key selection, the first byte of the crypto-variable is chosen to be used. Then, fifteen cycles, in which the next fifteen rounds of the encryption procedure are executed, follow. The input chosen is the result derived from the previous round and the key is the specific byte that corresponds to the current round (the n-th byte of the crypto-variable is used in the n-th round). Obviously, the duration of the whole procedure is 16 cycles.
Each cycle described above corresponds to a state of the FSM which is presented in Figure 4.6. Moreover, all the information described in this paragraph are summarized in Figure 4.7, which demonstrates the internal loop of the implemented task.
Finally, the only difference between the encipher and the decipher procedure is that the second one takes as inputs the key bytes with the inverse row compared to that of the encipher. Consequently, if the cipher gets as input the *key[r]*, the inverse cipher will take *key[16-r]*, where r is the number of the current cipher round and $0 \leq r \leq 15$.

4.1.2 Sensor node implementation

First of all, our idea of using TinySec with a CPLD implementation of Skipjack drove us to a dead-end. The main reasons of this failure is that the specific security protocol has been ported only to an obsolete version of sensor node operating system and the mini version of the cipher algorithm can not offer the expected

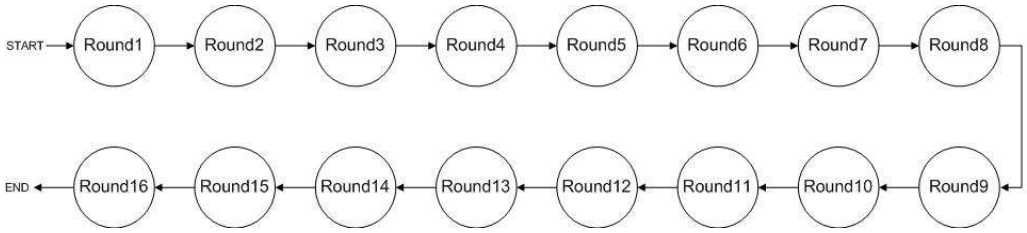

Figure 4.6: Blowfish FSM scheme

(Round1) 1st round of Blowfish encryption – enable round register, plaintext selection, 1st key byte selection

(Round2) 2nd round of Blowfish encryption – enable round register, previous result selection, 2nd key byte selection

(Round3) 3rd round of Blowfish encryption – enable round register, previous result selection, 3rd key byte selection

...

(Round16) 16th round of Blowfish encryption – enable round register, previous result selection, 16th key byte selection, enable output

Figure 4.7: Blowfish procedure

level of security.

Consequently, we decide to proceed with the Blowfish implementation. Since the previously described hardware implementation of the Blowfish cipher encryption supports only 16-bit block, we use CBC in order to be able to efficiently encrypt more data bytes. A software implementation of this mode of operation is used for supporting variable lengths of plaintexts up to 128 bits. This software implements CBC for both encryption and decryption process.

According to the CBC encryption, the first input block (16 bits) is XORed with the input initialization vector (IV) and the result of this port is send to the CPLD in order to be enciphered by Blowfish cipher. The first output block is equal to the ciphertext derived from encipher process and sent back from the CPLD representing the initialization vector of the next CBC step. This procedure previously described

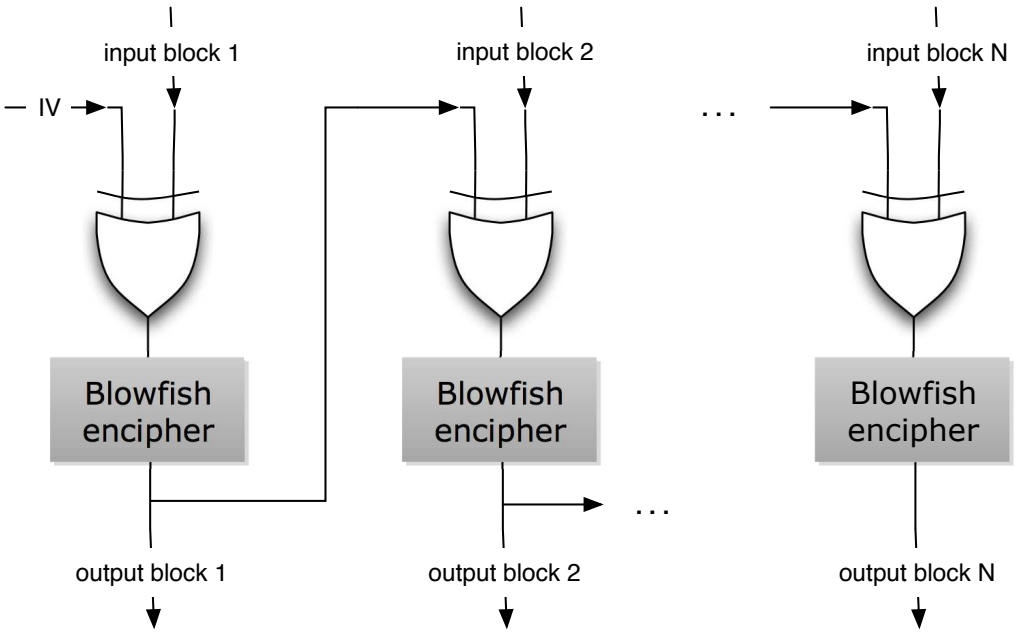

Figure 4.8: CBC encryption implementation

is a single step of the CBC encryption and is repeated such times equal to the number of input blocks (size of plaintext divided by the size of every block being equal to 16 bits). Figure 4.8 shows the scheme of CBC encryption implementation.
Moving to the decryption procedure, every step gets as input an 16-bit long block that is sent to the CPLD in order to be directly deciphered by the Blowfish decipher hardware implementation. The CPLD output is sent back to the mote and is XORed with the initialization vector of the current step which is equal to the input block of the previous CBC step. Finally, the decryption process consists of N steps, where N is the number of input blocks. The scheme of the CBC decryption is illustrated in Figure 4.9.

4.1.2.1 Implementation details

The basic sensor nodes utilized in our infrastructure are the *MICAz* and *IRIS* [72] ones which are probably the most widely used such motes worldwide made by Crossbow Technology. They include IEEE 802.15.4 compliant, ZigBee ready radio

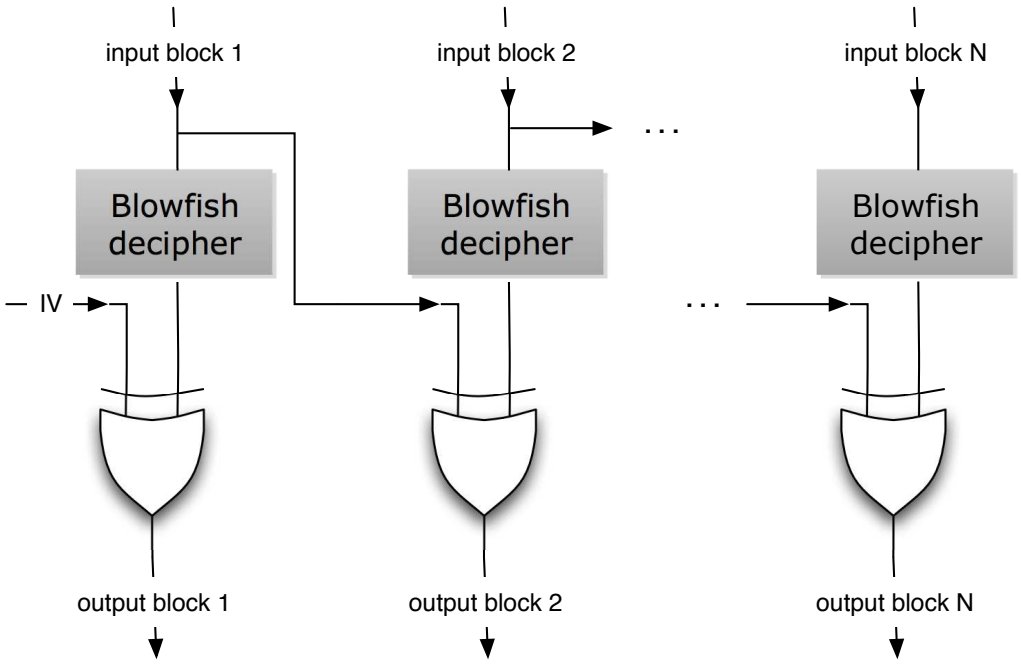

Figure 4.9: CBC decryption implementation

frequency transceivers which are integrated with an Atmega1281 micro-controller. The actual WSN nodes are connected to Crossbow MDA100 sensor and data acquisition boards [73] which provide a precision thermistor, a light sensor/photocell and a general prototyping area.

We expanded this platform by connecting to it a Xilinx CoolRunner-II CPLD. The CoolRunner-II CPLD family [74] utilizes Xilinx second-generation RealDigital technology so as to provide high performance, advanced features and low power consumption, all at a very low price. Featuring a 100% digital core, up to 323 MHz performance and ultra-low stand-by current, CoolRunner-II CPLDs offer a wide range of densities, plus abundant I/O, the flexibility to move from one density to another in the same package and the lowest cost per I/O pin in industry.

The specific prototyping CPLD board utilized is the Digilent X-Board [75] which is a complete circuit development platform for Xilinx CoolRunner-II CPLD. It contains all essential support circuits for the CoolRunner-II including an on-board USB2 port which provides a data port for CPLD configuration as well as for user data

transfers. This board includes a very low-cost 256 macrocell CoolRunner-II CPLD device (XC2C256) in a TQ-144 package while more than 75 CPLD signals are routed to expansion connectors so our designs can be easily extended.
The tool used to implement our design was Xilinx ISE 10.1 [76] while its embedded simulator was used in order to verify the correct operation of our architecture via the process of *behavioral simulation*. Next, we had to carry out *post-fit simulation* and, for this purpose, we preferred Modesim SE 6.3f [77]. Finally, the CPLD was programmed using the Digilent ExPort.
One of the most important issues when implementing our reference platform was the connection between the motes and the reconfigurable device. Regarding the CPLD connection, the JTAG ports were chosen for data transfers between the motes and the CPLD. For the mote connection, only 24 pins out of the 102 of the prototyping area are actually available since the remaining pins are either *open* or dedicated to a specific operation of the main mote micro-controller. Based on a traffic profiling of our applications, we decided to use 8 of those pins as an input to the mote, 8 for the output traffic and the remaining ones for several input/output control signals.
Regarding the actual system testing, the motes and the CPLD were programmed and connected together via a custom-made cable. Figure 4.10 presents our pioneering wireless sensor node platform.
The development of custom sensor applications is facilitated through the TinyOS 2.1 tools. TinyOS [78] is an operating environment designed to run on embedded devices used in distributed WSNs. For the programming of the sensor nodes, we have utilized the nesC language. NesC [79] (network embedded systems C) is a component-based, event-driven programming language which is an extension to the commonly used C one with components *wired* together to run applications on TinyOS.
As mentioned above, the I/O bus has limited size. Unfortunately, the available mote pins are only 24 and only 8 bits can be imported to or exported from the mote at the same time, but, in most applications, the input/output bandwidth requested is higher. Consequently, the use of a certain intercommunication protocol is necessary in order to implement an efficient platform.
For example, if the application implemented in the reconfigurable device requires 16-bit input and 16-bit output, the system should spend two phases in order to import the input data and additionally two for sending the output of the CPLD to the mote.

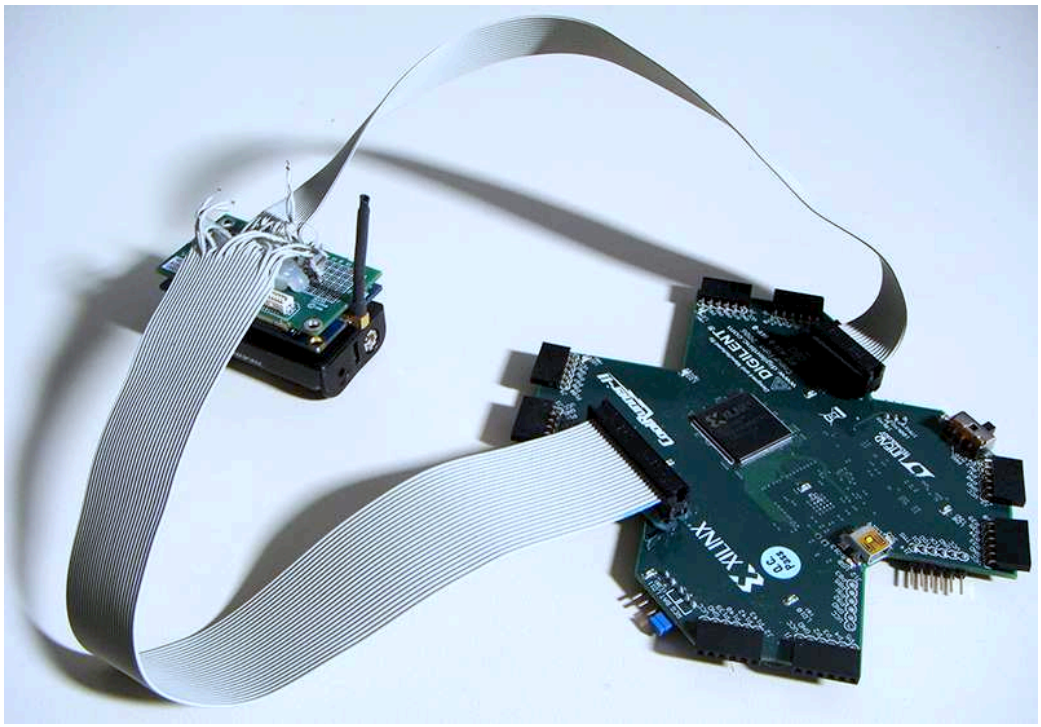

Figure 4.10: Wireless sensor node platform

In order to efficiently and correctly synchronize the data exchange between the CPLD and the mote, a simple toggle (hand-shake) synchronization protocol (see appendix 1) was also implemented; in particular a specific output toggle bit gets the inverted value of the input toggle bit when the correct result is ready.

4.1.2.2 Verification

The correct operation of our sensor node platform must be certified using a complete testbench and testing all its existing parts. First of all, hardware implementation of Blowfish cipher is tested using the *behavioral simulation* process and the specific simulator provided by the ISE tool. The testing samples used for this process are derived from the execution of the open-source software provided by the developers of Blowfish cipher [80]. After completing the simulation procedure for both the Blowfish encipher and decipher, the reconfigurable device is connected

with the wireless mote via the custom-made cable in order to test our platform in real-world experiments. Moreover, the software implementation of CBC encryption and decryption is integrated for supporting different block sizes. A total of plaintexts is collected and encrypted while the encrypted data is aggregated to messages which are sent to other nodes via the wireless link. The messages exchanged among the nodes can be captured by the Integration Wireless Platform Analyzer using a ZigBee dongle (Integration IA-OEM-DAUB1-2400 - ZigBee ready, 2.404 - 2.481GHz / IEEE 802.15.4), which is installed on a monitoring PC. These messages are transferred to the PC, so as to certify the correctness of the messages exchanged between the nodes that formed the network. For this purpose, the derived ciphertexts are also calculated manually. The same procedure is followed for certifying the correct functionality of CBC decryption too. The final step of the sensor node verification is to implement both encryption and decryption in different nodes. In this step, there are two groups of nodes; the first one consists of nodes that create different plaintexts, encrypt them and send the encrypted data via the link and the second one contains nodes that receive these messages with the encrypted data, decrypt them, send back a message with this decrypted data. All the exchanged messages among these nodes of the network can be captured by the monitoring PC where the decrypted data is compared with the initial plaintext.

4.2 Base station

4.2.1 Base station architecture

In contrast with a sensor node, base station needs a higher level of security because it usually collects the packets from all the sensor nodes and is more common target of attackers for obtaining information according to the network. For this purpose, a more powerful FPGA is embedded in every base station in order to improve its processing power and enable us to implement a stronger security algorithm such AES-128 due to the more available resources of the given FPGA compared to the CPLD ones.

This section describes the device that implements the AES-128 algorithm since it is optimal for this cipher encryption scheme while the next sections present the way each component of the whole system is implemented.

The top level of our design consists of two main module; the first one is the *cipher* module which implements the main part of the encryption procedure and the second

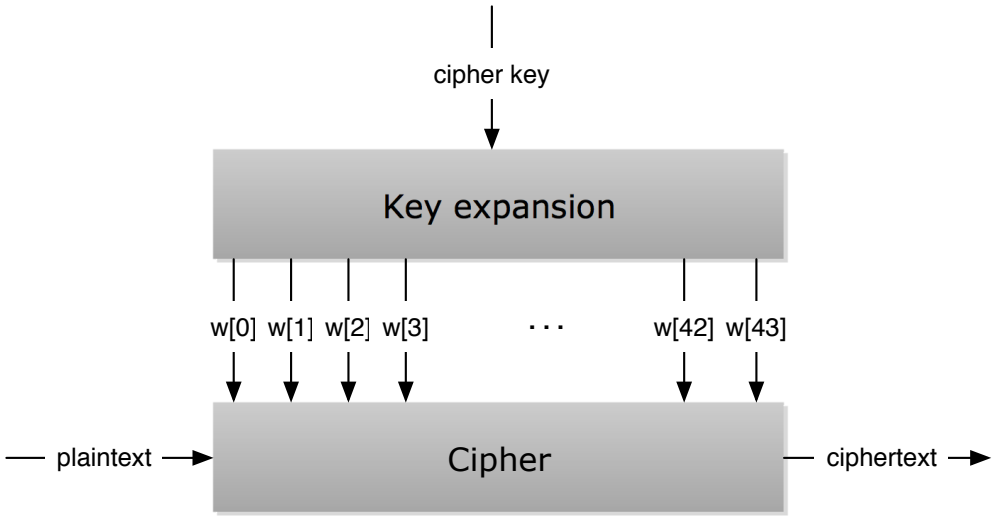

Figure 4.11: AES encryption block diagram.

one is the module that creates the key schedule required for this procedure and is called *key expansion*.
The AES-128 top level gets as input two 128-bit long quantities; the official data or *plaintext* and the *key* or *cryptovariable* used for creating its final output which is the 128-bit long encrypted data or the so-called *ciphertext*.
The block diagram of the AES encryption architecture is illustrated in the Figure 4.11.

4.2.1.1 Cipher

As referred above, this module implements the main part of the encryption procedure. The whole procedure is completed after 10 rounds. Initially, the input (plaintext) is copied to a state in order to create the initial one in which a round key is added using the 4 first keywords (32-bit long) deriving from the *Key expansion* module. Then, the first 9 cipher rounds are followed. Finally, the encryption process is completed with the last cipher round which contains only the three quarters compared to the previous rounds. Especially, the state is transformed through the *SubBytes*, *MixColumns* and *AddRoundKey* modules using the 4 last keywords from

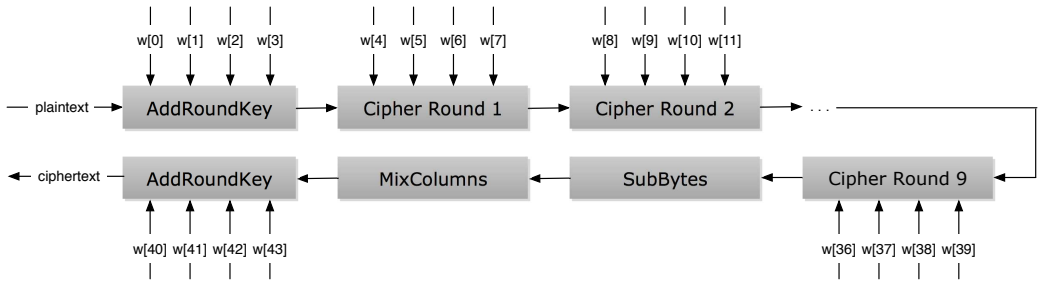

Figure 4.12: AES cipher block diagram.

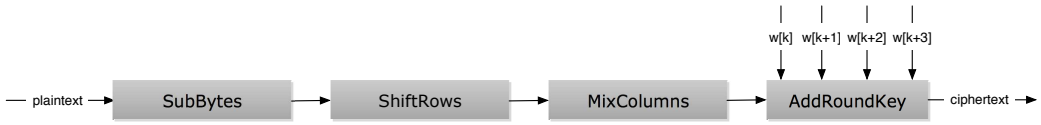

Figure 4.13: AES cipher round block diagram.

the key schedule in order the final result (ciphertext) of the encryption procedure to be created. Figure 4.12 presents the previously described block diagram of the *Cipher* module.

Each of the first 9 cipher rounds is implemented as shown in Figure 4.13. Its main input is the current state which is transformed using the four modules that implement the specific transformations presented in the previous chapter (*SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*) in order the new value of the *State* to be derived. This new value of the state is the output of every cipher round. Moreover, four keywords from the key schedule are imported to each round as input for adding them to the current through the *AddRoundKey* module. The specific keywords are selected according to the round number. Consequently, the keywords imported are *w[k]*, *w[k+1]*, *w[k+2]* and *w[k+3]* where $k = 4 \cdot round$ and $1 \leq round < 10$.

In the following subsections, the four modules used in order to implement the specific transformations are described in detail.

***SubBytes module*:** *SubBytes* module implements a non-linear transformation in which each byte of the *State* is substituted by another one using a substitution table (S-box). This modules consists of 8 S-boxes generated as dual-port block read-only

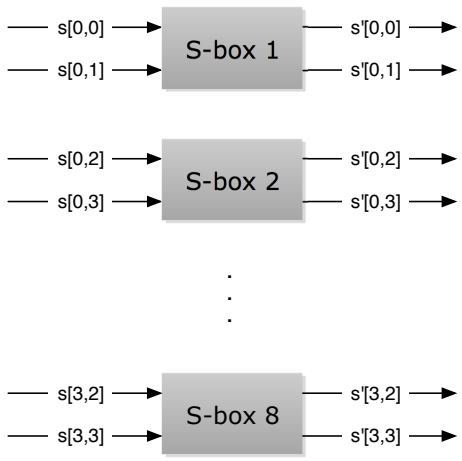

Figure 4.14: SubBytes module block diagram.

memories (DPROMs) containing 256 positions of 8-bit data. So the length of the DPROM input addresses and output data is 8 bits while each of the 16 input ports is used to substitute a specific byte of the *State*. These DPROMs are generated using Xilinx CORE Generator and initialized using a .coe file which contains the data of the table presented in Figure 3.10.
The block diagram of this module is illustrated in Figure 4.14, where *s[i,j]* and *s'[i,j]* are the bytes of the *i-th* row and *j-th* column of the *State* before and after the transformation, respectively.

***ShiftRows module*:** In this subsection, the module that implements the *ShiftRows()* transformation is described. The bytes in the last three rows of the *State* are cyclically shifted over different number of bytes (offsets). The first row is not shifted. Specifically, the second row is rotated left by one byte, the third by two and the fourth one by three. So the bytes move to "lower" positions in the row, while the "lowest" ones wrap around into the "top" of it.
ShiftRows module is implemented using three cyclic shifters that get as input the specific row of the *State* and rotate it by an offset according to its number. Figure 4.15 provides the block diagram used to implement this module, where *r[i]* and *r'[i]* are the *i-th* rows of the *State* before and after the transformation, respectively.

***MixColumns module*:** The *MixColumns* module operates on the *State* column-by-column, treating each column as a four-term polynomial. This module is imple-

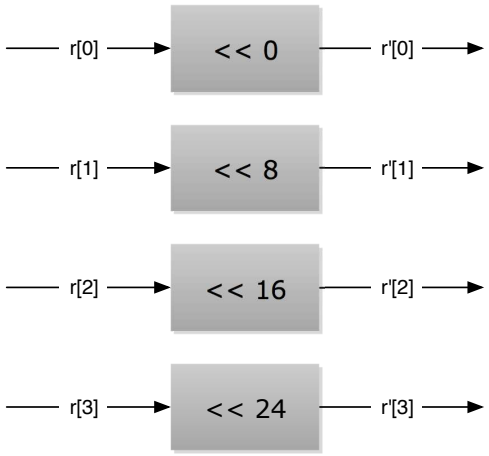

Figure 4.15: ShiftRows module block diagram.

mented using four components (*MixColumn* module) that take as input a column of the *State* and calculate its new value using the *MixColumn* transformation. The block diagram of the *MixColumns* module is shown in Figure 4.16, where *c[i]* and *c'[i]* are the *i-th* columns of the *State* before and after the transformation, respectively.
According to the *MixColumn* component, the columns are considered as polynomials over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by

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

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$\begin{aligned}
s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \\
s'_{3,c} &= (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c})
\end{aligned}$$

Taking into account that $\{03\} \cdot x$ can be analyzed as $\{02\} \cdot x \oplus \{01\} \cdot x$ and $\{01\} \cdot x$ is equal to x, the previous equations can be written as follows:

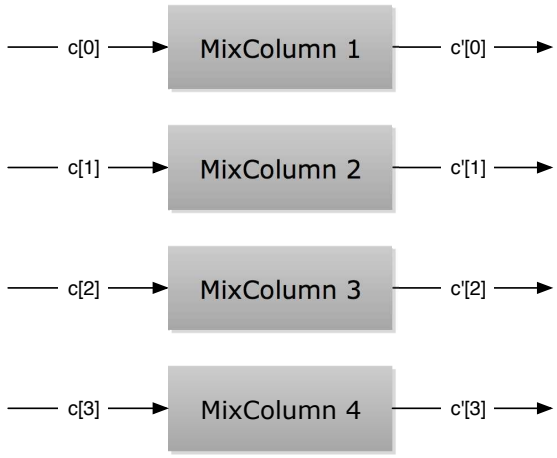

Figure 4.16: MixColumns module block diagram.

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{02\} \cdot s_{1,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{02\} \cdot s_{2,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{02\} \cdot s_{3,c}) \oplus s_{3,c} \\
s'_{3,c} &= (\{02\} \cdot s_{0,c}) \oplus s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c})
\end{aligned}
$$

Furthermore, $\{02\} \cdot x$ can be calculated using the *xtime()* transformation and, consequently, the above equation table can be changed as follows:

$$
\begin{aligned}
s'_{0,c} &= xtime(s_{0,c}) \oplus xtime(s_{1,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus xtime(s_{1,c}) \oplus xtime(s_{2,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus xtime(s_{2,c}) \oplus xtime(s_{3,c}) \oplus s_{3,c} \\
s'_{3,c} &= xtime(s_{0,c}) \oplus s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus xtime(s_{3,c})
\end{aligned}
$$

According to the aforementioned simplifications, *MixColumn* component is implemented with the use of four XOR gates that have five 8-bit long inputs each one and four modules which implement the *xtime()* transformation. Its complete block diagram is illustrated in Figure 4.17, where *c[i,j]* and *c'[i,j]* are the bytes of the *i-th* row and *j-th* column before and after the transformation, respectively.
Finally, it is indispensable to describe the operation of the *xtime()* transformation and the way implemented for providing a full view of the *MixColumns* module. Multiplication by x can be implemented at the byte level as a left shift and a subsequent

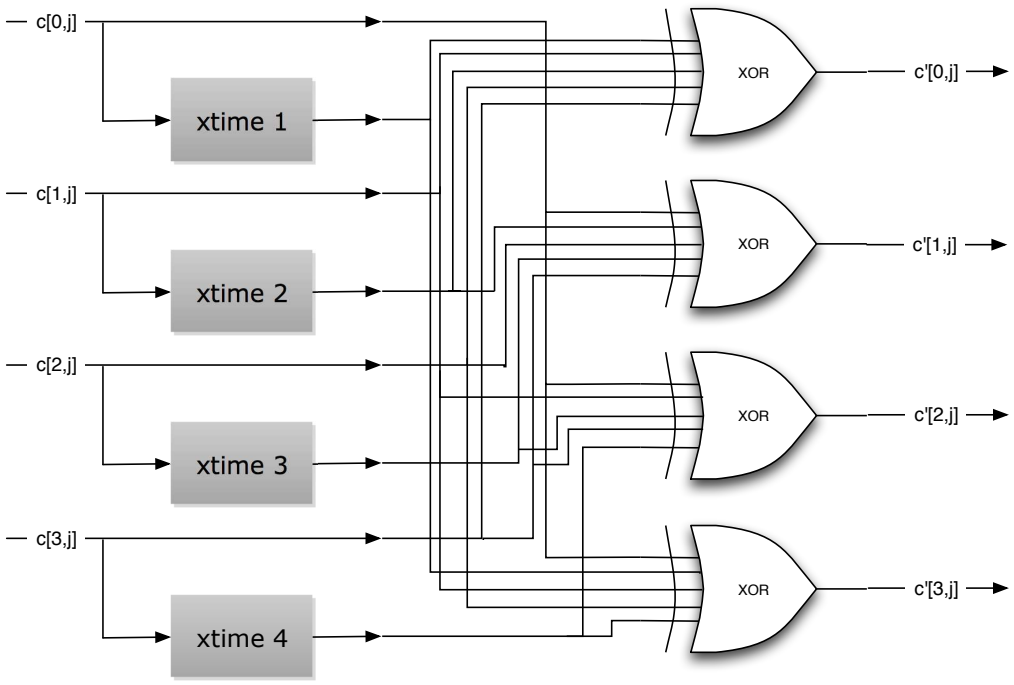

Figure 4.17: MixColumn module block diagram.

conditional bitwise XOR with $\{1b\}$. This operation on bytes is denoted by *xtime()*. Multiplication by higher powers of x can be implemented by repeated application of *xtime()*. By adding intermediate results, multiplication by any constant can be implemented. Consequently, *xtime* component is implemented using a left shifter where the input byte is shifted by one bit, a 2-to-1 multiplexer with 8-bit inputs used to choose the specific constant for the subsequent XOR operation according to the MSB of the input byte and a XOR gate of two 8-bit inputs in which the outputs of the other two components are XORed in order to create the final result of the *xtime* module.

Figure 4.18 shows the block diagram used in order to implement the *xtime()* transformation, where *c[i,j]* and *c'[i,j]* are the bytes of the *i-th* row and *j-th* column before and after the transformation, respectively. In addition, *c[i,j](7)* is the MSB of the input byte.

***AddRoundKey module*:** In this module, a *Round Key* is added to the *State* by

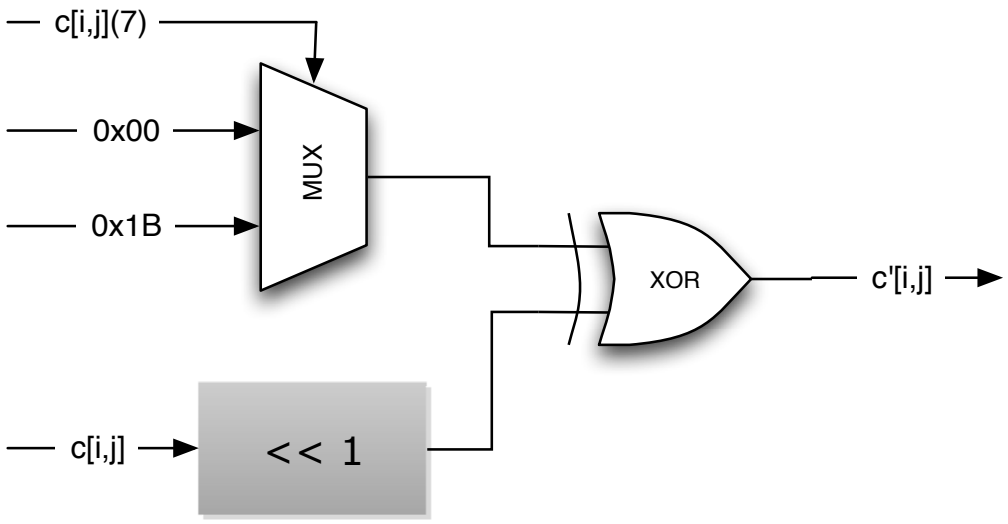

Figure 4.18: xtime module block diagram.

a simple bitwise XOR operation. Each *Round Key* consists of four words from the key schedule deriving from the *key expansion* procedure. Those keywords are added into the columns of the *State*. Consequently, *AddRoundKey* module is implemented with the use of four XOR gates with two 32-bit inputs each one, where every column of the *State* is XORed with the specific keyword selected according to the number of the current cipher round. The block diagram of this module is presented in Figure 4.19, in which *c[i]* and *c'[i]* are the columns of the *State* before and after the transformation, respectively, and *w[k]*, *w[k+1]*, *w[k+2]* and *w[k+3]* are the four words from the key schedule, where $k = 4 \cdot round$ and $0 \leq round \leq 10$.

4.2.1.2 Key expansion

Using this module, the AES algorithm takes the *Cipher Key* and performs a *key expansion* routine to generate a key schedule. This component generates a total of 44 words, four of which are the initial set required from the algorithm and the remaining ones are used in each of the ten rounds of the cipher process. The resulting key schedule consists of a linear array of 4-byte words, denoted as *w[i]*,

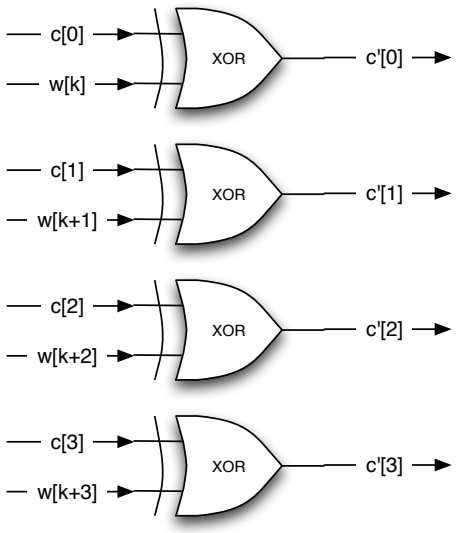

Figure 4.19: AddRoundKey module block diagram.

with i in the range $0 \leq i < 44$.
The expansion of the input key into the key schedule proceeds as shown in Figure 4.20.
Initially, the first four keywords (*w[0]*, *w[1]*, *w[2]*, *w[3]*), which are the initial set required from the cipher procedure, derive from the four words of the *Cipher Key*. The remaining 40 keywords are calculated in groups of four (*w[k]*, *w[k+1]*, *w[k+2]*, *w[k+3]* for $k = 4 \cdot round$ and $1 \leq round \leq 10$), where the *w[k-1]* is left rotated by 8 bits (implementation the *RotWord()* transformation), its bytes are replaced using *SubWord()* transformation, it is XORed with the round constant word array, *Rcon[i]*, and the value of *w[k-4]* in order to create the value of *w[k]*. The remaining three words are calculated as follows:

$$w[k+1] = w[k] \oplus w[k-3]$$
$$w[k+2] = w[k+1] \oplus w[k-2]$$
$$w[k+3] = w[k+2] \oplus w[k-1]$$

Furthermore, the structure of the component that implements the *SubWord()* transformation is similar to that of the *SubBytes* module. The only difference between

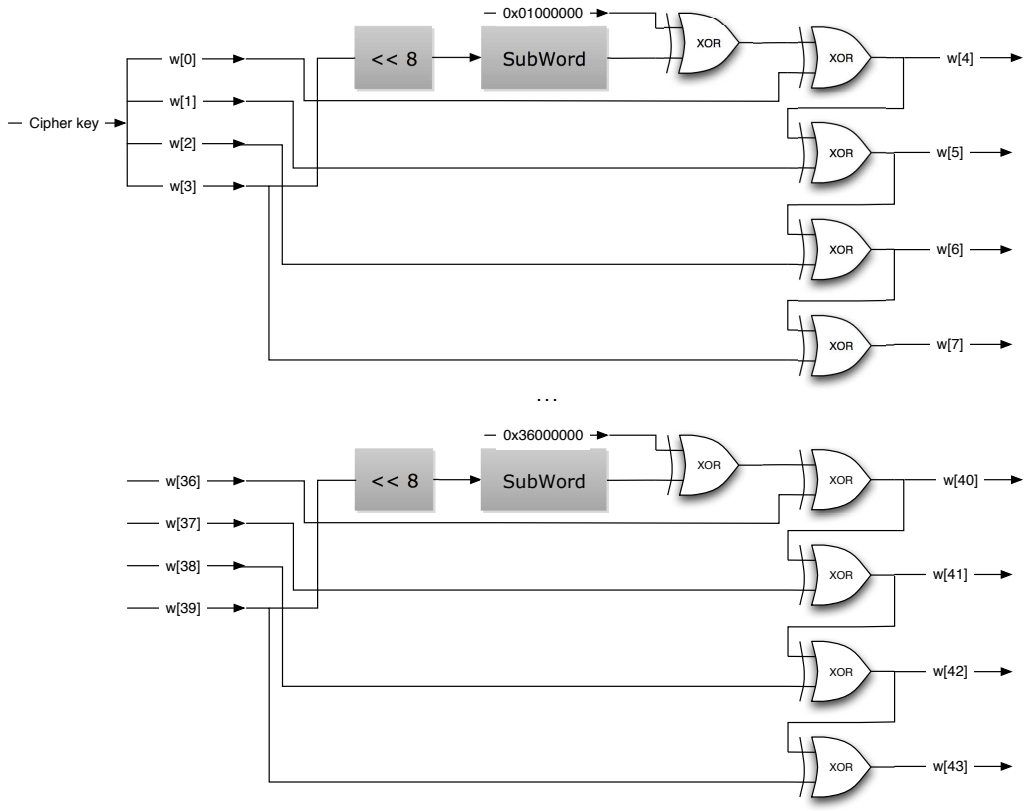

Figure 4.20: Key expansion block diagram.

them is that the input of *SubBytes* is 128-bit (16-byte) long in contrast with the one of *SubWord* that is 32-bit (4-byte) long. So only two DPROMs are required for the implementation of *SubWord* in contrast with the eight ones used for the *SubBytes* implementation. Moreover, 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 of $GF(2^8)$.
Consequently, the key expansion is implemented using ten left cyclic shifters, ten *SubWord* modules (or 20 DPROMs) and 50 XOR gates with two 32-bit inputs each one, as shown in Figure 4.20.

4.2.2 Base station implementation

Except for a stronger cipher, such AES, our base station must be well-protected from other attacks too. As mentioned in chapter 2, cryptographic algorithms are implemented on specific devices in order to mitigate the threats to the network, but these solutions make them vulnerable to side-channel analysis, such timing and power consumption analysis. We are going to concentrate to differential power analysis.
This section describes six different FPGA-based implementations of the AES-128 cipher used for improving the security of our system and making it less susceptible to such threats. The following subsections present the changes and the additions made in each implementation compared with the system architecture that is previously referred. Finally, the two lasts subsections provide some implementation details and information for the verification of this base station.

4.2.2.1 Single-rail

Single-rail implementation is the simplest one from AES designs. No difference compared with the system architecture that is described in the previous chapter exists. It is a fully pipelined implementation designed in order to improve the value of the system throughput.

4.2.2.2 Dual-rail

The next step for achieving better results in the power consumption of our system is to apply the very common technique of dual-rail in the given system architecture. This step enable us the creation of two new implementations; the single and the dual (or alternating) spacer dual-rail one, which are presented to the following subsections.
***Single spacer dual-rail*:** According to the single spacer dual-rail implementation, all the bits of the architecture quantities are changed and, consequently, all the bitwise gates have to be replaced by others that calculate the specific results based on 2-bit arithmetic.
So the value of every bit is changed from {0} and {1} to {01} and {10}, respectively. This process is made using a single-to-dual rail converter, the structure of which is very simple. The MSB of the new quantity is equal to the initial one and the LSB to

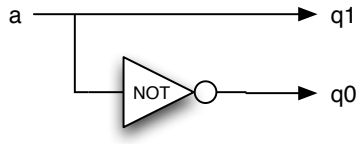

Figure 4.21: Single-to-dual rail converter block diagram.

Table 4.1: Single-to-dual rail converter truth table.

a	$q_1 q_0$
0	01
1	10

its inverse. The block diagram and the truth table of this converter are illustrated in Figure 4.21 and Table 4.1, respectively.

Except for this conversion of bits into 2-bit quantities, there is a total of many changes in the logic gates of our architecture that have to be made. Especially, a new structure for NAND and XOR gates have to be defined in order to support the dual-rail method with its 2-bit quantities. Moreover, the NCL approach is used for the conversion of these circuits from single-rail to dual-rail.

Regarding the NCL NAND gate, four single AND and a OR gates are utilized for its implementation. As it can be derived from the truth table of this gate shown in Table 4.2, all the possible of two inputs bits combinations are logically conjuncted and the results of the three out of four AND gates are disjuncted in order to calculate the value of the final result MSB. Referring to the final result LSB, it is equal to the result of the fourth AND gate (logic conjunction of the two inputs MSBs). This structure is presented in Figure 4.22.

Another logic gate that is widely used in AES architecture is the XOR one and a redefinition according to the rules of dual-rail circuit and NCL approach is though indispensable. Consequently, an NCL XOR gate is implemented using the previously defined NCL NAND one. Especially, four NCL NAND gates must be used in order to implement this new circuit to follow the values of its truth table shown in Table 4.3. Initially, the two inputs are negatively conjuncted while the results of this gate is negatively conjuncted with both the two inputs, separately. The results

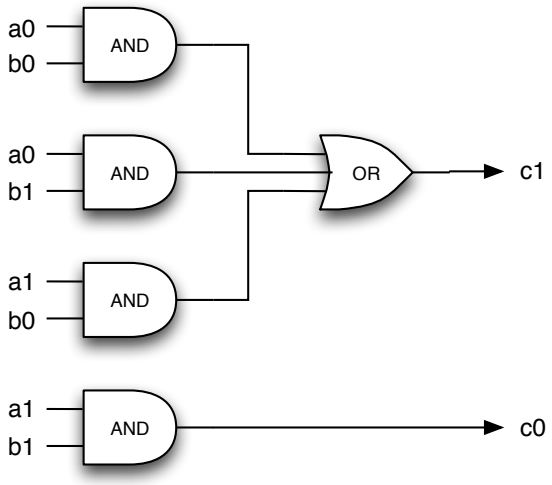

Figure 4.22: NCL NAND gate block diagram.

Table 4.2: NCL NAND gate truth table.

a_1a_0	b_1b_0	c_1c_0
01	01	10
01	10	10
10	01	10
10	10	01

of the two last NCL NAND gates are NANDed in order to reach the final result of the whole NCL XOR gate. The block diagram of this structure is illustrated in Figure 4.23.

In hardware implementations, a positive gate is usually constructed out of a negative gate and an inverter. In addition, the total area overhead in dual-rail logic is more than twofold compared to single-rail. The use of positive gates is not only a disadvantage for the size of a dual-rail circuit, but also for the length of the critical path. This is the reason we implement NCL NAND gates instead of AND ones and use them in order to define a new structure for the NCL XOR gate. Thus, the use of positive gate is inevitable in some case. Consequently, a new method for negative gate optimization must be found.

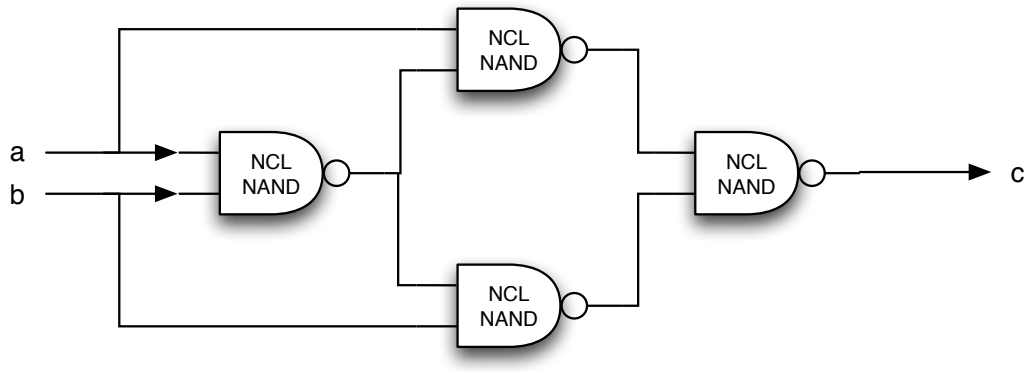

Figure 4.23: NCL XOR gate block diagram.

Table 4.3: NCL XOR gate truth table.

a	b	c
01	01	01
01	10	10
10	01	10
10	10	01

In order to optimize a dual-rail circuit for negative gates, the following transformations should be applied: First, all gates of positive dual-rail logic are replaced by negative gates. Then, the output rails of those gates are swapped. So NCL NOT gate can be optimized and implemented as a swap of the two bits of every quantity, taking advantage of the structure of the dual-rail circuits. The block diagram and the truth table of this optimized negative gate is illustrated in Figure 4.24 and Table 4.4.

Shifters in single spacer dual-rail design is changed and their inputs are shifted (either cyclically or not) by twice the bits they shifted in the given architecture. For example, a shifter that moves its input by 8 bits is replaced by another one that shifts it by 16. Moreover, all the inputs and outputs of the components and the intermediate signals used by the AES architecture are doubled in length. So the inputs (plaintext and cipher key) and the output (ciphertext) of the AES encryption

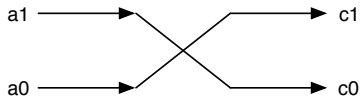

Figure 4.24: NCL NOT gate block diagram.

Table 4.4: NCL NOT gate truth table.

a	c
01	10
10	01

are 256-bit long.

Another critical issue about this dual-rail implementation is the conversion of *SubBytes* module. In contrast with the other parts of the architecture, *SubBytes* module does not change in this implementation, because it would be infeasible to use so many S-boxes with 16-bit long input and output. This memory change means that every S-box would consists of $2^{16} = 65536$ positions of 16-bit long data and, consequently, 1Mbit of memory. Taking into account that the number of the S-boxes used in our architecture is equal to 100, our design had to be implemented in a FPGA-device with 100Mbit memory, but such a device does not exist in retail. Furthermore, only 2^8 out of 2^{16} positions would contain useful data. For all these reasons, we prefer not to change the *SubBytes* module. In order to use this module in the dual-rail implementation, two converters are placed to its input and output for converting dual-rail data to single-rail one and vice versa. The output converter that changes single-rail data in dual-rail one is the same to the previously described one. Referring to the input converter, every 2-bit quantity is converted in a specific bit using a circuit consisting of a NOT and a NOR gate. The final bit is the result of the NOR between the LSB and the inverse MSB of the initial quantity. The block diagram and the truth table of this converter are presented in Figure 4.25 and Table 4.5, respectively.

***Dual spacer dual-rail*:** In order to balance the power signature, the use of two spacers is proposed. As opposed to single spacer dual-rail, where a particular rail is switched up and down (i.e., the same gate always switches) in each cycle, in the *alternating spacer protocol*, both rails are switched from *all-zeros* spacer to

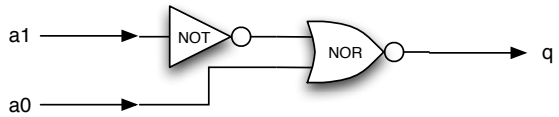

Figure 4.25: Dual-to-single rail converter block diagram.

Table 4.5: Dual-to-single rail converter truth table.

a_1a_0	q
01	0
10	1

all-ones spacer and back. The intermediate states in this switching are code words. In the scope of the entire logic circuit, this means that, for every computation cycle, we always fire all gates forming the *dual-rail* pairs. This makes the circuit more resistant to DPA.

The new alternating spacer discipline can be directly applied to the implementation techniques by using a single-to-alternating spacer converter, the structure of which is presented in Figure 4.26. This component consists of a alternating spacer dual-rail flip-flop (including simple AND and OR gates) and an OR gate. It also uses a toggle to decide which spacer to inject all-ones or all-zeros. The toggle can be constructed out of two latches, as shown in Figure 4.27.

Consequently, in order to convert the implementation described in the previous subsection into a dual spacer one, we must insert this converter in every circuit used. Unfortunately, this is infeasible due to the limited resources of the given FPGA device. A dual spacer dual-rail implementation for the AES cipher encryption might require more than ten times the available logic slices of the specific device. So the idea of this specific implementation has been abandoned prematurely.

4.2.2.3 Duplicate dual-rail

Due to the failure of the last idea of a dual spacer dual-rail implementation, we must find another solution for balancing the power consumption of AES algorithm. Studying the results derived from the single spacer implementation, we

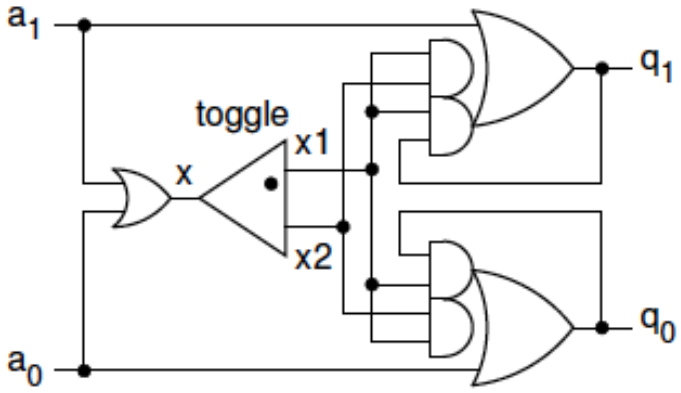

Figure 4.26: Single-to-alternating spacer converter block diagram.

conclude that this method does not perform well for cases that the combination of plaintext and cipher key consists of a small number of *ones*. Consequently, we decide to use the *duplication* method. Another implementation of the AES cipher encryption (including key expansion routine) is placed to run in parallel with the initial one while this duplicated implementation takes as input the inverse of the plaintext and the cipher key. Unfortunately, another problem has derived in this implementation too. A full AES procedure of ten cipher rounds cannot be implemented to run in parallel with the initial one due to the memory blocks required for this implementation that are more than the available ones included in the specific FPGA device. For this purpose and taking into account that the result of this second implementation does not affect the final result of the encryption, we decide to implement some other designs consisting of two, four and eight cipher rounds in order to study their performance based on the power consumption balance. Additionally, the remaining rounds are replaced with pipeline registers in order the result of these parallel implementations to be created in the same clock cycle that the ciphertext is derived from the initial implementation. Furthermore, the output bits of these additional implementations are conjuncted in order to drive an output bit of our system. Otherwise, the development tools used would trim this component because it does not affect the general output of the system. Finally, the key expansion routine of these implementations is implemented in such a way to have

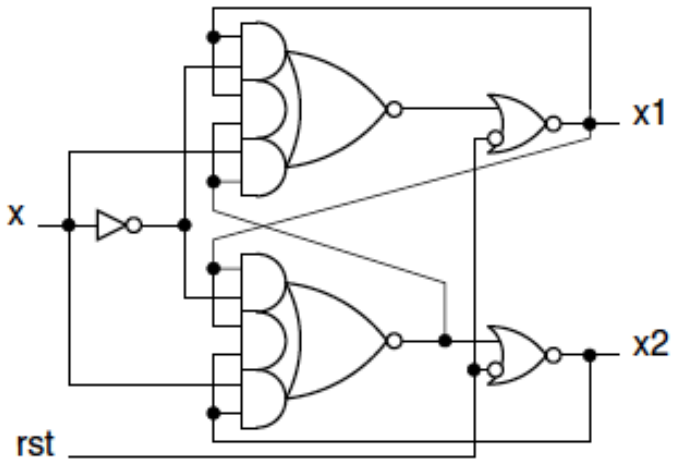

Figure 4.27: Toggle block diagram.

less rounds according to the key schedule requirements. So the key expansion routines produce a key schedule of 12, 20 and 36 keywords for the 2-, 4- and 8-round duplicate dual-rail implementations, respectively.
Figure 4.28 illustrates an abstract block diagram of AES duplicate DR implementations.

***2-round duplicate dual-rail*:** A parallel implementation with two cipher rounds plus eight pipeline registers and a key expansion routine that produces a key schedule of 12 words is added to the previous designs in order to balance the power consumption. No further changes are needed to create a new 2-round duplicate dual-rail system.

***4-round duplicate dual-rail*:** Moving to the 4-round duplicate dual-rail implementation, the additional component contains four cipher rounds, six pipeline registers and a 20-word key schedule. Moreover, a minor change is made to fulfill this new design. The blocks of memory required are more than the available ones, but the number of the used logic slices is limited. Consequently, some of the required memories (S-boxes) are replaced from distributed ones (look-up tables constructed by logic slices) which are generated by the Xilinx CORE generator. A register is also added after the output of these memories in order to synchronously read data

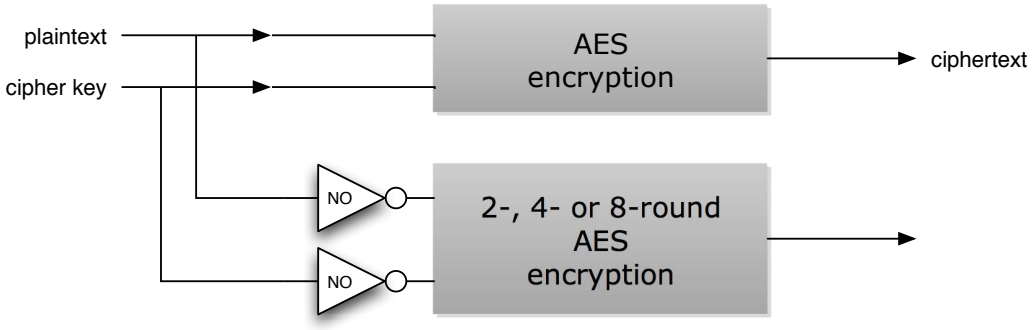

Figure 4.28: AES duplicate DR block diagram.

such the block memories that are used by the remaining parts of our system.
***8-round duplicate dual-rail*:** The last implementation described by this work is the 8-round duplicate dual-rail one. This implementation is the most sophisticated compared to the previous ones and achieves the best results according to the balance of power consumption. Four cipher round is added to the system referred to the previous subsection and the specific pipeline registers are removed. Referring to its key expansion routine, its key schedule contains 36 words. Finally, the S-boxes of the additional cipher rounds are implemented as described in the previous subsection using distributed memories.

4.2.2.4 Implementation details

In order to create a complete view for the implementations presented in this chapter and their development, it is though indispensable to point out some implementation details.
First of all, VHSIC (Very High Speed Integrated Circuits) hardware description language (VHDL) [81] and Xilinx ISE 10.1i are used during the whole implementation procedure. ISE different tools enable us to develop, synthesize and implement our designs while its embedded simulator (ISE simulator) is used for verifying their correct functionality. After completing the implementation procedure in each of the above designs, the area cost and the performance (clock period and frequency) are measured using the synthesis and place and route tools of ISE.
Regarding the reconfigurable hardware utilized for implementing the above sys-

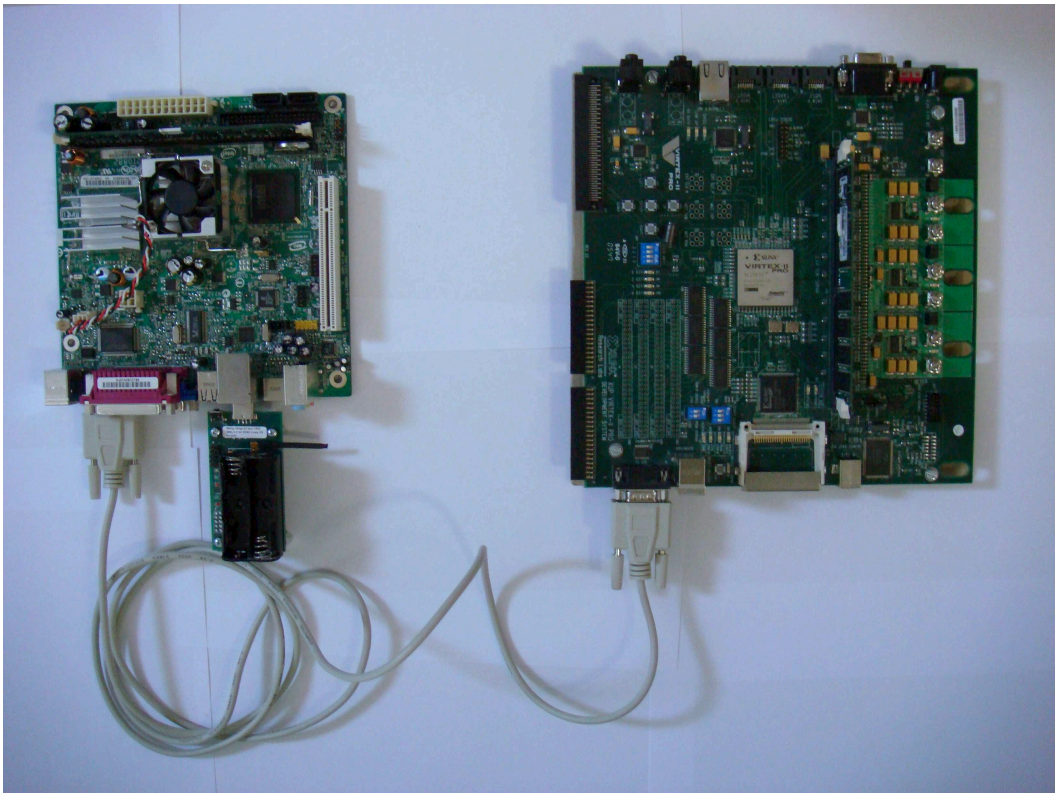

Figure 4.29: Base station top view

tems, we have selected a Virtex-II Pro Evaluation Platform based on the Digilent XUPV2P Development System [82]. Digilent XUPV2P is a feature-rich general purpose evaluation and development platform with on-board memory and industry standard connectivity interfaces. It features a Xilinx Virtex-II Pro "XC2VP30" [83] FPGA device supporting USB host, peripheral controllers, programmable system clock generator and many other I/O devices including RS-232 port. This specific FPGA device consists of 30,816 Logic Cells, 136 18-bit multipliers, 2,448Kb of block RAM and two PowerPC processors. The speed grade of the device used is equal to -6. The reason behind selecting this specific development platform for implementing and evaluating our systems is because it provides a complete framework to measure the power consumption on the reconfigurable device due to its external connectors of the voltage powers which enable us to power the FPGA using an external device. Furthermore, Xilinx iMPACT accessory is the tool used in order to program the

FPGA device by downloading the specific bitstream. According to the I/O problem, the serial port (RS-232) of the development board is used for importing/ exporting data to/ from our reconfigurable system. For this purpose, a new I/O component is designed, implemented and placed in the reconfigurable device (see appendix 2) in order to take the role of the interface between the AES algorithm implemented on the FPGA board and the external world.
The previously described device is connected with an Intel Desktop Board D945GCLF2 that contains an integrated Intel Atom processor [84]. The Intel Desktop Board D945GCLF2 is designed to support Internet-centric computing in a Mini-ITX form factor using the Intel 945GC Express Chipset. Besides that, our board is equipped with a 1-GB DDR2 RAM module and an external SATA2 hard disk drive.
The connection of the previously described devices can be made via RS-232 port. In our platform, we connected the two development systems with a serial cable, utilizing the RS-232 ports of each device at 115200 Kbps.
As far as the wireless part of our innovative platform is concerned, we used the Crossbow MIB520CB USB Gateway connected to a ZigBee card.
Figure 4.29 shows the top view of our wireless base station platform.
Apart from the hardware modules utilized in our WSN base station, a software suite was also developed in order to enhance our platform with the appropriate functionality. To begin with, one of the most crucial issues was the correct selection of the Operating System of the Intel Atom Board. The operating system should be as minimal as it can so as to meet the WSN need for low power consumption; as a result, Linux Xubuntu 8.10 with Linux kernel 2.6.27 was selected which is claimed to be appropriate for low power solutions [85]. A python software suite was developed on the top of this OS which controls the efficient and correct data transfer between the FPGA and the ZigBee interface. *Python* [86] is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools and comes with extensive standard libraries. For our application, we used the open-source *PySerial* and *Socket* python libraries. *PySerial* [87] is a library which provides support for serial connections over a variety of different devices: old-style serial ports, Bluetooth dongles, infra-red ports, and so on. In our case, *PySerial* provides all the necessary functions for the communication between the Atom Processor and the FPGA Board, whereas the *Socket* [88] library provides access to the BSD socket interface. The *Socket* python library is also utilized for the interconnection with the ZigBee card,

through the *SerialForwarder* Tool, which is described later in this subsection.
Furthermore, the *SerialForwarder* program opens a so called *packet source* and let many applications connect to it over a TCP/IP stream. For example, a *SerialForwarder* whose packet source is the serial port can be executed; instead of connecting to the serial port directly, applications connect to the *SerialForwarder*, which acts as a proxy to read and write packets. Since the applications connect to the *SerialForwarder* over TCP/IP, those applications can also connect over the Internet. In general, the base station is used to receive, process, forward and, optionally, store data packets using all the aforementioned software tools.
Figure 4.30 presents the flowchart of our basic software suite while Figure 4.31 presents the complete software stack of our development tools.
Utilizing our development tools, any application implemented on the Atom CPU, the FPGA or a combination of the two, can easily process incoming packets from the motes. Upon a packet is received, our suite reads its size and the actual data itself. The received packet, then, is either processed by the software executed on the Atom or written to the RS-232 port in order to be processed by the FPGA. When an incoming packet is monitored in the RS-232 port, our software suite reads it, forms it according to the *SerialForwarder* Protocol and sends its size and the actual data to the socket, so as to either be further processed by the software executed on the Atom or be broadcast to the air via the ZigBee card. Our development tools can also store this packet to a database.

4.2.2.5 Verification

A critical issue in the development process of our designs was the verification. In order to certify the correct functionality of our implementation, complete testbenches had to be created. AES specification [36], as most of such documents, contain examples that provide us data for testing procedure. These examples give the input values of different *plaintexts* and *cipher keys* and the value of the *ciphertexts* that must derive after the whole encryption process. Moreover, all the intermediate values of the *State* after each transformation during the cipher procedure are provided by the authors of the specification and they were used in order to debug our designs. Apart from the examples referring to the cipher procedure, some examples for the second part of the encryption, the key expansion routine, also exist. These additional examples are used for debugging our implementations

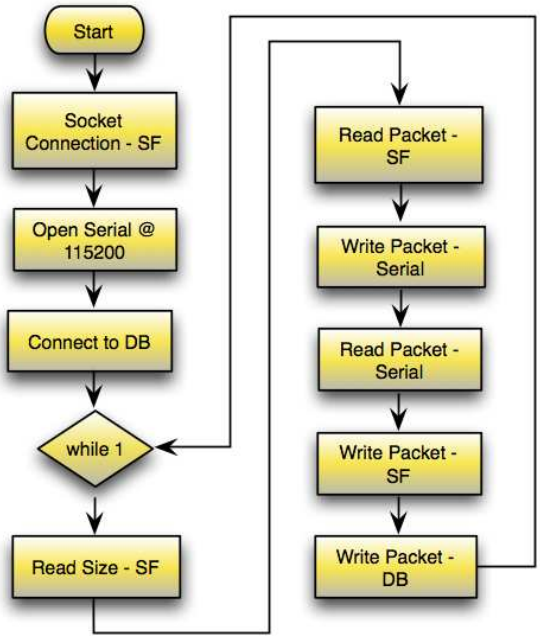

Figure 4.30: Python script flowchart of base station SW suite

too.

The process of verification mainly consisted of two steps; the behavioral simulation and the testing on-board. Firstly, taking into account the input and output data proposed by the algorithm specification, we created an complete testbench and, then, every of our implementations was simulated using the specific tool referred to the previous subsection. After completing the simulation process and certifying that the results derived from it are the desirable ones, we kept on with the on-board testing. All the testing values used from the previous process were embedded to the specific software (python script) and, after connecting the development platform with a PC via a serial port, this software enabled us to send the input values we wanted to the FPGA. After the completion of the encryption procedure, the FPGA sent the deriving output values back to the PC where the specific software depicted them to its monitor while these values were compared with the desirable ones.

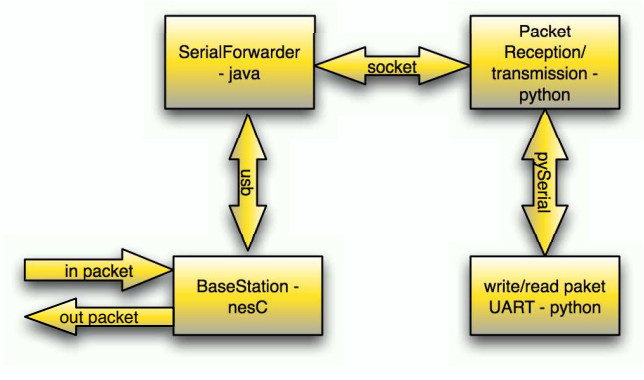

Figure 4.31: Base station development tools

Chapter 5

Performance

In previous chapters, the different sensor node and base station platforms and their system architectures are presented. We refer to the idea on which every architecture is based and, then, we describe the design process plus the implementations of it. In this chapter, we have to evaluate these implementations and illustrate their energy and power consumption as well as the performance (throughput) and the area cost. These values will give us a clear view of our designs.

Firstly, the sensor node platform is evaluated based on three major metrics: execution time, energy and maximum power consumption. All these are critical parameters in WSNs, since it is certainly desirable to increase the limited processing power of the node while also increasing the life time of the wireless mote by lowering the energy and maximum power consumption.

Referring to the base station platform, the different systems are evaluated based on two major metrics: maximum and average power consumption. All these are critical parameters in order to show how data independent the power consumption of the initial design is and how can be improved by applying the previously presented methods, since it is certainly desirable to make our cryptographic systems well-protected from differential power analysis.

Our performance results are based on real-world experiments in which a mixed signal oscilloscope has been used in order to take the power measurements. An extra signal has been used in both the software and the hardware implementations of the specific application in order to measure the execution time; this signal transits to high when the execution of the specific process starts and then toggles back to low, when the process ends. Regarding our base station, it is significant to point out that the measured results referred only to the encryption process (the calculation of ciphertext) and that the procedure of receiving input or sending does not been taken into account.

The framework that is used for taking the required power measurements is mainly consisted of a high precision resistance that is placed in row with the power supply

of the measured system (sensor node CPLD or base station FPGA). The FPGA of our base station is powered using an external DC supplier of 1.5 V while the value of the reference resistance used is equal to 0.1 Ω.
First of all, the energy consumption is calculated using the integral of the measured voltage V_m for the measured execution time period $\Delta\tau$. The result is divided with the reference resistance R_{ref} in order to calculate the reference current I_{ref}.
Multiplying the I_{ref} with the reference voltage V_{ref} that is equal to 2.7 V for the mote and 3.3 V for the CPLD, the overall energy consumption is calculated based on the formula below.

$$E = I_{ref} \cdot V_{ref}, \text{ where } I_{ref} = \frac{\Sigma_i V_{m,i} \Delta\tau}{R}$$

Furthermore, the maximum power consumption is calculated by multiplying the reference voltage of the system V_{ref}, which is equal to 3.3 V for the FPGA, with the maximum measured value of the current $I_{m,max}$, which is calculated by the division of the maximum measured value for the voltage $V_{m,max}$ with the reference resistance R_{ref} . The maximum power consumption is calculated based on the formula below.

$$P_{max} = I_{m,max} \cdot V_{ref}, \text{ where } I_{m,max} = \frac{V_{m,max}}{R_{ref}}$$

Regarding average power consumption, its calculation is derived by the multiplication of the reference voltage V_{ref} and the average measured value of the current $I_{m,avg}$, which is calculated by the division of the average measured value for the voltage V_{avg} with the reference resistance R_{ref}. The actual equation used is the one below.

$$P_{avg} = I_{m,avg} \cdot V_{ref}, \text{ where } I_{m,avg} = \frac{1}{N} \cdot \frac{\Sigma_{i=1}^{N} V_{m,i}}{R_{ref}}$$

5.1 Sensor node results

Regarding sensor node, we measure four different implementations of Blowfish plus CBC encryption and decryption. These four implementations differ only in the block size. Consequently, encryption and decryption of 16, 32, 64 and 128-bit long blocks are used for the evaluation of our sensor node platform based on execution time, energy and maximum power consumption. Tables 5.1 and 5.2 illustrate

Table 5.1: Blowfish encryption results

Block size (bits)	**Execution time (us)**		**Energy consumption (uJ)**		**Maximum power consumption (mW)**	
	Mote	**Mote plus CPLD**	**Mote**	**Mote plus CPLD**	**Mote**	**Mote plus CPLD**
16	75.0	79.8	106.6	7.2 (Reduction: 93.3%)	446.9	196.1 (Reduction: 56.1%)
32	169.2	546.0	182.0	14.0 (Reduction: 92.3%)	443.4	206.3 (Reduction: 53.5%)
64	337.0	954.0	313.6	29.5 (Reduction: 90.6%)	433.1	218.6 (Reduction: 49.5%)
128	955.0	2735.0	507.4	58.0 (Reduction: 88.6%)	417.7	210.4 (Reduction: 49.6%)

the deriving results for these three metrics of Blowfish encryption and decryption, respectively.

As these tables show, the time needed for the calculation of the final result (ciphertext for the encryption procedure or plaintext for the decryption one) from the new platform including the CPLD is much larger than that needed from the initial mote. Especially, in most of the cases, this execution time of the proposed platform is three times larger in both Blowfish encryption and decryption. Thus, the low reference current provided by the CPLD leads us to a significant reduction in energy and maximum power consumption. For the encryption procedure, the proposed scheme consumes around 50% less power than the single mote and, subsequently, it performs an important reduction in energy consumption that is around 90%. Moving to the decryption procedure, the maximum power consumption is decreased at the same level as in the encryption one while the reduction in energy consumption achieved by our sensor node platform is a little smaller than the specific one in encryption, but it still remains high (more than 80%). Another issue that must be pointed out is that the execution time and the energy consumption in both processes and platforms increase relatively to the block size (size of input/ output) while the maximum power consumption remains almost invariable.

Figures 5.1, 5.3 and 5.5 present the results of the encryption process in a schematic way while Figures 5.2, 5.4 and 5.6 schematically illustrate the previously referred

Table 5.2: Blowfish decryption results

Block size (bits)	**Execution time (us)**		**Energy consumption (uJ)**		**Maximum power consumption (mW)**	
	Mote	**Mote plus CPLD**	**Mote**	**Mote plus CPLD**	**Mote**	**Mote plus CPLD**
16	48.9	136.8	47.1	8.8 (Reduction: 81.2%)	417.7	222.8 (Reduction: 46.7%)
32	99.4	275.5	96.7	17.0 (Reduction: 82.4%)	432.1	214.5 (Reduction: 50.4%)
64	198.0	555.0	189.7	34.3 (Reduction: 81.9%)	434.9	216.8 (Reduction: 50.2%)
128	354.0	984.0	343.6	56.3 (Reduction: 83.6%)	438.3	189.8 (Reduction: 56.7%)

results of the decryption one.

In Tables 5.3 and 5.4, the CPLD utilization of Blowfish encipher and decipher is presented in order to have a complete view of the resources needed for the implementation of this cipher on such a small reconfigurable device.

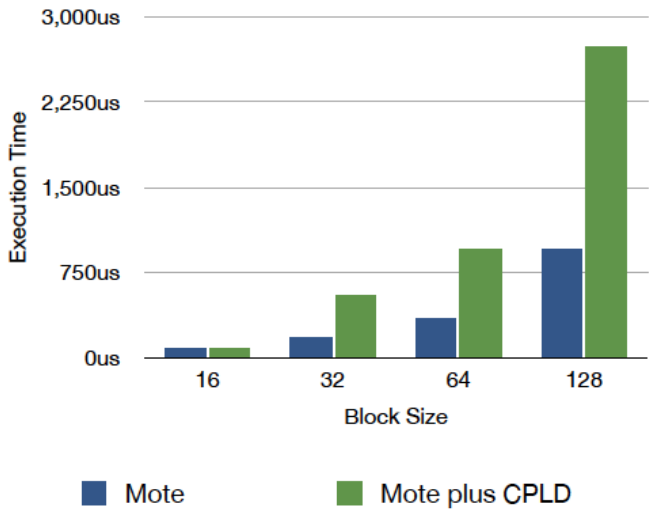

Figure 5.1: Blowfish encryption execution time results.

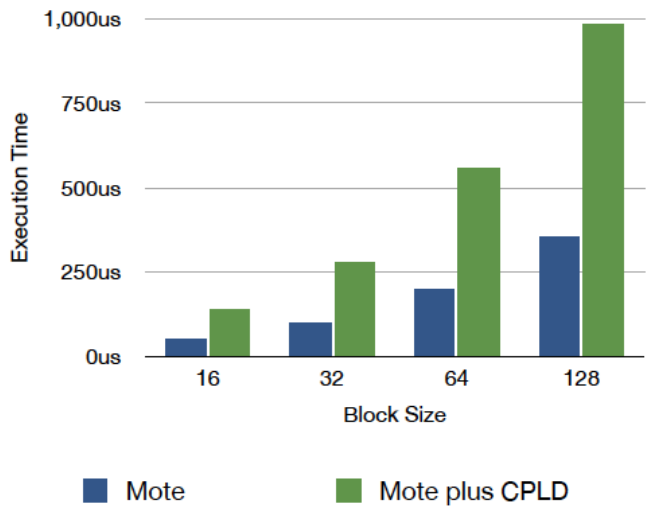

Figure 5.2: Blowfish decryption execution time results.

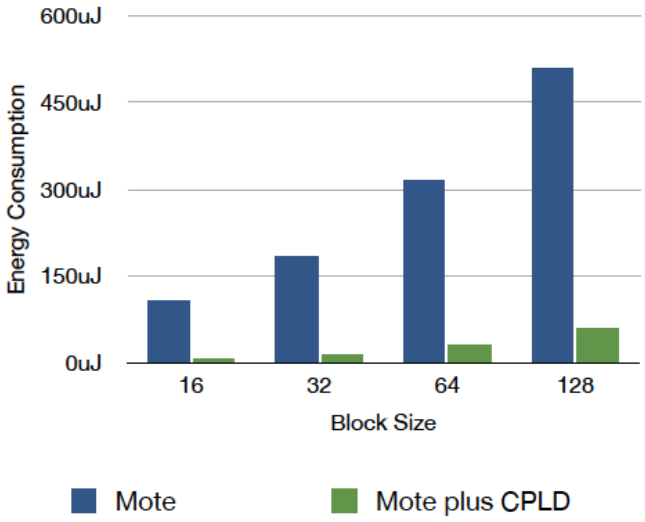

Figure 5.3: Blowfish encryption energy consumption results.

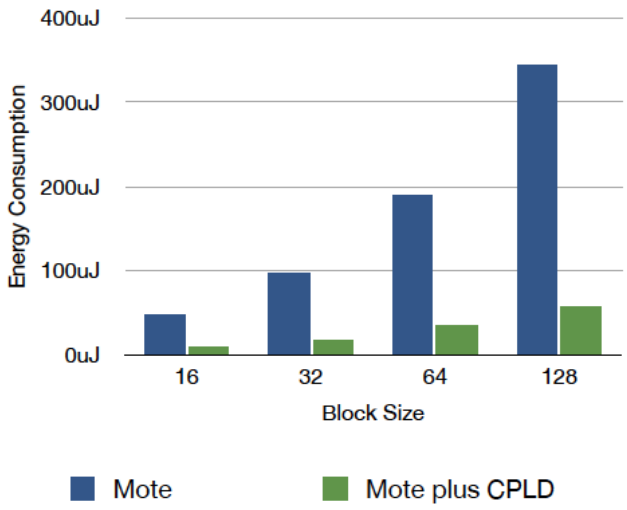

Figure 5.4: Blowfish decryption energy consumption results.

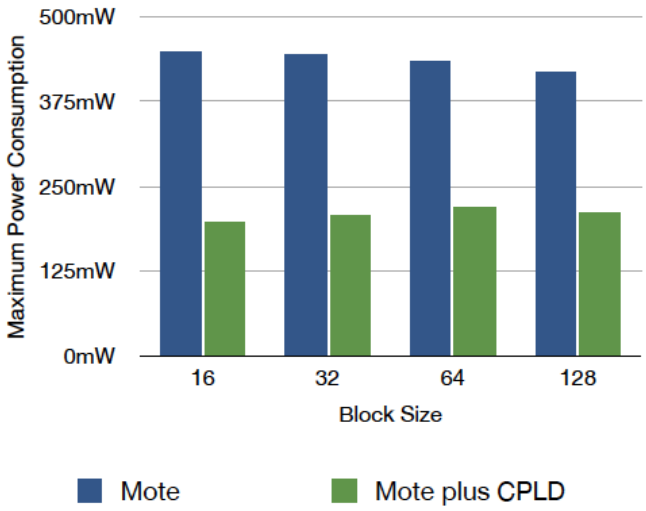

Figure 5.5: Blowfish encryption maximum power consumption results.

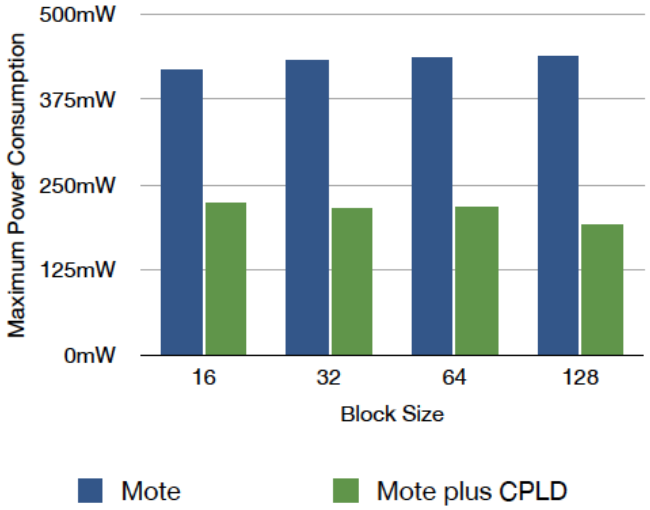

Figure 5.6: Blowfish decryption maximum power consumption results.

Table 5.3: Blowfish encipher CPLD utilization results

Macrocells Used	118/256 (47%)
Pterms Used	613/896 (69%)
Registers Used	71/256 (28%)
Pins Used	28/118 (24%)
Function Block Inputs Used	271/640 (43%)

Table 5.4: Blowfish decipher CPLD utilization results

Macrocells Used	113/256 (45%)
Pterms Used	621/896 (70%)
Registers Used	71/256 (28%)
Pins Used	28/118 (24%)
Function Block Inputs Used	251/640 (40%)

5.2 Base station results

Referring to our base station platform, a critical issue for its implementations is the values of the system inputs (plaintext and cipher key) that must be selected in order to lead us to correct conclusions evaluating our platform. The different values that a 128-bit quantity such plaintext and cipher key can take are 2^{128}. So it is impossible to measure the performance of all these different combinations. Consequently, the selection of plaintexts and cipher keys is of a great importance and the specific values must chosen in such a way that can give safe results about the power consumption of all our implementations.
Taking into account all the above, we conclude to try 6 different groups of plaintext values which are chosen to contain different number of *ones*. Some of these groups contain values with either no or being full of *ones*. In addition, some other groups consist of small or intermediate number of *ones*. It could be also pointed out that each group has values with only minor changes which are selected in order to evaluate the performance of AES designs in such changes. Especially, the plaintext values in a group differ only in their 8 LSBs. The different groups of plaintext values in hexadecimal radix are the following ones:

- 0x000000000000000000000000000000XX,

- 0x01010101010101010101010101010101XX,
- 0x10101010101010101010101010101010XX,
- 0x11111111111111111111111111111111XX,
- 0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAXX and
- 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXX,

where XX is a 8-bit counter that inserts the minor changes previously referred. Moreover, every group consists of 100 different plaintext values.
Regarding the cipher key, its different values are six and they are selected with a similar way to that of the plaintext ones. These values are chosen as follows:

- 0x00000000000000000000000000000000
- 0x01010101010101010101010101010101
- 0x10101010101010101010101010101010
- 0x11111111111111111111111111111111
- 0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
- 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Combining these values, 36 different groups containing 100 combinations of plaintexts and cipher keys each one are derived while these 3600 different experimental power measurements that are a representative sample can lead us to safe conclusions about the data-independence of our AES implementations.
The last issue for this work is the way the results are going to be presented. The oscilloscope prints the different graphs in a removable disk as a .csv file. So every of these files represents the measured values of voltage over the reference resistance during the encryption procedure of one experimental measurement. All these files are collected and processed by a MATLAB script in order to calculate the power values and, then, some statistical metrics of every group. The first statistical metric which is calculated is the mean value for both the maximum and average power consumption of each group. This metric give us a general view about the real value of the power consumed. The mean value μ is calculated as the arithmetic mean of the measured values and is given by the following formula:

$$\mu = \frac{1}{N} \cdot \Sigma_{i=1}^{N} x_i = \frac{1}{N} \cdot (x_1 + x_2 + ... + x_N)$$

Another statistical metric that is useful for showing how spread out the measured values are is the deviation. Deviation σ is defined as the square root of the variance σ^2 which is equal to the mean value of the square of difference between the measured values and the mean one. This metric is calculated as follows:

$$\sigma = \sqrt{\sigma^2}, \text{ where}$$
$$\sigma^2 = \frac{1}{N} \cdot \Sigma_{i=1}^{N} (x_i - \mu)^2 = \frac{1}{N} \cdot ((x_1 - \mu)^2 + (x_2 - \mu)^2 + ... + (x_N - \mu)^2)$$

The deviation is a very useful metric in order to show how much the measured values differ from the mean one in single group of combinations, but it represents only the real value of these difference. It is important to have an metric that is going to be used for illustrating the relative value of this difference compared to the mean one and this statistical metric is the coefficient of variation δ which is defined as the ratio of the deviation over the mean value. The equation used for calculating this metric is the following one:

$$\delta = \frac{\sigma}{\mu}$$

The first implementation that is going to be evaluated is the simple single-rail one the results of which are of great importance, because they will show if this work has a research interest. In other words, these results can lead us to the conclusion if a simple FPGA-based implementation of the AES algorithm consume data-dependent power. The results that were derived from this study have further statistically processed and are presented in Tables 5.5 and 5.6 where plainXY is the group of 100 plaintexts, each byte of which is equal to XY (e.g., 00, 01, 10, 11, AA and FF), apart from their last one which is equal to the current value of the counter, and keyWZ is that hexadecimal value of cipher key being equal to WZWZ...WZ. These tables consists of all the statistic metrics referred above of both the maximum and average power consumption for each group of plaintext and cipher key combinations.

As shown in these tables, the maximum power consumption of this AES implementation varies 19.68% on average in the total test case and, regarding its average power consumption, the coefficient of variation is more than 30% on average (see Table 5.15). So the above values of coefficient of variation lead us to the conclusion

Table 5.5: AES single-rail results

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key00	plain00	0.0712	0.0156	0.2183	0.0273	0.0100	0.3685
	plain01	0.1050	0.0107	0.1022	0.0550	0.0069	0.1257
	plain10	0.1154	0.0104	0.0904	0.0655	0.0058	0.0886
	plain11	0.0880	0.0089	0.1006	0.0408	0.0045	0.1098
	plainAA	0.0727	0.0131	0.1800	0.0312	0.0083	0.2667
	plainFF	0.1061	0.0127	0.1201	0.0588	0.0084	0.1431
key01	plain00	0.0615	0.0096	0.1554	0.0193	0.0026	0.1329
	plain01	0.1067	0.0094	0.0877	0.0614	0.0060	0.0981
	plain10	0.1212	0.0096	0.0796	0.0707	0.0049	0.0691
	plain11	0.0775	0.0197	0.2537	0.0327	0.0154	0.4701
	plainAA	0.0769	0.0113	0.1472	0.0345	0.0057	0.1643
	plainFF	0.1054	0.0123	0.1166	0.0613	0.0086	0.1407
key10	plain00	0.0643	0.0093	0.1438	0.0214	0.0033	0.1534
	plain01	0.1162	0.0115	0.0991	0.0695	0.0074	0.1071
	plain10	0.1289	0.0131	0.1013	0.0768	0.0088	0.1149
	plain11	0.0747	0.0127	0.1702	0.0300	0.0072	0.2404
	plainAA	0.0862	0.0125	0.1445	0.0451	0.0072	0.1600
	plainFF	0.0965	0.0109	0.1130	0.0508	0.0054	0.1056

that the power consumption is data-dependent which makes this implementation vulnerable to the differential power analysis and that must be improved. Studying more carefully the results of every group, this conclusion is strengthen, because maximum power consumption coefficient of variation ranges from 5.42% to 25.37% and average power consumption one from 4.11% to 47.01% according to the input values of system. Another issue that can not be presented in these tables is that the graph of power consumption during the encryption process differs from one input to another. The maximum value of the power consumption is performed in different time instances which can lead to many vulnerabilities, because if an attacker studies this performance of our system, he can retrieve information that he has not right to

Table 5.6: AES single-rail results - continue...

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key11	plain00	0.0895	0.0093	0.1043	0.0482	0.0047	0.0971
	plain01	0.1188	0.0079	0.0666	0.0705	0.0029	0.0411
	plain10	0.1367	0.0074	0.0542	0.0828	0.0031	0.0376
	plain11	0.0718	0.0123	0.1715	0.0268	0.0046	0.1722
	plainAA	0.0980	0.0104	0.1065	0.0552	0.0080	0.1440
	plainFF	0.0930	0.0180	0.1932	0.0460	0.0140	0.3047
keyAA	plain00	0.0801	0.0162	0.2018	0.0405	0.0129	0.3182
	plain01	0.1143	0.0107	0.0937	0.0605	0.0056	0.0926
	plain10	0.0960	0.0102	0.1061	0.0456	0.0077	0.1696
	plain11	0.0812	0.0101	0.1238	0.0365	0.0057	0.1559
	plainAA	0.0939	0.0129	0.1370	0.0503	0.0071	0.1406
	plainFF	0.1100	0.0133	0.1207	0.0653	0.0117	0.1784
keyFF	plain00	0.0849	0.0151	0.1778	0.0414	0.0105	0.2544
	plain01	0.1236	0.0103	0.0832	0.0696	0.0049	0.0706
	plain10	0.0942	0.0159	0.1689	0.0461	0.0139	0.3005
	plain11	0.0790	0.0114	0.1444	0.0350	0.0070	0.2005
	plainAA	0.0947	0.0121	0.1279	0.0505	0.0081	0.1597
	plainFF	0.1108	0.0106	0.0957	0.0633	0.0087	0.1372

access.

As previously referred, these results have to be improved in order to create a well-protected approach for implementing AES algorithm. The first idea for smoothing the power consumption of this cryptographic system is the use of the dual-rail method. Unfortunately, a alternating spacer dual-rail implementation have not been feasible to be created due to the limited resources of the given FPGA device. In contrast, we implemented a single spacer one, the deriving results of which are presented in Tables 5.7 and 5.8.

Regardless the results of this implementation are encouraging (see Table 5.15), they are not close to the desirable ones and must be further improved. If an alternating

Table 5.7: AES dual-rail results

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key00	plain00	0.1142	0.0091	0.0800	0.0671	0.0073	0.1092
	plain01	0.1189	0.0085	0.0713	0.0659	0.0047	0.0719
	plain10	0.0967	0.0137	0.1417	0.0504	0.0094	0.1867
	plain11	0.0824	0.0152	0.1841	0.0418	0.0098	0.2336
	plainAA	0.1117	0.0138	0.1236	0.0632	0.0094	0.1492
	plainFF	0.0981	0.0068	0.0692	0.0474	0.0033	0.0687
key01	plain00	0.0822	0.0158	0.1918	0.0372	0.0105	0.2835
	plain01	0.1096	0.0149	0.1357	0.0565	0.0115	0.2041
	plain10	0.0901	0.0121	0.1340	0.0460	0.0065	0.1417
	plain11	0.0827	0.0144	0.1735	0.0404	0.0094	0.2333
	plainAA	0.1151	0.0108	0.0936	0.0664	0.0084	0.1260
	plainFF	0.1095	0.0090	0.0820	0.0577	0.0036	0.0624
key10	plain00	0.0617	0.0125	0.2022	0.0224	0.0073	0.3274
	plain01	0.1160	0.0099	0.0851	0.0596	0.0061	0.1031
	plain10	0.0865	0.0112	0.1295	0.0448	0.0069	0.1534
	plain11	0.0840	0.0152	0.1812	0.0423	0.0100	0.2371
	plainAA	0.0925	0.0112	0.1213	0.0498	0.0070	0.1415
	plainFF	0.1071	0.0112	0.1044	0.0568	0.0060	0.1063

spacer implementation was feasible, the results would be better. The single spacer DR one performs coefficient of variation equal to 14.58% and 21.24% on average for the maximum and average power consumption (see Table 5.15) that means a total decrease of 25% and 30%, respectively, but it is not enough for supporting that this system is well-protected from differential power analysis. Furthermore, its maximum power consumption coefficient of variation ranges from 6.23% to 24.93% while its average one from 4.15% to 42.28% and the problem with the different distribution of the power consumption during the calculation is already exists.

For all the above reasons and taking into account the infeasibility of a dual-rail implementation of two spacers, we conclude that a new method must be found for

Table 5.8: AES dual-rail results - continue...

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key11	plain00	0.0752	0.0187	0.2493	0.0319	0.0135	0.4228
	plain01	0.1133	0.0117	0.1031	0.0579	0.0065	0.1129
	plain10	0.0846	0.0105	0.1240	0.0421	0.0079	0.1890
	plain11	0.0790	0.0148	0.1878	0.0386	0.0110	0.2838
	plainAA	0.1005	0.0173	0.1719	0.0549	0.0119	0.2174
	plainFF	0.0929	0.0106	0.1144	0.0481	0.0060	0.1254
keyAA	plain00	0.0791	0.0149	0.1885	0.0362	0.0103	0.2840
	plain01	0.0998	0.0101	0.1009	0.0474	0.0046	0.0980
	plain10	0.0931	0.0142	0.1528	0.0492	0.0096	0.1949
	plain11	0.1049	0.0185	0.1767	0.0586	0.0120	0.2053
	plainAA	0.1010	0.0108	0.1070	0.0552	0.0048	0.0868
	plainFF	0.1137	0.0071	0.0623	0.0638	0.0027	0.0415
keyFF	plain00	0.0844	0.0158	0.1871	0.0406	0.0123	0.3034
	plain01	0.1091	0.0102	0.0933	0.0566	0.0063	0.1111
	plain10	0.0908	0.0099	0.1090	0.0478	0.0062	0.1302
	plain11	0.1028	0.0148	0.1438	0.0559	0.0107	0.1920
	plainAA	0.0781	0.0099	0.1273	0.0343	0.0057	0.1649
	plainFF	0.1072	0.0081	0.0753	0.0599	0.0040	0.0665

the improvement of the power consumption results. If we study the results from the previous design more carefully, we can see that the dual-rail method performs well in test cases where the combination plaintext and cipher key contains no *ones* or a large number of them. On the contrary, test cases where these combinations consists of a relatively small number of *ones* are the main drawback of this implementation. The last observation leads us to the solution of using the "duplication" method where another implementation of the AES encryption is placed to run in parallel with the initial one. The inputs of this additional module are the bitwise inverses of the initial ones, because of the observation previously referred. Consequently, when the initial module gets inputs with a small number of *ones*, the inputs of the

Table 5.9: AES 2-round duplicate DR results

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key00	plain00	0.1188	0.0025	0.0208	0.0757	0.0014	0.0182
	plain01	0.1050	0.0052	0.0491	0.0740	0.0040	0.0542
	plain10	0.1110	0.0036	0.0323	0.0790	0.0021	0.0266
	plain11	0.1249	0.0029	0.0234	0.0908	0.0011	0.0121
	plainAA	0.1281	0.0032	0.0248	0.0928	0.0014	0.0150
	plainFF	0.0789	0.0027	0.0337	0.0534	0.0014	0.0257
key01	plain00	0.0920	0.0016	0.0178	0.0705	0.0012	0.0170
	plain01	0.1060	0.0043	0.0406	0.0754	0.0032	0.0430
	plain10	0.1225	0.0025	0.0206	0.0882	0.0008	0.0086
	plain11	0.1242	0.0028	0.0222	0.0902	0.0014	0.0151
	plainAA	0.1257	0.0033	0.0262	0.0912	0.0016	0.0179
	plainFF	0.0708	0.0032	0.0458	0.0469	0.0020	0.0417
key10	plain00	0.1044	0.0034	0.0322	0.0842	0.0025	0.0302
	plain01	0.1044	0.0044	0.0421	0.0735	0.0031	0.0419
	plain10	0.1241	0.0026	0.0213	0.0901	0.0010	0.0116
	plain11	0.1215	0.0035	0.0285	0.0876	0.0020	0.0228
	plainAA	0.1249	0.0031	0.0249	0.0905	0.0014	0.0160
	plainFF	0.0764	0.0024	0.0312	0.0521	0.0007	0.0141

additional module contain a large number of them and vice versa, which are going to help us to smooth the power consumption during the calculation. As described in the previous chapter, three different designs containing an additional module with different number of cipher rounds have derived from this method. The results of the first one, the 2-round duplicate DR system, are illustrated in Tables 5.9 and 5.10.

The results shown in these tables are quite impressive, because there is a significant reduction in the coefficient of variation of maximum and average power consumption being equal to 75% and 83% compared to the initial design, respectively. Especially, the maximum power consumption coefficient of variation is equal to 5.47% on average (see Table 5.15) and ranges from 1.78% to 6.25%. Regarding the average power

Table 5.10: AES 2-round duplicate DR results - continue...

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key11	plain00	0.1057	0.0033	0.0315	0.0749	0.0015	0.0199
	plain01	0.1021	0.0048	0.0467	0.0708	0.0026	0.0369
	plain10	0.1247	0.0030	0.0240	0.0901	0.0013	0.0143
	plain11	0.1225	0.0030	0.0245	0.0887	0.0014	0.0156
	plainAA	0.1259	0.0035	0.0279	0.0910	0.0013	0.0143
	plainFF	0.0777	0.0028	0.0355	0.0528	0.0014	0.0263
keyAA	plain00	0.1057	0.0039	0.0373	0.0746	0.0024	0.0323
	plain01	0.1007	0.0043	0.0432	0.0699	0.0024	0.0343
	plain10	0.1250	0.0029	0.0234	0.0907	0.0014	0.0155
	plain11	0.1248	0.0031	0.0249	0.0902	0.0012	0.0136
	plainAA	0.1244	0.0030	0.0243	0.0901	0.0009	0.0102
	plainFF	0.0794	0.0023	0.0284	0.0544	0.0011	0.0200
keyFF	plain00	0.1088	0.0035	0.0321	0.0773	0.0016	0.0206
	plain01	0.1002	0.0063	0.0625	0.0694	0.0043	0.0614
	plain10	0.1248	0.0032	0.0253	0.0906	0.0013	0.0144
	plain11	0.1253	0.0033	0.0263	0.0911	0.0013	0.0145
	plainAA	0.1249	0.0028	0.0227	0.0908	0.0012	0.0133
	plainFF	0.0825	0.0022	0.0263	0.0568	0.0007	0.0129

consumption of 2-round duplicate DR implementation, it presents coefficient of variation equal to 4.84% on average (see Table 5.15) ranging from 0.86% to 6.14%. In addition, the main advantage of this new design is that the power it consumes has a fixed distribution and, consequently, is independent from the data input inserted to the AES module. Observing the above result tables, we can conclude that, despite the fact that this new system performs well in minor changes of inputs (moving inside a group of plaintexts that differ only in their least significant bytes), it has a problem with larger ones (selecting input from different groups with very different values of plaintexts and cipher keys) and there is able to be further improved.

For improving the power consumption coefficient of variation for such large input

Table 5.11: AES 4-round duplicate DR results

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key00	plain00	0.1029	0.0037	0.0362	0.0852	0.0029	0.0337
	plain01	0.1142	0.0030	0.0261	0.0956	0.0020	0.0212
	plain10	0.1165	0.0029	0.0249	0.0989	0.0012	0.0125
	plain11	0.1063	0.0044	0.0416	0.0881	0.0038	0.0434
	plainAA	0.1125	0.0030	0.0268	0.0946	0.0012	0.0125
	plainFF	0.1148	0.0037	0.0319	0.0962	0.0019	0.0201
key01	plain00	0.1064	0.0031	0.0291	0.0891	0.0020	0.0224
	plain01	0.1222	0.0039	0.0320	0.1037	0.0020	0.0192
	plain10	0.1153	0.0047	0.0406	0.0973	0.0031	0.0319
	plain11	0.1099	0.0033	0.0305	0.0910	0.0014	0.0159
	plainAA	0.1110	0.0030	0.0269	0.0927	0.0009	0.0099
	plainFF	0.1145	0.0032	0.0280	0.0961	0.0018	0.0187
key10	plain00	0.1071	0.0031	0.0294	0.0895	0.0023	0.0256
	plain01	0.1175	0.0036	0.0308	0.0991	0.0024	0.0245
	plain10	0.1151	0.0042	0.0365	0.0977	0.0028	0.0288
	plain11	0.1128	0.0029	0.0256	0.0944	0.0012	0.0129
	plainAA	0.1114	0.0031	0.0282	0.0926	0.0013	0.0137
	plainFF	0.1141	0.0035	0.0309	0.0956	0.0019	0.0198

changes, it is thought that the number of cipher rounds in the additional module of AES must be increased. We decide to try to implement a design with an additional module containing four cipher rounds called 4-round duplicate DR, the results of which are illustrated in Tables 5.11 and 5.12.

Increasing the cipher rounds of the parallel module of AES leads us to a further improvement of the power consumption coefficient of variation which means a further decrease of its value. Maximum power consumption of 4-round duplicate DR implementation varies 3.47% on average (see Table 5.15) ranging from 2.43% to 5.66% while average power consumption of the same implementation varies 3.99% on average (see Table 5.15) ranging from 0.99% to 5.90%. The distribution of

Table 5.12: AES 4-round duplicate DR results - continue...

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key11	plain00	0.1069	0.0033	0.0305	0.0898	0.0021	0.0232
	plain01	0.1131	0.0064	0.0566	0.0945	0.0056	0.0590
	plain10	0.1128	0.0036	0.0318	0.0950	0.0019	0.0196
	plain11	0.1114	0.0033	0.0301	0.0929	0.0010	0.0107
	plainAA	0.1138	0.0039	0.0344	0.0954	0.0028	0.0294
	plainFF	0.1142	0.0030	0.0261	0.0955	0.0015	0.0161
keyAA	plain00	0.1110	0.0033	0.0301	0.0940	0.0016	0.0167
	plain01	0.1176	0.0051	0.0436	0.0993	0.0037	0.0375
	plain10	0.1114	0.0043	0.0386	0.0931	0.0031	0.0333
	plain11	0.1132	0.0029	0.0255	0.0955	0.0011	0.0112
	plainAA	0.1146	0.0039	0.0341	0.0957	0.0023	0.0241
	plainFF	0.1158	0.0028	0.0243	0.0979	0.0013	0.0128
keyFF	plain00	0.1061	0.0037	0.0349	0.0882	0.0023	0.0259
	plain01	0.1178	0.0036	0.0306	0.1000	0.0020	0.0198
	plain10	0.1133	0.0036	0.0315	0.0954	0.0025	0.0266
	plain11	0.1087	0.0029	0.0264	0.0903	0.0010	0.0116
	plainAA	0.1144	0.0048	0.0422	0.0957	0.0041	0.0427
	plainFF	0.1153	0.0029	0.0251	0.0961	0.0013	0.0137

power consumption still remains fixed and, consequently, data-independent during the whole encryption process.

In spite of the very good performance that the last implementation presents, we also try to implement another design with increased cipher rounds. This design consists of an additional module of eight cipher rounds and is called 8-round duplicate DR. The results of the 8-round duplicate DR implementation, which is the last one presented by this work, are presented in Tables 5.13 and 5.14.

8-round duplicate DR implementation is the one that presents the best results among the five implementations described by this work. This implementation leads to an important decrease of the power consumption coefficient of variation. Espe-

Table 5.13: AES 8-round duplicate DR results

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key00	plain00	0.1488	0.0012	0.0108	0.0910	0.0007	0.0110
	plain01	0.1402	0.0036	0.0258	0.0924	0.0021	0.0225
	plain10	0.1411	0.0040	0.0286	0.0916	0.0020	0.0213
	plain11	0.1437	0.0042	0.0292	0.0936	0.0021	0.0222
	plainAA	0.1347	0.0012	0.0092	0.0848	0.0007	0.0081
	plainFF	0.1420	0.0011	0.0077	0.0891	0.0019	0.0215
key01	plain00	0.1457	0.0009	0.0099	0.0991	0.0005	0.0110
	plain01	0.1423	0.0036	0.0253	0.0953	0.0030	0.0319
	plain10	0.1400	0.0042	0.0298	0.0897	0.0025	0.0281
	plain11	0.1446	0.0050	0.0349	0.0932	0.0022	0.0233
	plainAA	0.1493	0.0048	0.0323	0.0949	0.0034	0.0362
	plainFF	0.1394	0.0047	0.0340	0.0898	0.0020	0.0228
key10	plain00	0.1474	0.0043	0.0293	0.0945	0.0025	0.0266
	plain01	0.1292	0.0009	0.0071	0.0828	0.0007	0.0086
	plain10	0.1424	0.0044	0.0308	0.0929	0.0019	0.0201
	plain11	0.1431	0.0011	0.0076	0.0917	0.0023	0.0255
	plainAA	0.1511	0.0049	0.0322	0.0953	0.0022	0.0229
	plainFF	0.1433	0.0049	0.0345	0.0918	0.0023	0.0254

cially, the coefficient of variation of maximum and average power consumption is equal to 1.73% and 1.64% (see Table 5.15), respectively, while their ranges are from 0.70% to 3.45% for the first metric and from 0.61% to 3.76% for the second one. The last figures show a significant decrease more than 90% for both variations compared to the initial design and also make us to conclude that the power consumption of the final design is data-independent due to the low value of these variations among the different inputs tested and taking into account that its distribution during the calculation of the ciphertext still remains fixed. Besides the value of the coefficient of variation of both maximum and average power consumption is not equal to zero, it is very close to it which makes us to support that it may be caused by measure-

Table 5.14: AES 8-round duplicate DR results - continue...

		Maximum Power Consumption			Average Power Consumption		
		Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
key11	plain00	0.1501	0.0040	0.0266	0.0966	0.0033	0.0344
	plain01	0.1407	0.0045	0.0322	0.0901	0.0006	0.0070
	plain10	0.1421	0.0038	0.0267	0.0914	0.0032	0.0348
	plain11	0.1464	0.0047	0.0324	0.0951	0.0036	0.0376
	plainAA	0.1480	0.0011	0.0073	0.0926	0.0022	0.0242
	plainFF	0.1457	0.0010	0.0070	0.0936	0.0030	0.0320
keyAA	plain00	0.1507	0.0038	0.0255	0.0981	0.0019	0.0191
	plain01	0.1413	0.0038	0.0269	0.0918	0.0020	0.0214
	plain10	0.1444	0.0040	0.0274	0.0944	0.0028	0.0300
	plain11	0.0138	0.0010	0.0071	0.0868	0.0025	0.0290
	plainAA	0.1528	0.0046	0.0303	0.0979	0.0006	0.0061
	plainFF	0.1405	0.0047	0.0335	0.0907	0.0006	0.0069
keyFF	plain00	0.1423	0.0044	0.0308	0.0938	0.0030	0.0316
	plain01	0.1427	0.0047	0.0328	0.0922	0.0019	0.0202
	plain10	0.1455	0.0041	0.0283	0.0957	0.0033	0.0342
	plain11	0.1325	0.0011	0.0084	0.0811	0.0005	0.0060
	plainAA	0.1418	0.0011	0.0081	0.0881	0.0008	0.0089
	plainFF	0.1410	0.0043	0.0304	0.0914	0.0006	0.0063

ments faults.

The statistics of the total test case including 3600 samples are referred in the previous paragraphs, but they are not illustrated in the specific tables, because it is thought preferable to collect and show them in Table 5.15 in order to be compared. Observing this table, the great decrease between the initial design (single-rail) and the final one (8-round duplicate DR) referring to the data-independence of their power consumption is presented. In addition, this table illustrates the cost of improving the data-independence of the AES implementation power consumption which is the increase of its real value. Regarding the initial design, the maximum and average power consumption are equal to 95.7 and 49.6 mW, respectively, while

Table 5.15: AES total power results

	Maximum Power Consumption			**Average Power Consumption**		
	Mean value	**Deviation**	**Coefficient of variation**	**Mean value**	**Deviation**	**Coefficient of variation**
Single-rail	0.0957	0.0188	0.1968	0.0496	0.0161	0.3255
Dual-rail	0.0964	0.0140	0.1458	0.0498	0.0106	0.2124
2-round Duplicate DR	0.1097	0.0060	0.0547 (Reduction: 62.5%)	0.0783	0.0038	0.0484 (Reduction: 77.2%)
4-round Duplicate DR	0.1127	0.0039	0.0347 (Reduction: 76.2%)	0.0945	0.0038	0.0399 (Reduction: 81.2%)
8-round Duplicate DR	0.1418	0.0024	0.0173 (Reduction: 88.1%)	0.0920	0.0015	0.0164 (Reduction: 92.3%)

these metrics of the final design are equal to 141.8 and 92.0 mW, which means that the last implementation presents a 50% increase in maximum and 80% in average power consumption compared to the first one. In Table 5.15, the decrease values of coefficient of variation are shown in order to evaluate more precisely the results derived from the latter implementation of AES. Especially, this comparison is carried out between the three latter implementations (the "duplicate" implementations) and the dual-rail one, because the decrease of this metric performed between the single-rail and the dual-rail implementation is due to the large decrease of the system clock frequency, as it will be presented later in this chapter. So we decided that it would be more proper to compare systems with similar clock frequency.

Except for the power results previously presented, it is important to show the timing ones. The significant advantage of data-independent power consumption that means that our system is well-protected from differential power analysis must be followed by a high value of throughput in order to present an efficient system offering high security and performance. For this purpose, Table 5.16 illustrates the deriving timing results of the five designs described by this work. These results derive from the Xilinx ISE place-and-route tool.

The pipeline and the high level of parallelism that is used by our AES implementations enable us to have such high values of throughput. Regarding the single-rail AES implementation, cipher rounds unrolling leads us to a throughput value more than 13 Gbps which is one of the highest ones achieved by an FPGA implementa-

Table 5.16: AES implementations timing results

	Clock period (ns)	**Clock Frequency (MHz)**	**Throughput (Gbps)**
Single-rail	9.798	102.062	13.064
Dual-rail	16.647	60.071	7.689
2-round Duplicate DR	18.254	54.783	7.012
4-round Duplicate DR	17.568	56.922	7.286
8-round Duplicate DR	19.774	50.572	6.473

tion since now. Especially, only Hodjat et al. implementation in [89] achieves higher throughput that is equal to 21.54 Gbps, as a result of their exclusive study based on this metric. Although the throughput is not the main goal of our work, the value achieved is very encouraging for further research in the future. Moving to the other implementations, the throughput is dropping down due to the higher complexity of these circuits. The value achieved by our final design (8-round duplicate DR implementation) that is the most secure one according to its power independence is 50% smaller than the one of the initial design being equal to 6.5 Gbps. This drop in the throughput value is another cost for achieving high security and creating a well-protected system from differential power analysis. Despite this notable decrease, the throughput value still remains high. These values of throughput are shown in Figure 5.13 in a schematic way.

Another important issue for evaluating the performance of our platform is to compare it with the results derived when the whole encryption process is executed by a software running on a low-power CPU such Intel Atom. The metrics in terms of which the comparison is going to be made are execution time, energy and power consumption.

In order to measure the execution time of the AES software on the Atom Processor, we used the Intel Vtune Performance analyzer tool 9.1 [90] installed on Linux Xubuntu 8.10, whereas for the power draw measurement, we used the Linux PowerTop tool [91]. PowerTop is a tool that measures the percentage of power states

Table 5.17: AES software vs. hardware

	Execution time (us)	**Energy consumption (uJ)**	**Power consumption (W)**
Software	215.57	431.13	2.00
Single-rail	2.92 (Speedup: 73.8x)	0.15 (Reduction: 99.9%)	0.05 (Reduction: 97.5%)
Dual-rail	4.96 (Speedup: 43.5x)	0.25 (Reduction: 99.9%)	0.05 (Reduction: 97.5%)
2-round Duplicate DR	5.44 (Speedup: 39.6x)	0.43 (Reduction: 99.9%)	0.08 (Reduction: 96.1%)
4-round Duplicate DR	5.24 (Speedup: 41.1x)	0.50 (Reduction: 99.9%)	0.10 (Reduction: 95.3%)
8-round Duplicate DR	5.89 (Speedup: 36.6x)	0.54 (Reduction: 99.9%)	0.09 (Reduction: 95.4%)

at which a processor exists, when executing a software. We could not use the previously described methodology, in order to measure the execution time, energy and power draw of the Intel Atom board, since we could not isolate the voltage regulator, which drives the current to the processor. This is the only technique that is used in bibliography, in order to measure the power consumption of this kind of boards.
Regarding the power states of the Atom processor, these are described in detail in [92] datasheet. To summarize, the Atom 330 Family processors have 2 states: C0 and C1. The C0 state is the normal operating state for threads in the processor while C1 is a low-power state entered when a thread executes a halt or wait instructions. When the Atom processor enters in C0 state, the average power draw is 8 W, whereas when it enters in C1 state the average power draw is 2 W. These are the only experimental values the manufacturer provides, without giving details for this specific experimental topology.
In order our measurements on the FPGA to keep up with the measurements on the Atom processor, the same experiments were carried out; the same number of plaintext and cipher key combinations were inserted as inputs to the two systems.
The results of the above measurements were collected and illustrated in Table 5.17. As shown in this table, the proposed platform executes the encryption procedure from 36.6 to 73.8 times faster than the software does while there is a subsequent

Table 5.18: AES FPGA utilization results

	No of Slices (out of 13696)	**No of 4-input LUTs (out of 27392)**	**No of Slice Flip Flops (out of 27392)**	**No of RAMB16s (out of 136)**
Single-rail	2617 (19%)	4618 (16%)	492 (1%)	101 (74%)
Dual-rail	4848 (35%)	8932 (32%)	493 (1%)	101 (74%)
2-round Duplicate DR	6469 (47%)	12013 (43%)	511 (1%)	129 (94%)
4-round Duplicate DR	7936 (57%)	14996 (54%)	655 (2%)	136 (100%)
8-round Duplicate DR	13341 (97%)	25604 (93%)	1167 (4%)	136 (100%)

reduction in energy and power consumption more than 95%.
Finally, in order to have a complete view about the performance and the efficiency of our method, it is thought indispensable to show the figures of FPGA utilization. These figures are illustrated in Table 5.18.
Our initial design gets only a few of the available resources of the given FPGA device. Especially, the ratio of logic slices utilization is very low and equal to 19%, in contrast with the BRAMs utilization ratio which is higher and equal to 74%. Moving to the remaining implementations, a higher number of the available resources is utilized. Finally, the last system (8-round duplicate DR) uses the total of the FPGA available resources (97% of slices and 100% of BRAMs).

Figures 5.7 to 5.13 show the above results in a schematic way. Especially, Figures 5.7 and 5.8 present the comparison among the AES implementations based on the coefficient of variation of maximum power consumption while Figures 5.9 and 5.10 depict the same comparison according to the average power consumption. Each group in these four figures contains five columns representing the five different implementations of AES and depicts the coefficient of variation of every single plaintext and cipher key combinations set and, consequently, the internal variation among the 100 combinations of this specific set. Moreover, Figures 5.11 and 5.12 illustrate this comparison for the total results of maximum and average power consumption, respectively. Finally, Figure 5.13 shows the different values of throughput achieved by the five AES implementations.

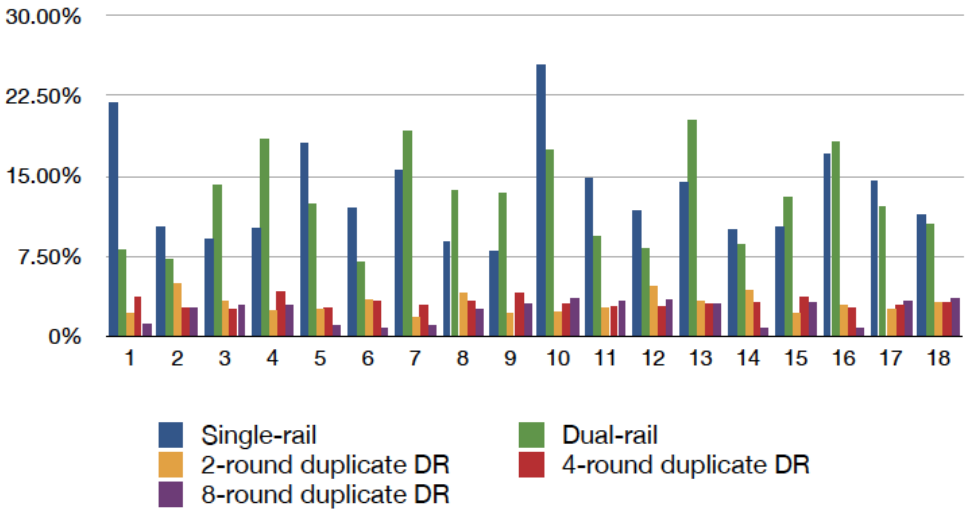

Figure 5.7: AES maximum power consumption results.

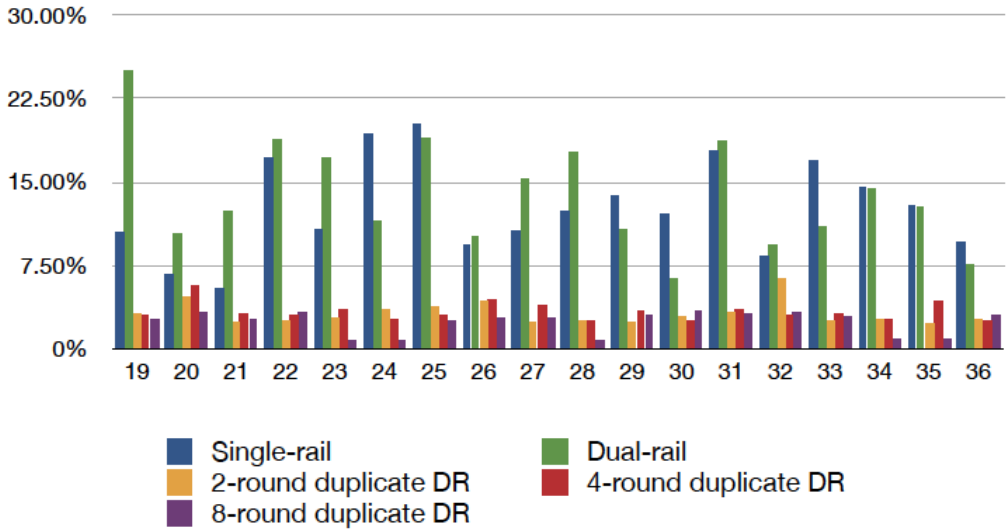

Figure 5.8: AES maximum power consumption results - continue...

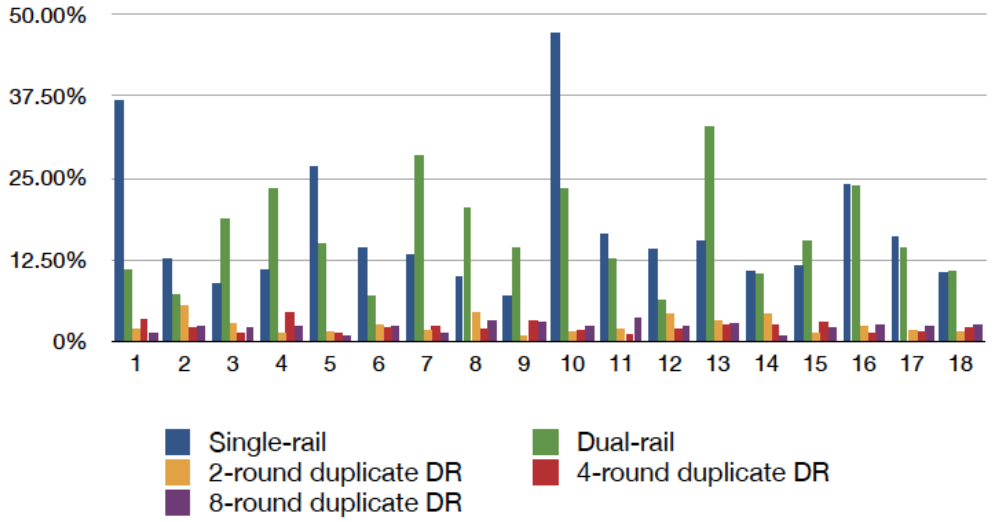

Figure 5.9: AES average power consumption results.

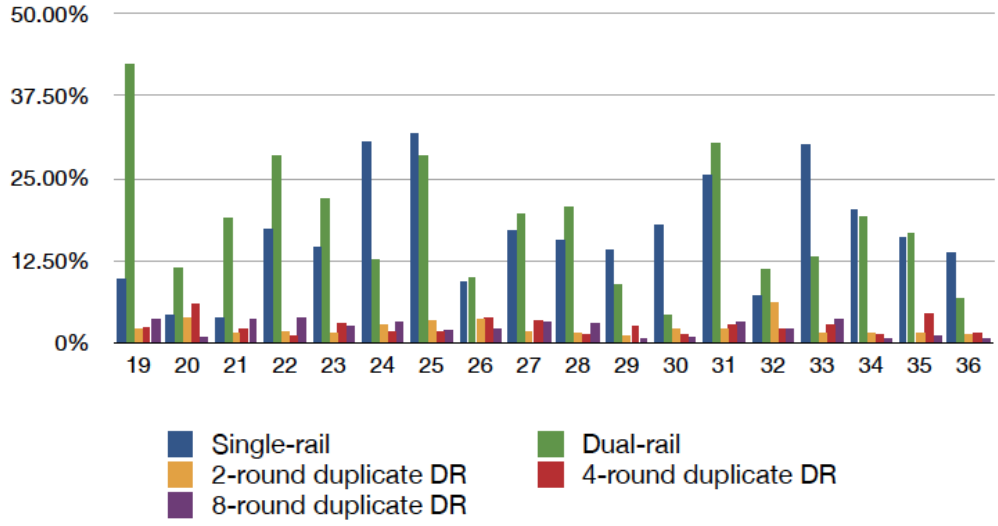

Figure 5.10: AES average power consumption results - continue...

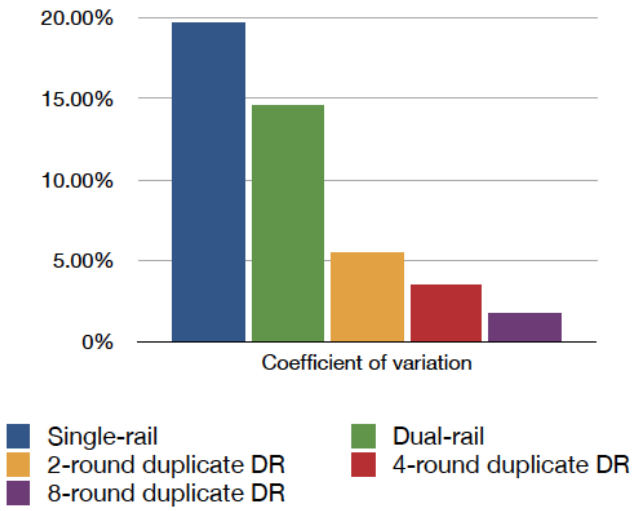

Figure 5.11: AES maximum power consumption total results.

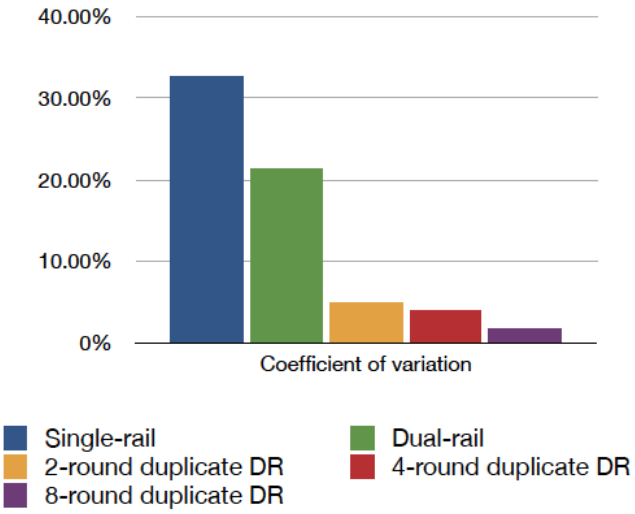

Figure 5.12: AES average power consumption total results.

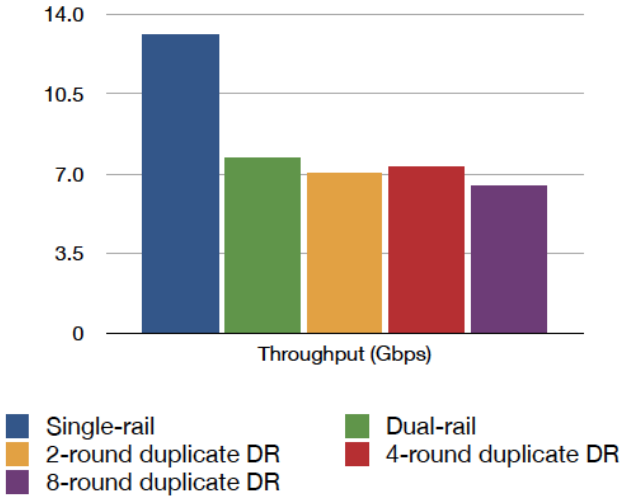

Figure 5.13: AES throughput results.

Chapter 6

Conclusions

This work introduces a completely new framework for creating wireless platforms using reconfigurable devices. Different kinds of wireless platforms (sensor nodes and base stations) are expanded using these specific devices in order to take benefit of them. Especially, a given wireless mote is expanded by connecting it with a CPLD for creating our sensor node platform while an FPGA is connected to a desktop board used in order to produce our new base station platform.
Regarding sensor node, our CPLD-based platform for wireless sensor networks provides higher level of security than the existing solutions at a significantly lower energy cost. This approach triggers a reduction in energy and power consumption equal to 90% and 50%, respectively, when executing certain encryption algorithms. Furthermore, different cipher encryption schemes are implemented on sensor node CPLD to offer the expected security level and the one that is selected for being the default of our platform is Blowfish-16. Its small block size makes this version of Blowfish perfectly suitable for wireless sensor network platforms and, moreover, taking into account the limited resources of a CPLD, Blowfish-16 seems a good candidate for increasing the security supported by the small WSN nodes. Since the previously described hardware implementation of the Blowfish cipher encryption supports only 16-bit block, we developed a software implementation of Cipher Block Chaining (CBC) in order to be able to efficiently encrypt more data bytes.
According to our base station platform, we develop an FPGA-based architecture for the Advanced Encryption Standard, which is a stronger cipher and is used for providing higher level of security compared to the sensor node. The main reason behind this decision is that base stations are more common targets of the attackers, because they collect data from all the other nodes and contain important information about the structure and the operation of the wireless network. Another issue is that platforms including such cryptographic schemes (i.e., AES) are vulnerable to *side-channel analysis*. The power that AES consumes varies according to the data input of the algorithm. Consequently, an attacker can retrieve information

about the network by performing *differential power analysis*. This drawback drove us to the decision of implementing an AES system with data-independent power consumption and, subsequently, a system that is well-protected from *side-channel analysis*. For this purpose, six different hardware implementations of AES are proposed for reaching the desirable results. Besides the coefficient of variation of our final implementation (8-round duplicate dual-rail) is not equal to zero, the results deriving from this system are very close to the ideal ones and the variation performed in this design can come from measurements faults. On the other hand, there are also some disadvantages in this method. Its main drawback is the increase in the value of power consumption which reaches up to 100% compared to the initial design.

Based on our real-world measurements, we believe that this work demonstrates a very promising approach for building highly secure WSN nodes that will incorporate low-power and high-performance reconfigurable devices.

Chapter 7

Future work

Regardless of the encouraging results derived from the above implementations, there are some things necessary for the further development of our framework. Referring to our sensor node, a more energy-efficient communication protocol can be designed for the CPLD-mote platform in order to improve its energy and power consumption. Besides that, other cryptographic schemes or, at least, a stronger version of the existing scheme can be implemented on it for improving the security level offered. This target can be achieved by the use of a larger CPLD, the more available resources of which can enable us to implement such stronger algorithms.
In addition, the custom-made cable can be replaced together with all the development areas of the boards, since they waste a lot of energy. Instead of these, a custom board can be designed which will contain only a CPLD device, the micro-controller of the mote and the ZigBee module.
Moving to the base station platform, the first idea would be to change the input/ output protocol (RS232) with another of higher performance (i.e., Gigabit Ethernet) for taking advantage of the high values of throughput that can be achieved by our system.
Furthermore, we can implement the AES inverse cipher by using the implementations of the given one, which is not so complex, because some parts of the AES decryption procedure are the same compared with the specific parts of the encryption one and the remaining parts are very similar.
In this work, we implement the AES-128 cipher, so another idea is to develop a similar method for data-independent power consumption referring to other "flavors" such as AES-192 and AES-256 including both the cipher process and the inverse one.
Moreover, it might be useful to implement other cryptographic standards such DES and Triple-DES or algorithms such AES competition finalists (Twofish, Serpent, MARS and RC6) in the FPGA of our base station platform and use the method of improving the data-independence of power consumption in these designs. These

implementations can lead us to more secure conclusions about the performance of this method.

Another subject for further research will be to apply the complete method to different kind of security algorithms such authentication algorithms or digital signatures. Taking into account that the development of a new hash standard (SHA-3) is in progress, a research on the data-independence of its FPGA implementation power consumption would be a great addition.

To sum up, these extensions lead closer to efficient FPGA implementations of cryptographic algorithms with data-independent power consumption that makes them well-protected from a power consumption analysis attack.

Appendices

Appendix 1 - Sensor node interconnection

One issue of great importance according our sensor node platform is the way via which the wireless mote and the CPLD are going to be connected, but before planning this interconnection, we must to choose the I/O ports that is going to be used from both the devices.

This I/O issue was confronted with the following way; regarding the CPLD connection, the JTAG ports were chosen for data transfers between the motes and the CPLD and the mote connection, only 24 pins out of the 102 of the prototyping area are actually available since the remaining pins are either *open* or dedicated to a specific operation of the main micro-controller of the mote. Based on a traffic profiling of several applications and since it was necessary for this connection to be used in many applications, we decided to use 8 of those pins as an input to the mote, 8 for the output traffic and the remaining ones for several input/output control signals, as shown in Table 1.

In order to efficiently and correctly exchange data between the CPLD and the mote, a simple toggle synchronization protocol was also implemented both in software (on the motes) and in hardware (on the CPLD). As described in the algorithm architecture, the input and the output of the CPLD should be 16-bit long. As a result, a number of the mote and CPLD pins should be used 4 times for a single Mote-to-CPLD transfer. The protocol we have designed works as follows; firstly, the mote sends the first 8 bits to the CPLD. This datum is stored in a register. When the CPLD receives the first block of data, a toggle bit transits from its current state to the other, so as to ensure the data freshness and that the datum is received correctly from the CPLD. Additionally, a block-offset bit is used, to indicate which block is transferred each time. So, when the first 8 out of 16 bits are sent to the CPLD, the value of the block offset is 0, whereas the value of the block offset bit is 1 when the second byte is sent to the CPLD. After the successful reception of the first block of data, the CPLD sends an inversed toggle bit to the mote, so as to trigger the sending of the second block of data. The same procedure is also followed for the second input block, which is stored to another register in the CPLD. At this point the CPLD is ready to start the processing of the received data. After the completion of the data

MDA100CB Pin	Mode	CPLD Pin	Mode
F2	DATA (out)	p117	DATA (in)
F3	DATA (out)	p136	DATA (in)
F4	DATA (out)	p134	DATA (in)
F5	DATA (out)	p132	DATA (in)
F6	DATA (out)	p57	DATA (in)
F7	DATA (out)	p59	DATA (in)
F8	DATA (out)	p119	DATA (in)
F13	DATA (out)	p45	DATA (in)
C10	DATA (in)	p129	DATA (out)
C11	DATA (in)	p126	DATA (out)
C12	DATA (in)	p124	DATA (out)
C13	DATA (in)	p120	DATA (out)
D10	DATA (in)	p118	DATA (out)
D11	DATA (in)	p116	DATA (out)
D12	DATA (in)	p114	DATA (out)
D13	DATA (in)	p112	DATA (out)
E2	RESET (out)	p106	RESET (in)
E3	TOGGLE (in)	p137	TOGGLE (out)
E4	OFFSET (out)	p135	OFFSET (in)

Table 1: Custom cable pins

processing, the CPLD starts sending the data to the mote. The protocol used for sending the data to the mote is almost the same with the one used for receiving data from the mote: every block is stored in an output register in the CPLD. Initially, a toggle bit is sent to the mote; then, the mote replies and, upon the reception of the answer, the CPLD sends the first block with the correct value of the block offset. At this point, it should be mentioned that the output of our hardware module is 16-bit long. So, every output block is 8-bit long and, consequently, two blocks are sent to the mote. Upon correctly receiving the first block of data, the mote notifies the CPLD about this fact and the latter starts sending the second block of data to the mote. Once the transfer is finished, the mote notifies the CPLD and sends the data to the other nodes in the network.

The implementation of the communication protocol on the CPLD is presented in Figure 1, whereas, for better explanation, a timing diagram of this protocol is pre-

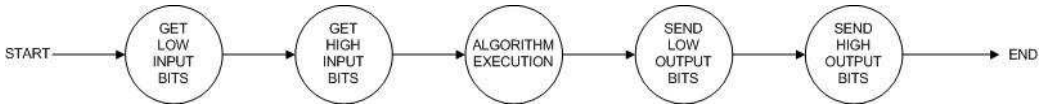

Figure 1: Sensor node communication protocol

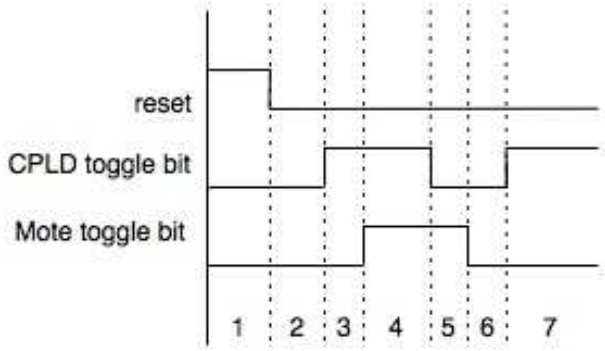

Figure 2: Sensor node communication timing diagram

sented in Figure 2.

Appendix 2 - RS-232 hardware module

As referred to the implementation chapter, serial port is used by the base station for sending/ receiving data to/ from the FPGA device. For this purpose, a RS-232 hardware module has been implemented in order to be used as the input/ output interface from the side of the reconfigurable device. An abstract block diagram of this module is illustrated in Figure 3.

The first components which are the main ones and are used for communicating with the external port are the UART receiver and transmitter; the first one is used for receiving data from the port and the other one for transmitting data to it. Another component used for the same purpose is called *baud* and is a clock divided utilized for adjusting the system clock with the UART one.

The remaining components of this module are used for certifying its correct functionality. RS-232 module also consists of two multiplexers (a 2-to-1 and a 16-to-1 one with 8-bit inputs/ output) and 32 8-bit registers. Finally, a memory containing 1024 8-bit long data is necessary for its right operation. It is generated by Xilinx CORE generator and captures an available block of the given FPGA RAM.

Furthermore, the use of a control unit is thought indispensable for securing the correct operation of this RS-232 module. According to the timing diagram defined, the whole procedure starts when the UART receiver is in waiting state. When an interrupt of receiving data is sent, the UART receiver is enabled in order to get this data. This data is received in quantities of 8 bits each one. Then, 32 clock cycles follow for receiving and storing the received 32 bytes (16 of the plaintext and 16 of the cipher key) to the RS-232 memory. The first multiplexer is set to select the output of the UART receiver in order to store the input data. The following 32 cycles are spent in order to read this input data byte-by-byte (reading each position of the memory) and store it to the 32 byte-registers, the outputs of which are connected to the input of AES module. This module calculates the specific ciphertext according the implementation chosen. The output of the calculation module which is one of these described in previous chapter (single-rail, dual-rail, 2-, 4- or 8-round duplicate DR) is divided in 16 8-bit quantities being the inputs of the second multiplexer. These 16 quantities are stored to the RS-232 memory in the following 16 clock cycles while

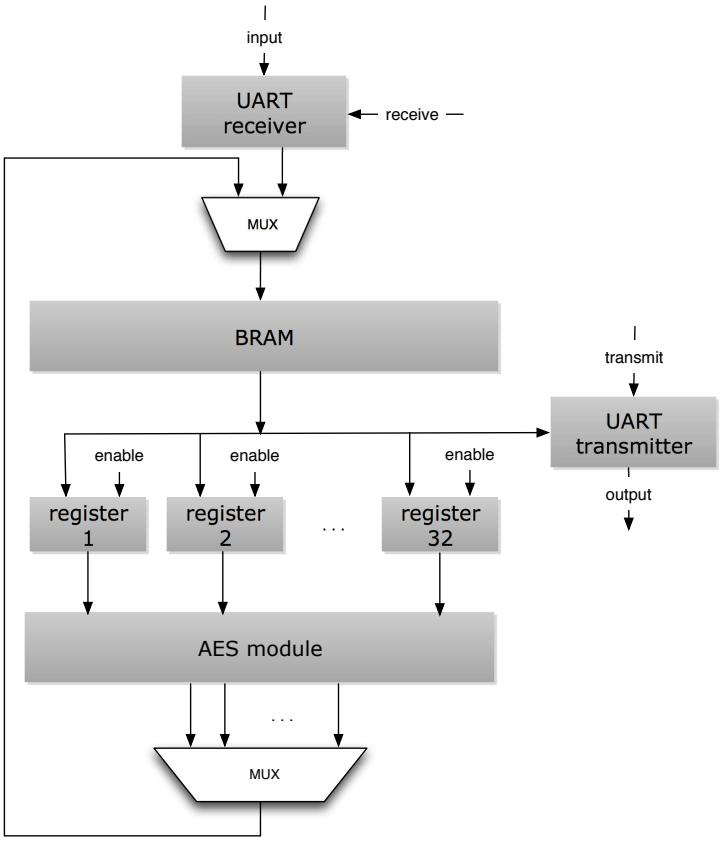

Figure 3: RS-232 module abstract block diagram.

the first multiplexer is set to select the derived system output. Finally, the output data are read from the BRAM and sent to the UART transmitter during the last 16 clock cycles in order to send it to the serial port enabling its transmit interrupt. The FSM scheme described above is schematically presented in Figure 4.

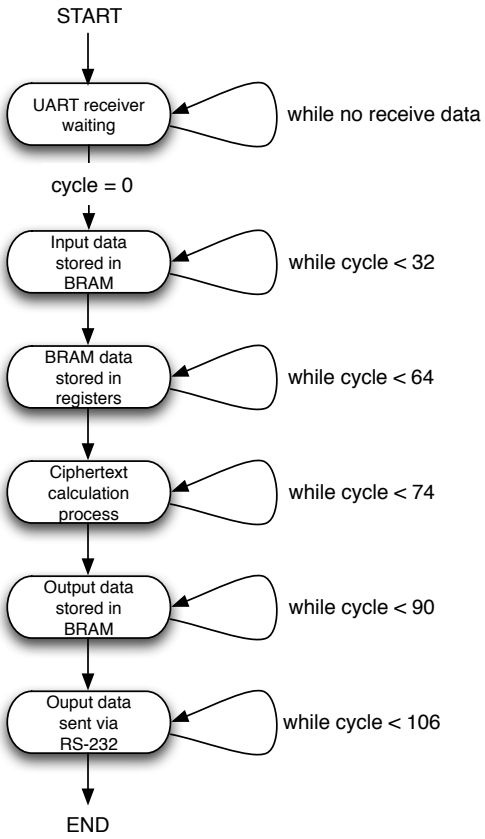

Figure 4: UART FSM scheme.

Appendix 3 - Acronyms

AES: Advanced Encryption Standard
CBC: Cipher Block Chaining
CPLD: Complex Programmable Logic Device
CPU: Central Processing Unit
DEMA: Differential EM Analysis
DoS: Denial-of-Service
DPA: Differential Power Analysis
DPROM: Dual-Port Read-Only Memory
EM: ElectroMagnetic
EP2P: Embedded Peer-to-Peer
FPGA: Field-Programmable Gate Array
FSM: Finite State Machine
IV: Initialization Vector
LEAF: Law Enforcement Access Field
LSB: Least Significant Bit
MSB: Most Significant Bit
NCL: Null-Convention Logic
NesC: Network embedded systems C
NIST: National Institute of Standards and Technology
NSA: National Security Agency
P2P: Peer-to-Peer
ROM: Read-Only Memory
RTL: Register Transfer Level
SCA: Side-Channel Analysis
SEMA: Simple EM Analysis
SPA: Simple Power Analysis
VHDL: VHSIC Hardware Description Language
VHSIC: Very High Speed Integrated Circuits
WSN: Wireless Sensor Network

Bibliography

[1] C. S. Raghavendra, *Wireless Sensor Networks*. Springer, July 2005.

[2] S. Brown, "FPGA architectural research: a survey," *Design & Test of Computers, IEEE*, vol. 13, no. 4, pp. 9–15, 1996.

[3] *Xilinx Online*. `http://www.xilinx.com`, Last accessed: 2009/05/15.

[4] *Altera Online*. `http://www.altera.com`, Last accessed: 2009/05/15.

[5] R. Anderson and M. Kuhn, "Tamper resistance: a cautionary note," in *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 1996.

[6] *Internet Denial-of-Service Considerations*. IETF Network Working Group, November 2006.

[7] V. D. Gligor, "A note on the denial-of-service problem," in *SP '83: Proceedings of the 1983 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), p. 139, IEEE Computer Society, 1983.

[8] J. R. Douceur, "The sybil attack," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, (London, UK), pp. 251–260, Springer-Verlag, 2002.

[9] J. M. McCune, E. Shi, A. Perrig, and M. K. Reiter, "Detection of denial-of-message attacks on sensor network broadcasts," in *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 64–78, IEEE Computer Society, 2005.

[10] C. Karlof, N. Sastry, and D. Wagner, "TinySec: a link layer security architecture for wireless sensor networks," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, (New York, NY, USA), pp. 162–175, ACM, 2004.

[11] F. Ye, H. Luo, S. Lu, and L. Zhang, "Statistical en-route filtering of injected false data in sensor networks," in *INFOCOM*, pp. 839–850, 2004.

[12] Y.-C. Hu, D. B. Johnson, and A. Perrig, "SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks," in *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, (Washington, DC, USA), p. 3, IEEE Computer Society, 2002.

[13] C. Hartung, J. Balasalle, and R. Han, "Node compromise in sensor networks: The need for secure systems," tech. rep., Department of Computer Science, University of Colorado, Boulder, January 2005.

[14] H. Chan, A. Perrig, B. Przydatek, and D. Song, "SIA: Secure information aggregation in sensor networks," *J. Comput. Secur.*, vol. 15, no. 1, pp. 69–102, 2007.

[15] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, (London, UK), pp. 104–113, Springer-Verlag, 1996.

[16] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2003.

[17] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," Tech. Rep. CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[18] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi, "Cryptanalysis of block ciphers implemented on computers with cache," in *25th International Symposium on Information Theory and Its Applications (ISITA 2002)*, 2002.

[19] D. J. Bernstein, "Cache-timing attacks on AES," 2004.

[20] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan 2005*, 2005.

[21] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, (London, UK), pp. 167–182, Springer-Verlag, 2000.

[22] A. Hevia and M. Kiwi, "Strength of two data encryption standard implementations under timing attacks," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 4, pp. 416–437, 1999.

[23] F. Koeune, J.-J. Quisquater, and J.-J. Quisquater, "A timing attack against Rijndael," tech. rep., 1999.

[24] H. Handschuh and H. M. Heys, "A timing attack on RC5," in *SAC '98: Proceedings of the Selected Areas in Cryptography*, (London, UK), pp. 306–318, Springer-Verlag, 1999.

[25] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," *Lecture Notes in Computer Science*, vol. 1666, pp. 388–397, 1999.

[26] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smart cards," in *WOST'99: Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 1999.

[27] L. Goubin and J. Patarin, "DES and differential power analysis (the "duplication" method)," in *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, (London, UK), pp. 158–172, Springer-Verlag, 1999.

[28] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, (London, UK), pp. 251–261, Springer-Verlag, 2001.

[29] J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and counter-measures for smart cards," in *E-SMART '01: Proceedings of the International Conference on Research in Smart Cards*, (London, UK), pp. 200–210, Springer-Verlag, 2001.

[30] S. Mangard, "Exploiting radiated emissions - EM attacks on cryptographic ICs," in *Proceedings of Austrochip 2003* (L. Ostermann, ed.), pp. 13 – 16, 2003.

[31] B. Schneier, *Applied Cryptography*. John Wiley & Sons, 1996.

[32] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[33] M. Liskov, R. L. Rivest, and D. Wagner, "Tweakable block ciphers," in *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, (London, UK), pp. 31–46, Springer-Verlag, 2002.

[34] N.I.S.T., "DES modes of operation." FIPS Publication 81, December 1980.

[35] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation," in *Proceedings of 38th Annual Symposium on Foundations of Computer Science (FOCS 97)*, 1997.

[36] N.I.S.T., "Announcing the Advanced Encryption Standard (AES)." FIPS Publication 197, November 2001.

[37] M. Robshaw, "Stream ciphers," 1995.

[38] RC*4 Page*. `http://www.wisdom.weizmann.ac.il/~itsik/RC4/rc4.html`, Last accessed: 2009/03/05.

[39] G. Rose, "A precis of the new attacks on GSM encryption," in *Proceedings of QUALCOMM*, September 2003.

[40] U. Blöcher and M. Dichtl, "Fish: A fast software stream cipher," in *Fast Software Encryption, Cambridge Security Workshop*, (London, UK), pp. 41–44, Springer-Verlag, 1994.

[41] D. Whiting, B. Schneier, S. Lucks, and F. Muller, "Phelix - fast encryption and authentication in a single cryptographic primitive," in *Proc. Fast Software Encryption 2003, volume 2887 of LNCS*, pp. 330–346, Springer-Verlag, 2003.

[42] R. J. Jenkins, "ISAAC," in *Fast Software Encryption, Cambridge Security Workshop*, (London, UK), pp. 41–49, Springer-Verlag, 1996.

[43] D. Watanabe, S. Furuya, H. Yoshida, K. Takaragi, and B. Preneel, "A new keystream generator MUGI," in *FSE '02: Revised Papers from the 9th International Workshop on Fast Software Encryption*, (London, UK), pp. 179–194, Springer-Verlag, 2002.

[44] J. D. Golic, "A weakness of the linear part of stream cipher MUGI," in *FSE*, pp. 178–192, 2004.

[45] A. Biryukov and A. Shamir, "Analysis of the non-linear part of MUGI," in *FSE*, pp. 320–329, 2005.

[46] *Panama Page.* `http://www.quadibloc.com/crypto/co4821.htm`, Last accessed: 2009/03/05.

[47] *Software-efficient pseudorandom function and the use thereof for encryption.* U.S. Patent No. 5,454,039, September 1995.

[48] *Computer readable device implementing a software-efficient pseudorandom function encryption.* U.S. Patent No. 5,675,652, October 1997.

[49] D. Wantanabe, S. Furuya, and T. Kaneko, "A MAC forgery attack on SOBER-128*," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E88-A, no. 5, pp. 1166–1172, 2005.

[50] P. Hawkes and G. Rose, "Primitive specification for SOBER-128." IACR ePrint archive, 2003.

[51] A. Beutelspacher, *The Future Has Already Started or Public Key Cryptography.* 1994.

[52] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish encryption algorithm: a 128-bit block cipher.* New York, NY, USA: John Wiley & Sons, Inc., 1999.

[53] *Serpent Page.* `http://www.cl.cam.ac.uk/~rja14/serpent.html`, Last accessed: 2009/03/05.

[54] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (Blowfish)," in *Fast Software Encryption, Cambridge Security Workshop*, (London, UK), pp. 191–204, Springer-Verlag, 1994.

[55] C. M. Adams, "Constructing symmetric ciphers using the cast design procedure," *Des. Codes Cryptography*, vol. 12, no. 3, pp. 283–316, 1997.

[56] R. C. Merkle and M. E. Hellman, "On the security of multiple encryption," *Commun. ACM*, vol. 24, no. 7, pp. 465–467, 1981.

[57] X. Lai and J. L. Massey, "A proposal for a new block encryption standard," in *EUROCRYPT '90: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, (New York, NY, USA), pp. 389–404, Springer-Verlag New York, Inc., 1991.

[58] D. Sokolov, J. Murphy, A. Bystrov, and A. Member-Yakovlev, "Design and analysis of dual-rail circuits for security applications," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 449–460, 2005.

[59] M. A. Kishinevskii and A. V. Yakovlev, *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1990.

[60] I. David, R. Ginosar, and M. Yoeli, "An efficient implementation of boolean functions as self-timed circuits," *IEEE Trans. Comput.*, vol. 41, no. 1, pp. 2–11, 1992.

[61] A. Kondratyev and K. Lwin, "Design of asynchronous circuits using synchronous CAD tools," *IEEE Design and Test of Computers*, vol. 19, no. 4, pp. 107–117, 2002.

[62] K. Fant and S. Brandt, "Null convention logic: A complete and consistent logic for asynchronous digital circuit synthesis," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, vol. 0, p. 261, 1996.

[63] D. Sokolov, J. Murphy, A. Bystrov, and A. Yakovlev, "Improving the security of dual-rail circuits," in *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES)*, 2004.

[64] N.S.A., "SKIPJACK and KEA algorithm specifications," tech. rep., May 1998.

[65] C. The White House, "Fact sheet: public encryption management," pp. 420–422, 1997.

[66] "Escrowed Encryption Standard (EES)," Tech. Rep. FIPS PUB 185, U.S. Department of Commerce, February 1994.

[67] B. Kaliski, "A survey of encryption standards," *IEEE Micro*, vol. 13, no. 6, pp. 74–81, 1993.

[68] X. Yi, S. X. Cheng, X. H. You, and K. Y. Lam, "A method for obtaining cryptographically strong 8x8 S-boxes," in *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 2, pp. 689 – 693, Nov 1997.

[69] J. Daemen and V. Rijmen, "AES proposal: Rijndael," 1999.

[70] J. Daemen and V. Rijmen, "The block cipher Rijndael," in *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, (London, UK), pp. 277–284, Springer-Verlag, 2000.

[71] M. Dworkin, *Recommendation for Block Cipher Modes of Operation*. National Institute of Standards and Technology, NIST special publication 800-38a ed., 2001.

[72] Crossbow, *MPR-MIB Users Manual*, June 2007. Revision A.

[73] Crossbow, *MTS/ MDA Sensor Board Users Manual*, June 2007. Revision A.

[74] Xilinx, *CoolRunner-II CPLD Family*. Xilinx, v3.1 ed., September 2008.

[75] Digilent, *X-Board Reference Manual*, January 2007.

[76] Xilinx, *ISE Design Suite 10.1 Release Notes and Installation Guide*.

[77] ModelTech, *ModelSim Users Manual*, 6.3g ed., May 2008.

[78] *Tiny*OS *Online Tutorial*. `http://www.tinyos.net`, Last accessed: 2009/03/05.

[79] *Nes*C *Online Tutorial*. `http://en.wikipedia.org/wiki/NesC`, Last accessed: 2009/03/05.

[80] *Blowfish Online*. `http://www.schneier.com/blowfish.html`, Last accessed: 2009/08/04.

[81] VASG*:* VHDL *Analysis and Standardization Group*. `http://www.eda.org/vhdl-200x/`, Last accessed: 2009/08/19.

[82] Digilent, *Virtex*-II *Pro Development System*. `http://www.digilentinc.com/Products/Detail.cfm?Prod=XUPV2P`, Last accessed: 2009/08/19.

[83] Xilinx, *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, November 2007.

[84] Intel, *Intel Desktop Board D945GCLF2, Technical Product Specification*, December 2008.

[85] *Xubuntu Online*. `http://www.xubuntu.com`, Last accessed: 2009/05/15.

[86] *Python Online*. `http://www.python.org`, Last accessed: 2009/05/15.

[87] *PySerial documentation*. `http://pyserial.sourceforge.net/`, Last accessed: 2009/08/19.

[88] *Socket: Low-level networking interface*. `http://docs.python.org/library/socket.html`, Last accessed: 2009/08/19.

[89] A. Hodjat and I. Verbauwhede, "A 21.54 Gbits/s fully pipelined AES processor on FPGA," in *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 308–309, IEEE Computer Society, 2004.

[90] *Intel* VT*une* P*erformance* A*nalyzer*. `http://software.intel.com/en-us/intel-vtune/`, Last accessed: 2009/06/29.

[91] P*ower*T*op*. `http://www.lesswatts.org/projects/powertop/`, Last accessed: 2009/06/29.

[92] Intel, *Intel* A*tom* P*rocessor 330 Systems Datasheet*, February 2009. Revision 002.