Hardware architecture for elliptic curve cryptography and lossless data compression

by

Miguel Morales-Sandoval

A thesis

presented to the National Institute for Astrophisics, Optics and Electronics
in partial fulfilment of the
requirement for the degree of

Master of Computer Science

Thesis Advisor:
Dra. Claudia Feregrino-Uribe

Computer Science Department
National Institute for Astrophisics, Optics and Electronics
Tonantzintla, Puebla
México
December 2004

Abstract

Data compression and cryptography play an important role when transmitting data across a public computer network. While compression reduces the amount of data to be transferred or stored, cryptography ensures that data is transmitted with reliability and integrity. Compression and encryption have to be applied in the correct way: data are compressed before they are encrypted. If it were the opposite case the result of the cryptographic operation would be illegible data and no patterns or redundancy would be present, leading to very poor or no compression. In this research work, a hardware architecture that joins lossless compression and public-key cryptography for secure data transmission applications is discussed. The architecture consists of a dictionary-based lossless data compressor to compress the incoming data, and an elliptic curve cryptographic module that performs two EC (elliptic curve) cryptographic schemes: encryption and digital signature. For lossless data compression, the dictionary-based LZ77 algorithm is implemented using a systolic array aproach. The elliptic curve cryptosystem is defined over the binary field F_{2^m}, using polynomial basis, affine coordinates and the binary method to compute an scalar multiplication. While the model of the complete system was implemented in software, the hardware architecture was described in the Very High Speed Integrated Circuit Hardware Description Language (VHDL). A prototype of the architecture was implemented on a Xilinx Virtex II *Field Programmable Gate Array* (FPGA) device. Two advantages of combining lossless compression and public-key encryption were demonstrated: 1) the improvement in the cryptographic module by reducing the amount of data to be encrypted, and 2) a better utilization of the available bandwidth when encrypted data is transmitted across a public computer network.

Resumen

Cryptografía y compresión de datos son dos tecnologias centrales en aplicaciones de red. La compresión reduce la cantidad de datos permitiendo ahorro de espacio de almacenamiento o reducción de tiempo para realizar la tranferencia de los datos. Por otra parte, la criptografía garantiza que los datos se transmitan con confiabilidad e integridad. Cuando ambos algoritmos se combinan, la compresión debe ralizarse primero y a continuación el cifrado. Si la aplicación de los algoritmos se realizara en el orden inverso, el resultado de la operación de cifrado daría como resultado datos totalmente inenteligibles con muy poca o nula redundancia. Cuando la compresión fuera aplicada, el resultado sería pobre o nula.

En este trabajo de investigación se presenta una arquitectura hardware que combina compresión de datos sin pérdida y criptografía de llave pública para aplicaciones de transmisión segura de datos. La arquitectura consiste de un compresor de datos sin pérdida basado en diccionario y un módulo que ejecuta dos esquemas de criptografía de curvas elípticas (ECC): cifrado y firma digital. El método de compresión seleccionado es el LZ77 implementado bajo el enfoque de arreglos sistólicos. El criptosistema de curvas elípticas se define sobre el campo binario F_{2^m} usando base polinomial y coordenadas affine. El método binario es utilizado para realizar la operación de multiplicación escalar.

Una implementación en software del sistema propuesto fue realizada a fin de validar la arquitectura hardware. El sistema completo fue descrito usando el lenguaje estándar para descripción de hardare VHDL, fue simulado en Active-VHDL y sintetizado para la familia de FPGAS (*Field Programmable Gate Array*) Virtex II de Xilinx. Las ventajas que presenta el enfoque de combinar compresión de datos com criptografía son: 1) el mejoramiento en el desempeño del modulo criptográfico al reducir la cantidad de datos a ser procesados y, 2) una mejor utilización del ancho de banda disponible cuando la información se transmite a través de una red.

Acknowledgements

I want to thank the Consejo Nacional de Ciencia y Tecnología (CONACyT) for financial support through scholarship number 171577. I thank the Instituto Nacional de Atrofísica, Óptica y Electrónica (INAOE) for academic formation and different supports and services.

I am grateful to my advisor Dra. Claudia Feregrino-Uribe for her reviews, help, guidance and encouragement in conducting this research. I count myself as being the most fortunate to be able to work under her supervision.

The same acknowledgement should also go to Dr. René Cumplido-Parra, Dr. Miguel Arias-Estrada and Dr. Guillermo-De Ita-Luna who acted as my graduate committee for their valuable input.

I wish I would never forget the company I had from my friends at the Institute. In particular, I am thankful to Alberto-Téllez, Moisés-Gutiérrez, Ignacio-Algredo, Eric-Rodríguez and Carlos-Diaz.

This thesis might not exist at all without the love and support of my family. The love and moral support of my sisters Lucila and Esperanza, my brothers Victor and Alfredo. My mother Anastacia for her love, encouragement and inspiration, I can never thank her enough. My niece Jasmine has made my life much more cheerful and colorful.

Lastly but not least, thanks are also due to Leticia-Flores and Manuel-Juárez for their help and friendship. Many more persons participated in various ways to ensure my research succeeded than those and I am thankful to them all.

To my father[†]*, by his teachings an example.*

Contents

Preface

Two of the most interesting areas in computer science are data compression and cryptography, mainly in applications of data transmission across a public computer network.

On one hand, because of the information processing and telecommunications revolutions, there is an increasing demand for techniques to keep information secret, to determine that information has not been forged and to determine who authored pieces of information. On the other hand, a common problem in computer data networks has been always the transfer data rate; the most important technique to improve the performance of a network is data compression. Data compression benefits in the sense that the process *compression-transmission-decompression* is faster than the process of transmitting data without compression.

So, while compression reduces the amount of data to be transferred or stored, cryptography ensures that data is transmitted with reliability and integrity. Using a data compression algorithm together with an encryption algorithm, in the correct order, makes sense for three reasons:

- Compressing data can reduce the redundancies that can be exploited by cryptanalysts to recover the original data.
- Compressing data can speed-up the encryption process.
- If encrypted data are transmitted in a computer network, the bandwidth can be better utilized.

Although some effort has been done by some enterprises for combining compression and cryptography, asymmetrical encryption has not been considered. Traditionally, public key cryptography has been used only to generate a shared secret value, which is used for bulk encryption.

This work explores the combination of lossless data compression and Elliptic Curve Cryptography (ECC) in a single hardware architecture in order to demonstrate the three advantages listed above. ECC employs smaller length keys than other cryptosystems like RSA, what implies less space for key storage and less costly modular operations.

Furthermore, it has been shown in the literature that ECC's security is higher than that provided by RSA, which is the most widely used public key cryptosystem.

Hardware implementation of compression and cryptographic algorithms are better suited than software implementations. When implementing compression algorithms, the search for redundancy implies many complex operations that can not be implemented efficiently with the available instruction set of a general purpose processor (like the ones used in personal computers). And when cryptographic algorithms are implemented, it is necessary to perform a high amount of mathematical operations between large numbers in a finite field. Again, general purpose processors do not have instructions to support these operations, leading to inefficient implementations. A hardware solution is well suited to implement both kinds of algorithms, especially for real time data processing.

In addition, field programmable devices like FPGAs (Filed Programmable Gate Array) and advanced synthesis tools make possible the development of a hardware architecture that allows exploring such algorithm combination. FPGAS gathers the advantages of hardware and software and so, they are preferred to use in this implementation.

According to the review of the literature, this work is novel in the sense that there is no hardware implementation where lossless data compression and elliptic curve cryptography have been considered jointly, neither a hardware implementation of elliptic curve cryptographic schemes.

Chapter 1

Introduction

This chapter gives an outline on this research work. It gives an introduction to the field of study, states the tackled problem and the motivation, exposes the research goals and the selected methodology. Finally, the organization of the thesis is presented at the end of the chapter.

1.1 Introduction

Data compression and cryptography play an important role when transmitting data across a public computer network. Theoretically, compression and cryptography are opposite: while cryptography converts some legible data into some totally illegible data, compression searches for redundancy or patterns in data to be eliminated in order to get a reduction of data.

The basic foundations of both, compression and encryption are presented in the following sections. Most of the information about compression was gathered from [1] and [2]. The information about cryptography can be extended in [3] and [4].

1.1.1 Data compression

A common problem in computer data networks has been always the transfer data rate; the most important technique to improve the performance of a network is data compression. Data compression benefits in the sense that the process *compression-transmission-decompression* is faster than the process of transmitting data without compression.

Data compression is the process of encoding information using fewer bits or information units by using specific encoding schemes. The compression algorithms search for redundancy in data and remove it, so the more the redundancy the more the compression ratio achieved. Because most real-world data are very redundant, compression is often possible, although there is not compression algorithm that performs well for all

kind of data [1]. Compression helps to reduce the consumption of expensive resources, such as disk space or connection bandwidth at the expense of data processing power, which can also be expensive. That is because the searching for redundancy implies many operations, many times complex.

The compression algorithm performance is measured according to the compression ratio it achieves; this measurement is obtained following equation 1.1, where $Size_{out}$ means the size in bits of the processed data, which original size is $Size_in$. A good data compressor must achieve a compression ratio less or equal to 0.5.

$$R_c = \frac{Size_{out}}{Size_{in}} \tag{1.1}$$

Compression is divided in *lossy* and *lossless compression*. In lossless data compression, it is possible to recover the original data from their compressed form. On the contrary, in lossy data compression, only an approximation of data is recovered. Lossless data compression is commonly used in applications where the loss of a bit of information is not accepted, like in text and executable files. Lossy compression is used for images, audio and video, achieving better compression ratios than lossless methods. The challenge has been always to ideate a compression method that executes in shortest time, achieves the highest compression ratio, requires few computational resources and do not lose data.

Lossless compression

For lossless data compression, algorithms can be dictionary-based or statistical. A statistical compressor can achieve better compression ratio than a dictionary-based method but its computational complexity is higher [1]. In both statistical and dictionary-based methods, a trade off between compression ratio and computational resources needs to be established; depending on the application, not always the best compression method is required. In table 1.1 are listed some lossless compressors, statistical and dictionary-based.

Statistical methods use the statistical properties of data to assign variable length codes to each individual symbol in data. Often, these methods are composed of two parts: a probability model and the compressor itself [1]. The statistical methods proposed in the literature, differ in the model used to get the statistics of the symbols [2]. The compression ratio depends on how good this model gets such statistics, often the statistics of the symbols are obtained according to the context in which they occur. Given the probabilities of the symbols, the compressor assigns to each symbol a variable length bit string to get the final bitstream for transmission or storage, this process is called *coding*. The statistical methods most commonly used are Huffman coding,

Table 1.1: Lossless compression methods

Lossless methods	
Dictionary-based	Statistical
LZ77	Huffman coding
LZSS	Arithmetic coding
LZW	PPM
LZS	Shannon-Fano
BTW	Adaptive Huffman

Arithmetic coding and PPM [1].

On the contrary, dictionary-based methods do not use the statistical properties of the symbols, instead of that, they use the principle of replacing symbol strings with *codewords* identifying that string in a dictionary [2]. The dictionary contains a list of substrings and codewords associated with them. Almost all dictionary-based lossless compressors are based on the work of Jacov Ziv and Abraham Lempel from the 70's [5], [6]. The main differences between dictionary-based methods are about how dictionary is implemented and in the imposed limitations on what the pointers can represent. In these methods, compression is achieved only if the length of the pointer in bits is less than the length of the string replaced. Some dictionary-based methods are LZ77, LZ78 and LZW [1]. Practical software implementations of dictionary methods are compress, bzip2, gzip, zoo, pkzip and WinZip.

Lossy compression

In lossy compression, compressing data and then decompressing them retrieves data that may well be different to the original, but are close enough to be useful in some way. It is used a lot on the Internet and especially in streaming media (delivery of multimedia information just in time) and telephony applications. These methods are typically referred to as codecs (coder/decoder), which describes a device or program capable of performing transformations on a data stream or signals. Lossy methods are more often used for compressing sound or images. In these cases, the retrieved file can be quite different to the original at the bit level while being indistinguishable to the human ear or eye for most practical purposes. Many methods focus on the idiosyncrasies of the human anatomy, taking into account, for example, that the human eye can see only certain frequencies of light. The psychoacoustic model describes how sound can be highly compressed without degrading the quality of the sound. Some lossy methods and their application are summarized in table 1.2.

The advantage of lossy methods over lossless methods is that in some cases a lossy

Table 1.2: Lossy compression methods

Lossy methods		
Image	Audio	Video
Fractal compression	AAC	H.261
JPEG	ADPCM	H.263
Dolby	ATRAC	MNG
Wavelet compression	AC-3	Motion JPEG
	MP2	MPEG-1
	MP3	MPEG-2
	Ogg Vorbis	MPEG-4
	WMA	Ogg Theora
	CELP	Sorenson video codec
	G.711	
	G.726	
	HILN	

method can produce a much smaller compressed file than any known lossless method, while still meeting the requirements of the application.

1.1.2 Cryptography

Cryptography derives from the Greek *kryptós*, that means hidden, and *gráphein* meaning writing. Cryptography is traditionally, the study of ways to convert information (messages) from their normal, comprehensible form into an obscured guise, unreadable without special knowledge. The process of hiding the substance of information (*plaintext*) is called *encryption* and the process of turning encrypted data (*ciphertexts*) back into plaintext is called *decryption*. Contrary to cryptography, *Cryptoanalysis* intends to break the ciphertext, that is, to recover the original information from the ciphertext without knowing how it was encrypted.

In the past, cryptography helped to ensure secrecy in important communications, such as those of spies, military, leaders, and diplomats. In recent decades, the field of cryptography provides mechanisms for more than just keeping secrets: schemes like digital signatures and digital cash for example. Secondly, cryptography has come to be in widespread use by many civilians (not military people) who do not have extraordinary needs for secrecy, although typically it is transparently built into the infrastructure and telecommunications. Early in the 20th century, several mechanical devices were invented for performing encryption, including rotor machines. The most famous machine is the *Enigma* cipher used in World War II. The ciphers implemented by these machines brought about a significant increase in the complexity of cryptanalysis. The various

attacks on Enigma, for example, succeeded only after considerable effort. With the advent of digital computers and electronics, very complex ciphers could be implemented. A characteristic of computer ciphers is that they operate on binary strings, unlike classical and mechanical schemes which use an alphabet of about 26 letters, depending on the language.

Cryptography is an interdisciplinary subject, drawing from several fields. Before the time of computers, it was closely related to linguistics. Nowadays, cryptography makes extensive use of technical areas of mathematics, notably number theory, information theory, computational complexity, statistics and finite (discrete) mathematics.

Cryptography is commonly used for securing communications. Four properties are desirable and cryptography provides them, they are:

- *Confidentiality.* The information access is limited only to authorized entities.
- *Authentication.* Ensures that parties involved in a transaction are who they say they are. Also it lets to establish the origin of data.
- *Integrity.* Information is protected from alterations when it is transmitted. Data alterations can be deletion, insertion and substitution.
- *Non-repudiation.* It prevents an entity deny it had been involved in a previous action.

A cryptographic algorithm, also called a *cipher*, is one that provides one of the properties or services listed above. This algorithm is called *restricted* if it bases its security on keeping the way of working in secret. This kind of algorithms are rarely used today, they are popularly used only for low-security applications. Modern cryptographic algorithms base its security on the use of a *key*. The encryption and decryption functions, denoted as E and D respectively, use and depend on that key.

Similar to the compression methods, cryptography is classified in two groups, *symmetric* and *asymmetric cryptography* [3]. This classification is according to how they use the key for performing the cryptographic operations.

Symmetrical algorithms

Symmetric cryptography provides only the confidentiality service through encryption and decryption functions. Here, encryption and decryption are denoted as E_k and D_k respectively, indicating that both depend on the same key k. This is shown in figure 1.1.

Symmetrical algorithms require that the sender and receiver agree on a key before they can communicate securely. Because often the symmetrical algorithm is well know,

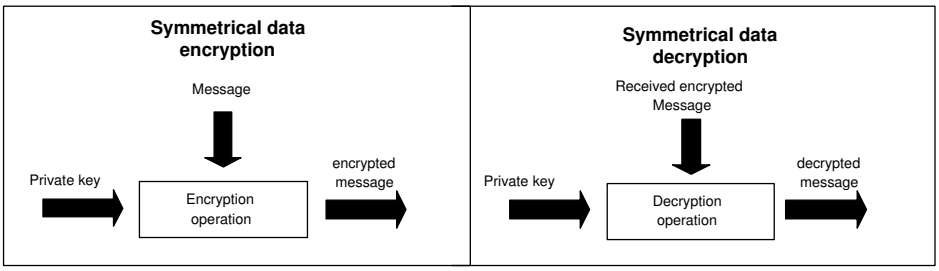

Figure 1.1: Symmetrical cryptography

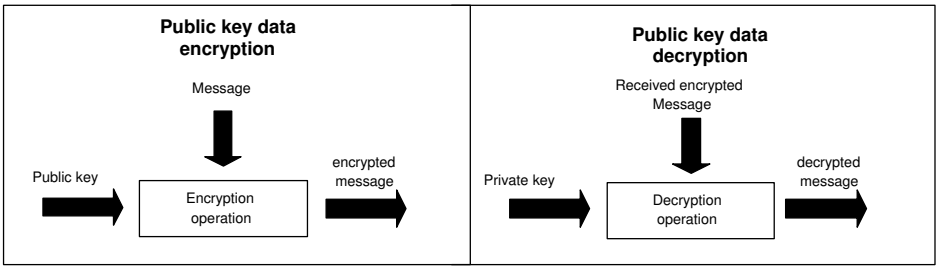

Figure 1.2: Asymmetrical cryptography

the security rests on the key. This key must be kept secret as long as the communication needs to remind secret. Symmetrical algorithms can be divided into two categories: *block ciphers* and *stream ciphers*. The first ones operate on plaintext one bit or byte at a time. The second ones operate in groups of bits or bytes, a typical block size used is 64 bits. Examples of symmetrical algorithms are AES, FEAL, IDEA, DES, and 3DES [3].

Asymmetrical or public-key algorithms

Public-key cryptography was invented in 1976 by Withfield Diffie and Martin Hellman [7]. In these algorithms two different keys are used, one for encryption, the *public key*, and other for decryption, the *private key*. This is shown in figure 1.2.

The use of two different keys solves the distribution key problem that exists in symmetric algorithms.

Public key cryptography provides the authentication, integrity and no-repudiation services by using the concept of *digital signature*. A digital signature is similar to a hand writing signature; it is used as proof of authorship of, or the agreement with, the content of a document.

A digital signature is represented in a computer as a string of binary digits. A digital signature is computed using a set of rules and a set of parameters such that the identity of the signatory and integrity of the data can be verified. Signature generation makes

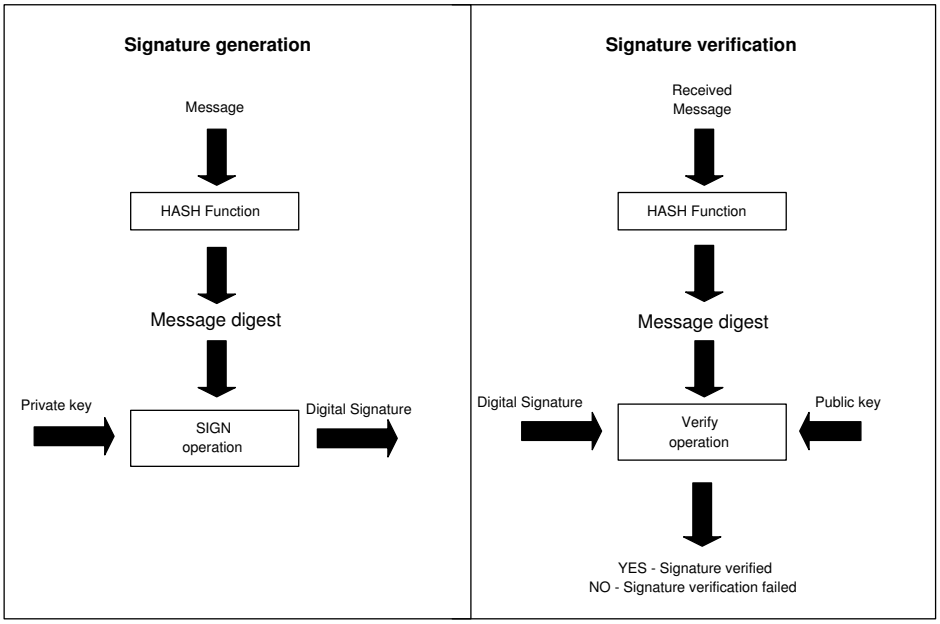

Figure 1.3: Digital signature

use of a private key to generate a digital signature. Signature verification makes use of a public key which corresponds to, but is not the same as, the private key. Each user possesses a private and public key pair. Public keys are assumed to be known to the public in general. Private keys are never shared. Anyone can verify the signature of a user by employing that user's public key. Signature generation can be performed only by the possessor of the user's private key.

Figure 1.3 shows the signature generation and verification operations. A hash function [3] is used in the signature generation process to obtain a condensed version of data, called a message digest. The message digest is then input to the digital signature algorithm to generate the digital signature. The digital signature is sent to the intended verifier along with the signed data (often called the message). The verifier of the message and signature verifies the signature by using the sender's public key. The same hash function must also be used in the verification process.

A hash function H is a transformation that takes a variable-size input m and returns a fixed-size bit string, which is called the hash value h ($h = H(m)$). Hash functions have a variety of general computational uses but when employed in cryptography, the hash functions need to have the following properties [3]:

1. $H(m)$ is relatively easy to compute for any given message m.
2. $H(m)$ is one-way.
3. $H(m)$ is collision-free.

A hash function H is said to be one-way if given a hash value h, it is computationally infeasible to find some input m such that $H(m) = h$. If a message m_1 is given, it is computationally infeasible to find a message m_2 different to m_1, such that $H(m_1) = H(m_2)$, then H is said to be a collision-free hash function. A hash function enables the determination of a message's integrity: any change to the message will result, with a very high probability, in a different message digest. This property is used in the generation and verification of digital signatures and message authentication code algorithms. Examples of hash functions are MD2, MD5 and SHA-1 [4].

A *MAC* algorithm is intended for determining if data have been modified when transmitted over a computer network. For example, in a text the word *do* may have been changed to *do not* or $1,000 may have been changed to $ 3,000. Without additional information, a human could easily accept the altered data as being authentic. These threats may still exist even when data encryption is used. It is therefore desirable to have an automated mechanism for detecting both intentional and unintentional modifications to data. A cryptographic Data Authentication Algorithm (DAA) can protect against both accidental and intentional, but unauthorized, data modification.

MAC stands for *Message Authentication Code*. In general, a MAC can be thought of as a checksum for data passed through an unreliable (or more importantly, unsecure) pipeline. A sender will typically generate a MAC code by first passing their message into some MAC algorithm. The sender will then send their message D with the $MAC(D)$ value. The receiver can then generate their own $MAC_1(D)$ and verify if the sent MAC value matches the currently computed one. A MAC algorithm can be generated using multiple different techniques; however, sender and receiver generally need to have a shared secret key, k_{MAC}.

A hash function alone cannot act as a MAC function. An attacker could intercept the message D and $Hash(D)$. He could then resend as D' and $Hash(D')$. The receiver could then not tell that the message had been altered. In other words, Hash functions can help prevent error in an unreliable channel, but not in an insecure channel.

Although public key cryptography was proposed in 1976, the first cryptosystem appeared in 1978, it was the RSA cryptosystem. This is the most accepted public-key cryptosystem and used in all over the world. Other cryptosystems are Diffie-Hellman [7], ElGamal, DSA, Rabin [4] and ECC [8]. Elliptic curve cryptography is a type of public-key algorithm based on the mathematics of elliptic curves (defined by certain cubic equations). It has been claimed that ECC can be faster and use shorter keys than other public-key algorithms available while providing an equivalent level of security. Using short keys implies lower storage requirements and faster arithmetic computations. This is specially important when implementing public-key cryptography on constrained environments, like in mobile and wireless devices.

The security of public-key algorithms is usually based on hard mathematical problems. Just three conjectured hard problems have been cosidered in public-key cryptographic schemes: big integer factorization, computation of discrete logarithms and the computation of discrete logarithms in elliptic curves. Because computations in public-key algorithms are more complex than in symmetrical ones, for efficiency reasons, hybrid encryption systems are used in practice; a key is exchanged using a public-key cipher, and the rest of the communication is encrypted using symmetrical encryption, which is typically much faster.

The security of all practical encryption schemes remains unproven, both for symmetric and asymmetric ones. For symmetric ciphers, confidence gained in an algorithm is usually anecdotal, no successful attack has been reported on an algorithm for several years despite intensive analysis. For asymmetric schemes, it is common to rely on the difficulty of the associated mathematical problem, but this, is not provably secure.

Extensive academic research into modern cryptography is relatively recent, it began in the open community during the 1970s with the specification of DES and the invention of RSA algorithms. Popular applications such as the Internet and mobile phones have repositioned cryptography.

1.2 Platforms for algorithm implementation

There are two primary methods in conventional computing for the execution of algorithms [9]. The first one is to use hardwired technology, either an Application Specific Integrated Circuit (*ASIC*) or a group of individual components forming a board-level solution, to perform the operations in hardware. ASICs are designed specifically to perform a given computation, and thus they are very fast and efficient when executing the exact computation for which they were designed. However, the circuit cannot be altered after fabrication. This forces a redesign and refabrication of the chip if any part of its circuit requires modification. This is an expensive process, especially when one considers the difficulties of replacing ASICs in a large number of deployed systems. Board-level circuits are also somewhat inflexible, frequently requiring a board redesign and replacement in the event of changes to the application.

The second method is to use software-programmed microprocessors, a far more flexible solution. Processors execute a set of instructions to perform a computation. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, the downside of this flexibility is that the performance can suffer, if not in clock speed then in work rate, and is far below that of an ASIC. The processor must read each instruction from memory, decode its meaning, and only then execute it. This results in a high execution overhead for each individual operation.

Additionally, the set of instructions that may be used by a program is determined at the processor fabrication time. Any other operations that are to be implemented must be built out of existing instructions.

Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. Reconfigurable devices, including field-programmable gate arrays (FPGAs), contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, sometimes known as logic blocks, are connected using a set of routing resources that are also programmable. In this way, custom digital circuits can be mapped to the reconfigurable hardware by computing the logic functions of the circuit within the logic blocks, and using the configurable routing to connect the blocks together to form the necessary circuit.

Compression and cryptographic algorithms are expensive in terms of time when they are implemented in general purpose processors. When implementing compression algorithms, the search for redundancy implies a lot of complex operations that can not be implemented efficiently with the available instruction set of a general purpose processor. And when cryptographic algorithms are implemented, it is necessary to perform a high amount of mathematical operations between large numbers in a finite field. Again, general purpose processors do not have instructions to support these operations leading to inefficient implementations. For these reasons, a hardware solution is well suited to implement both kinds of algorithms, especially for real time data processing.

FPGAs and reconfigurable computing have shown to accelerate a variety of applications. Recent applications that have shown to exhibit significant speedups using reconfigurable hardware include: automatic target recognition, string pattern matching, transitive closure of dynamic graphs, data compression, and genetic algorithms for the traveling salesman problem. Specifically, in cryptographic applications, there are many criteria which lead to prefer programmable devices instead of an ASIC option [10], these are:

- **Algorithm Agility**: Switching of cryptographic algorithms during operation of the targeted application. Whereas algorithm agility is costly with traditional hardware, FPGAs can be reprogrammed on the fly.

- **Algorithm Upload**: Devices are upgraded with a new encryption algorithm because of different reasons, for example, algorithm was broken or a new standard was created.

- **Architecture Efficiency**: A hardware architecture can be much more efficient

if it is designed for an specific set of parameters, for example the key or the underlying finite field. The more specific an algorithm is implemented the more efficient it can become. FPGAs allow this type of design and optimization with an specific parameter set. Due to the nature of FPGAs, the application can be changed totally or partially.

- **Resource Efficiency**: The majority of security protocols, like IPSec and SSL, are hybrid protocols. Since the algorithms are not used simultaneously, the same FPGA device can be used for both through runtime reconfiguration.

- **Throughput**: General-purpose processors are not optimized for fast execution especially in the case of public-key algorithms. This is because they do not have instructions for modular arithmetic operations on long operands, which is necessary in those kinds of algorithms. Although typically slower than ASIC implementations, FPGA implementations have the potential of running substantially faster than software implementations.

The advantages of software implementations include easy of use, ease of upgrade, portability, low development costs and low unit prices and flexibility; a software implementation offers moderate speed, high power consumption compared to custom hardware, and only limited physical security, especially with respect to key storage. ASIC implementations show lower price per unit, can reach high speeds, and can have low power dissipation. Hardware implementations of cryptographic algorithms are more secure because they cannot be easily read or modified by an outside attacker as software implementations. The lack of flexibility of ASIC implementations with respect to the algorithms and parameter, leads to higher development costs and switching. Although reconfigurable hardware devices, such as FPGAs, seem to combine the advantages of software and hardware implementations, there are still many open questions regarding FPGAs as a module for security functions [10].

1.2.1 Field programmable gate array (FPGA)

Field programmable gate arrays are specific integrated circuits that can be programmed easily. The FPGA contains versatile functions, configurable interconnects and an input/output interface to adapt to the user specification. FPGAs allow rapid prototyping using custom logic structures, and are very popular for limited device production. Modern FPGA are extremely dense, with a complexity of several millions of gates which enable the emulation of very complex hardware such as parallel microprocessors, mixture of processor and signal processing. One key advantage of FPGAs is the possibility of reprogramming them, in order to create a completely different hardware by modifying

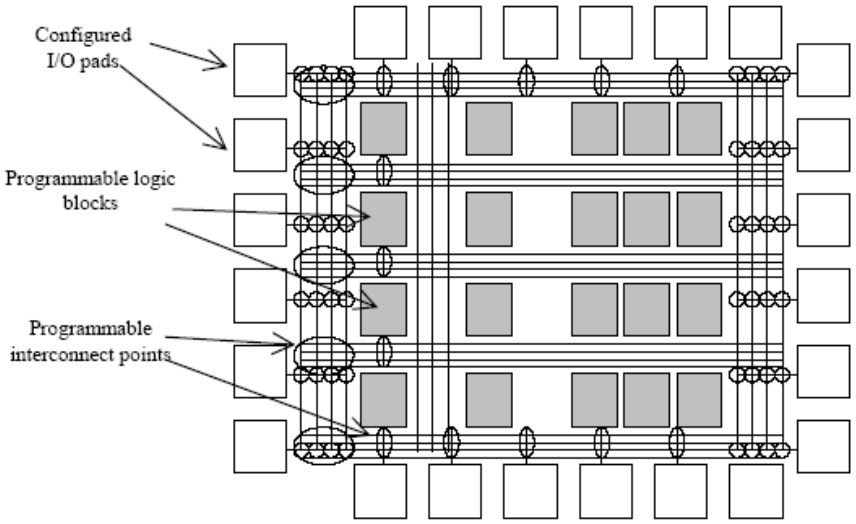

Figure 1.4: Internal structure of an FPGA

the logic gate array. The usual structure of FPGA is given in figure 1.4. FPGAs not only exist as simple components, but also as macro-blocks in system-on-chip designs.

Combinatorial logic implementation

The programmable logic blocks implement all basic logic functions, that are INV, AND, NAND, OR, NOR, XOR, XNOR. Several approaches are used in the FPGA industry to implement these basic gates. The first approach consists in the use of multiplexers, the second one in the use of look-up tables.

The look-up table (LUT) is by far the most versatile circuit to create a configurable logic function. The look-up table consist on n main inputs F_0, F_1,...F_n and a main output F_{out} that is a logical function of the n inputs. The value of F_{out} is defined by the values given in locations 0, $1 \ldots 2^n$, stored in a table or array. The input values F_0,F_1,...F_n create an n-bit address between 0 and 2^n, so that F_{out} gets the value of location in that address. For example, if the inputs create the number 5, the value stored at position 5 is routed to F_{out}.

Sequential logic implementation

Memory elements are essential components of the configurable logic blocks. The memory elements are used to store one logical value, corresponding to a logic truth table. For a n-input function (F_0,F_1,...F_n in the previous LUT), it is necessary an array of 2^n memory points to store the information of values 1...2^n. There exist several approaches to store one single bit of information. One of them consists of D-register cells. The

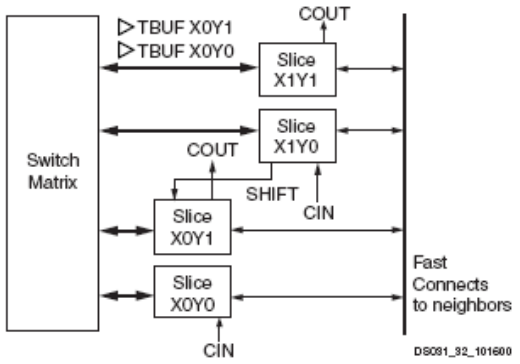

Figure 1.5: Internal structure of a typical configurable logic block of the Virtex II family

D-register cells are chained in order to limit the control signals to one clock and one data signal. The logical data i-value of a LUT is fully programmed by a word of 2^n bits sent in series to the signal input data of the *D*-register.

The programmable logic block structure of an FPGA varies according the fabricant. Often, it consists of a look-up table, a *D*-register and some multiplexers. The LUT and the *D*-register, may work independently or be mixed together. A typical configurable logic block in the Virtex II family is depicted in figure 1.5. Finally, configurable blocks are ordered in a bi-dimensional array interconnected by programmable points and switching matrix.

1.3 Description of the problem

To use a data compression algorithm together with an encryption algorithm, in the correct order, makes sense for three reasons:

- Compressing data before encryption reduces the redundancies that can be exploited by cryptoanalists to recover the original data.
- Compressing data speeds-up the encryption process.
- If encrypted data are transmitted over a computer network, the bandwidth is better utilized by compressing them before encryption.

It is important to underline the order of execution of the compression and encryption algorithms. If data were encrypted before they were compressed, it will lead to very poor or not compression because the result of the cryptographic operation are data with minimal redundancy.

The approach of combining compression with cryptography has been adopted in some software applications like HiFn [11], PKWare [12], PGP [3] and CyberFUSION [13]. Also, some network security protocols like SSL, SSH and IPSec compress data before encryption as part of the transferring process.

PGP (Pretty Good Privacy) provides the confidence and authentication services for sending e-mail and information storage. RSA is used for public key cryptography and CAST-128, IDEA y 3DES for symmetric encryption. The message to be sent is compressed, either after if it is signed or before if it is encrypted, using the ZIP algorithm. The 6.0 version of the compression program PKZip can encrypt data for storage or transfer. The 256-bit key AES algorithm is used for symmetric encryption and RSA for asymmetical one. In CyberFUSION, similar to an FPT application, data are encrypted using either the DES or 3DES algorithm. Compression is performed using the RLE (Run Length Encoding) algorithm.

HiFn has proposed a processor to perform both compression and encryption operations. Cryptographic algorithms supported by this processor are AES and 3DES for symmetrical encryption and SHA-1 and MD5 for authentication [3]; compression is performed by the LZS (Lempel-Ziv-Stac) [14] algorithm. CISCO offers some hardware and software modules to encrypt and compresses data that can be incorporated into routers in order to improve the performance of data transmission. Data can be compressed by the LZS algorithm or by the IPPCP compression protocol; the compressed data are encrypted by the AES algorithm with a 128, 192 or 256-bit key.

Although some effort has been done by some enterprises for combining compression and cryptography, asymmetrical encryption has not been considered. In addition, just software implementations have been considered. And as mentioned before, compression and cryptographic algorithms are expensive in terms of time when they are implemented in general purpose processors. When implementing compression algorithms, the search for redundancy implies a lot of complex operations that can not be implemented efficiently with the available instruction set of a general purpose processor. And when cryptographic algorithms are implemented, it is necessary to perform a high amount of mathematical operations with large numbers in a finite field. Again, general purpose processors do not have instructions to support these operations leading to inefficient implementations. For these reasons, a hardware solution is well suited to implement both kinds of algorithms, especially for real time data processing.

In this research work, a hardware architecture, combining lossless compression and asymmetric encryption is presented. The system architecture consists of a dictionary-based lossless data compressor that compresses the incoming data, and an elliptic curve cryptographic module that performs two EC (elliptic curve) cryptographic schemes: encryption and digital signature. The lossless compressor was selected because of its

relatively low requirements in area and average compression ratio. This kind of compressors performs well in universal data and it is faster to compress if compared with other lossless compressors. The asymmetrical cryptosystem ECC was selected because of the advantages it presents with respect to other public-key schemes: it uses shorter length keys while providing the same security level. The use of shorter keys implies less space for key storage and saving of time when the keys are transmitted. Also, arithmetic operations are computed faster when operands are reduce in size. ECC bases its security on a harder mathematical problem which makes it be more secure than traditionally schemes, for example RSA.

1.4 Motivation

The combination of two of the most important technologies in network applications, compression and encryption, has not been exploited extensively. Elliptic Curve Cryptography (ECC) is a novel approach that is beginning to be used in practical applications as an alternative to RSA, the most widely used public key cryptosystem. ECC offers the same security level than RSA using shorter length keys, up to six times. This fact leads to require less storage memory for keys and improves the execution of the arithmetic operations. These characteristics of ECC make it attractive to be implemented in constrained devices, for example, in wireless devices.

The main motivation to implement the complete system in hardware is the potential for higher execution speed. In both, compression and encryption, a lot of complex operations are required, so they run inefficiently in general purpose processors that do not count with specialized instructions to support that kind of operations. The speed-up is obtained by designing custom architectures executing the most time consuming parts in each of the algorithms.

Although it seems to be a high and complex system joining two algorithms, with the available technology, FPGAs and high level hardware description languages, it seems to be possible the realization of such system in a relative short period of time. That is what this work intends to achieve in order to demostrate the premisses given in the previous section. Encrypting compressed data gathers the advantages of compression and encryption: data is transmitted in a secure way reducing space and time costs for storage, and the time to encrypt data.

1.5 Thesis objectives

1.5.1 General objective

The general objective of this research work is to demonstrate the theoretical advantages of combining lossless data compression and public key cryptography. To achieve this objective, a hardware architecture for dictionary-based data compression and elliptic curve cryptography that meets real time requirements is implemented. Unlike traditional approaches, this work considers public-key cryptography instead of symmetrical one.

1.5.2 Specific Objectives

Specific objective is to validate the performance of elliptic curve cryptography by adding data compression. On this specific objective, it is desired to prove the following statements:

1. Compression can improve the performance in the cryptographic module, how occurs in symmetrical methods.

2. The available bandwidth is better utilized when encrypted data are transmitted.

3. It is possible to combine two different algorithms in a single hardware architecture.

1.6 Methodology

To meet real time requirements, the design and implementation of the system is carried out by applying parallelism at algorithmic level. Although parallelism at the level of design can be applied, it was decided not to apply it or minimally. That is because it is preferable a generic architecture that can be implemented in any programmable device. When such device is selected, optimizations for that device can be done. However, with current sophisticated synthesis tools, such optimizations often are transparent to the user.

The searching for redundancy in data compression is implemented using a systolic array approach. Systolic arrays achieve high clock rates while using low area resources and provide better testability. In the cipher, the arithmetic operations are implemented as parallel as possible, identifying data dependences and utilizing custom datapaths and specialized architectures.

For validating the hardware system, a software version of the proposed system was developed for results comparison and test benches generation. Both, the compressor

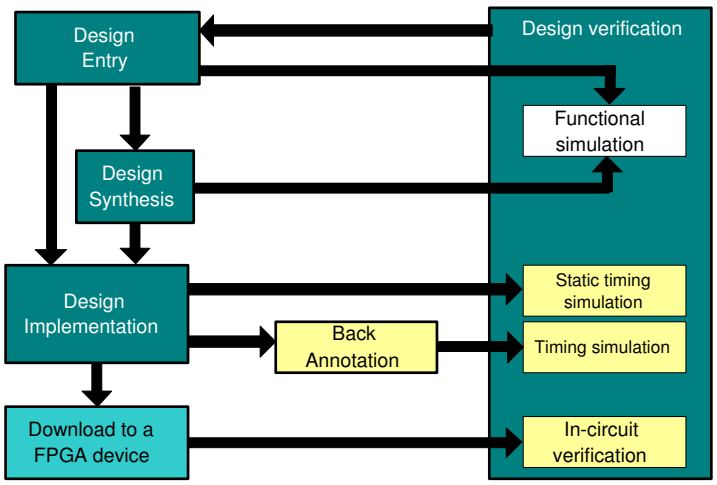

Figure 1.6: Flow design of the system

and the ECC cipher are written in C, letting to migrate the code to any platforms with very few or no changes at all.

The hardware architecture is described in the standard hardware description language VHDL, making possible any rapid further modification. Hardware modules are described as parameterelizable as possible, leading to a scalable implementation. The hardware architecture was simulated on Active-HDL software and synthesized for the Virtex II FPGAS family using ISE software. The design flow of the system was carried out following the flow design of an FPGA system, shown in figure 1.6.

1.7 Overview of the thesis

The thesis is organized as follows: The next chapter presents the background for the lossless data compressor and the mathematical background and cryptographic schemes for elliptic curve cryptography. Chapter 3 presents the architecture design of the system explainig its internal modules in detail. Chapter 4 shows the synthesis results and a results comparison against reported work. Finally, Chapter 5 summarizes this work and the contributions of this thesis. Conclusions and directions are presented at the end of Chapter 5.

Chapter 2

LZ compression and Elliptic Curve Cryptography

In this chapter, a discussion about the selection of the data compression algorithm, a variant of the original LZ77 algorithm, is presented. Also, this chapter provides the mathematical background of elliptic curve cryptography and additional algorithms used in the elliptic curve cryptographic schemes treated in this work.

2.1 The data compression module

The main system is focused to encryption of text, so *lossless compression* is selected as a compressor module in the system. In all lossless compression implementations, there is a trade-off between computational resources and compression ratio. Often, in both statistical and dictionary-based methods, the best compression ratios are obtained at expenses of long execution time, high memory and logic gates. Statistical compressors are characterized for consuming higher resources than dictionary based when they are implemented in both software and hardware, however they can achieve compression ratios near to the source entropy. The most demanding task in this kind of algorithms is the implementation of the model to get the statistics of the symbols and to assign the bit string. Perhaps, the most representative statistical method is the proposed by David Huffman [15] in 1952. In this algorithm a tree is built according to the frequency of the symbols. All symbols are placed at the leaves of the tree. The Huffman method achieves compression by replacing every symbol by a variable bit string. The bit string assigned to every symbol is determined by visiting every internal node from the root up to the leaf corresponding to the symbol. Initially the bit string is the null string. For every internal node visited, one bit is concatenated to the bit string, 1 or 0, depending on the current visited node whether it is a right or left child of its father. Symbols at

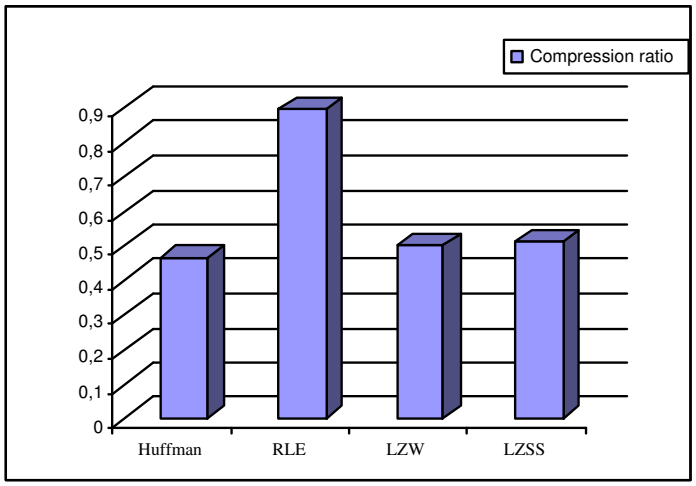

Figure 2.1: Average compression ratio for some lossless data compressors

longer branches will be assigned larger bit strings.

In the dictionary-based methods, the most time consuming task is searching for strings in a dictionary, which usually has hundreds of locations. Dictionary-based algorithms are considered simpler to implement than statistical ones but the compression ratio is poorer.

2.1.1 Selecting the algorithm

Selecting a compression method among the existent ones is not easy. While one method can be faster, other can achieve better compression ratio and still other may require less computational resources. In terms of execution time, the implementations of compression methods are being improved continually so it is difficult to give definitive results. Hardware and software implementations of lossless methods have been reported in the literature. The main aspects to consider in such implementations are the compression ratio achieved, throughput, and processing time.

In figure 2.1, the graph shows the compression ratio of some lossless data compressors implemented in software, compressing the Canterbury Corpus. Although the Huffman algorithm achieves better compression ratio than RLE, the difference in compression ratio compared with LZW and LZSS (as slight modification of LZ77) is not significant.

Lossless data compression hardware implementations have been also reported, statistical and dictionary based. Similar to software implementations, statistical compressors have shown to be more area expensive than dictionary-based implementations. In figure 2.2, aspects like frequency, throughput and area reported in those implementations are shown. In [16], Park y Prassana proposed a hardware architecture for the Huffman

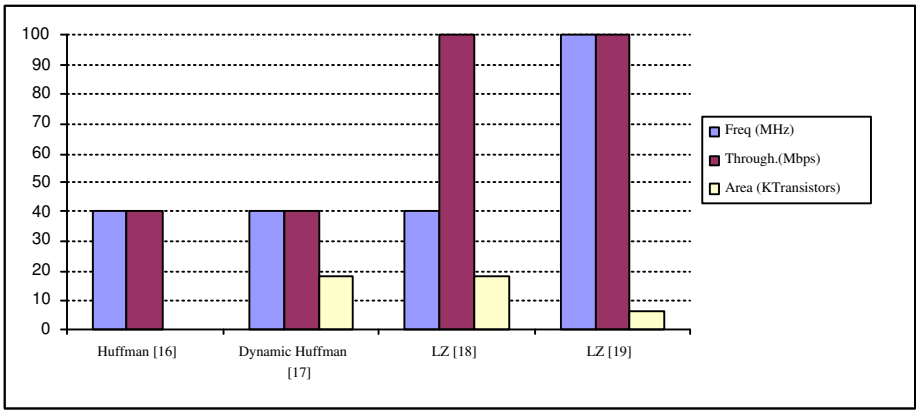

Figure 2.2: Some lossless hardware implementations

algorithm. They proposed an efficient scheme for mapping the Huffman's tree to the memory. Two main memories were used; one of 256x9 bits for coding and one of 64x18 bits for decoding, also, an arithmetic unit with two adders and one comparator was required. The design was implemented using 1.2 microns CMOSN technology. Liu *et al.* [17], implemented the dynamic Huffman algorithm. This method is faster than the static one because the Huffman's tree is built as symbols are processed, avoiding a second pass over the data. The implementation is based on the concept of CAM (Content Addressable Memory) memories. This method is more complex and requires more hardware resources.

Representative reported hardware implementations of dictionary-based methods are one by Ranganatan and Enriques [18] and other by Jung [19]. These jobs and others will be discussed with more detail in section 2.1.3.

Based on the appointment states in software and hardware implementation, it leads to think that a dictionary-based method is the better choice for the lossless compressor module required in the proposed cryptosystem in this work. It is necessary a compressor that does not require high hardware resources and at the same time performs well with all kind of data.

A dictionary-based compressor is considered for implementation in this work. This selections was taken according to the following facts: 1) a dictionary method does not require prior knowledge or statistical characteristics of the symbols, so, avoiding the second pass lets faster compression. 2) it is stated in the literature that the memory for dictionary-based methods is less than that for statistical methods.

Among dictionary methods, the original LZ algorithm (LZ77) has some advantages over other methods. The memory requirements are lower and the compression process is simpler [2]. Also, the decompression process is simpler than the compression one,

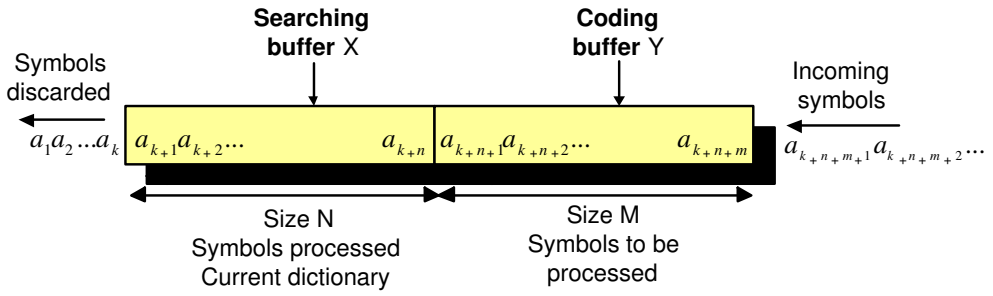

Figure 2.3: Buffers for LZ data compression

practically it consists in memory access while in the other methods compression and decompression have similar complexity. In addition, LZ77 is free of royalties, that does not occur with LZW, for example.

In the following section, the LZ algorithm and its implementation issues are discussed.

2.1.2 The LZ algorithm

The LZ77 algorithm was proposed by Ziv and Lempel in 1977 [5]. It is a universal dictionary-based algorithm for lossless data compression; it does not depend on the type of data being compressed. LZ77 algorithm was the first proposal of data compression based on a dictionary instead of symbols's statistics. Since its proposal, this algorithm has been improved in order to achieve better compression ratios and to reduce the required processing time [6], [20] and [21]. The idea behind the LZ77 algorithm is to replace a symbol string by a pointer or position in a dictionary where such strings occur. The algorithm constructs the dictionary as symbols are processed, the bigger the size of the dictionary, the higher the possibility of finding future occurrences of the strings. One modification of the original LZ77 method, is implemented in this work. It is known as the LZSS method and prevents expansion of data instead of reduction.

The LZ77 algorithm uses two buffers called the *searching-buffer*, containing N symbols, and the *coding-buffer*, containing M symbols, see figure 2.3.

These two buffers can be viewed as a $N + M$ sliding window over the source data. Data compression is achieved by performing two steps. The first step consists in inserting new source symbols to the coding buffer. For every new entered symbol, the current content of both, the searching and the coding buffer is shifted to the left one position. The first step is depicted in figure 2.4 b). Initially, M symbols are entered because both the searching and the coding buffers are empty. In the second step, the longest substring in the searching buffer being the prefix in the coding buffer is searched

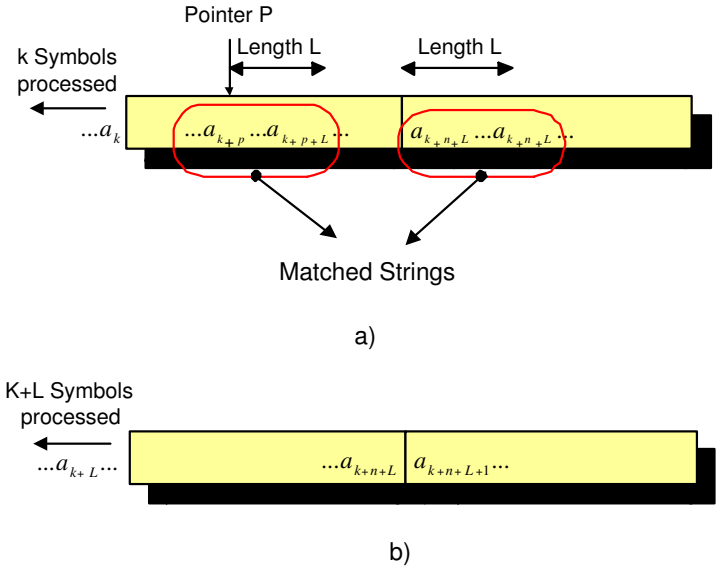

Figure 2.4: Step 1 a) and 2 b) in the LZ77 algorithm

for. After testing all possible string matches, a *codeword* is generated consisting of the pointer P, or position in the searching buffer where the string starts, and its length L. Step two is depicted in figure 2.4 a). The process restarts by executing the step one again, entering L new symbols to the coding buffer.

There is no a data compressor that compress all kind of data. For preventing data expansion, Storer and Yamasaki [21] proposed to codify the codeword found only if the length L was greater than a threshold value. If it does not, symbols in the coding buffer should be written without codification. The threshold should be one that avoids a codeword to be greater than the length of the current substring processed. So, applying this mechanism, the output stream consists of compressed and uncompressed symbols. In order to decode the compressed data, one extra bit is used to distinguish among codewords and source symbols.

The serial algorithm for performing the step two in the LZ77 method is listed in algorithm 1. X and Y are the searching and coding buffers respectively. In algorithm 1, N is the size of X and M is the one of Y. Searching for all possible substrings sequentially is a problem of $O(MN)$ complexity, and the most time consuming part of the algorithm. The execution time and compression ratio of LZ77 algorithm depends strongly on the size of the buffers X and Y.

Algorithm 1 Step two in the LZ algorithm

Require: X and Y two buffers with valid data
Ensure: A codeword $C = P$, L

```
1: maxLength ⇐ 0
2: for i from 1 to N do do
3:    currentLength ⇐ 0
4:    for j from 1 to M do do
5:       if X[i + j] = Y[j] then
6:          currentLength ← currentLength + 1
7:       else
8:          break
9:       end if
10:   end for
11:   if currentLength > maxLength then
12:      maxLength ← currentLength
13:      Pointer ← i
14:   end if
15:   if maxLength = M then
16:      break
17:   end if
18: end for
```

2.1.3 Related work on LZ77 hardware implementations

Three approaches are distinguished in the hardware implementations of dictionary based methods: the microprocessor approach, CAM (Content Addressable Memory) approach and systolic array approach [22]. Although the first approach lets easy implementations, it does not explore full parallelism and is not attractive for real time applications. The second one is very fast because explore full parallelism but it is costly in terms of hardware requirements. The systolic array approach is not as fast as the CAM approach but its hardware requirements are lower and testability is better. The main advantage of this approach is that it can achieve a higher clock rate and it is easier to implement than the CAM approach.

Some papers have reported systolic implementations for the LZ77 algorithm. In these papers, the systolic array is derived by studying the data dependences in the computations. In [19], the systolic array is composed of two different types of processing elements designed in such way that each one consumes few resources. The number of type 1 PE's is only determined by the size of the coding buffer. This design was implemented on an ASIC platform running at 100 MHz. They reported a throughput of 10-100 Mbps using a 1.5 kB FIFO for the dictionary; using one 4.1K SRAM for the dictionary and 32 processing elements. The reported throughput of the design was 100 Mbps operating at a clock rate of 100 MHz. In [23], an FPGA LZ77 implementation

is reported. The implementation requires four Xilinx 4036XLA FPGAs to achieve 100 Mbps throughput. The buffer's size was 512 for the searching buffer and 63 for the coding one. In [22], a VLSI chip is fabricated. It contains 16 processing elements to find the longest substring in the LZ77 algorithm. The chip operates at 100 MHz and achieves a throughput of 10 Mbps if a 1K-searching buffer and a 16-coding buffer are used. As mentioned in the paper, if ten chips are connected in parallel, a compression rate of about 100Mbps can be achieved. These reported papers give us an idea of what can be expected from a hardware implementation but does not give us a guide to select the best parameters (size of buffers) for the algorithm according to a specific application. In chapter 4, the results and comments about a software implementation show how the size of the buffers impacts the compressor performance.

Since this thesis work uses elliptic curve cryptography as the public-key cryptosystem, in the following sections, aspect related to public key cryptography and specifically, aspects related to elliptic curve in cryptography are presented.

2.2 Foundations of public-key cryptography

Almost all public-key algorithms are based on hard mathematical problems, that are defined on a set of numbers. In cryptography, this set is finite and it has to meet several properties.

2.2.1 Groups and Finite Fields

Groups and fields are part of abstract algebra, a branch of mathematics. Finite fields are increasingly important in several areas of mathematics, including linear and abstract algebra, number theory and algebraic geometry, as well as in computer science, statistics, information theory, and engineering. Also, many cryptographic algorithms perform arithmetic operations over these fields. [3].

A *group* Γ is an algebraic system $\{S, \diamond\}$ consisting of a set S and a binary operation $\diamond$ defined on S that satisfies the following axioms:

1. Closure: $\forall\ x,\ y \in G,\ x \diamond y \in G$.
2. Associativity: $\forall\ x,\ y,\ z \in G,\ (x \diamond y) \diamond z = x \diamond (y \diamond z)$.
3. Identity: $\exists\ I \in G$ such as $x \diamond I = x$.
4. Inverse: $\forall\ x \in G$, exist only one $y \in G$ such as $x \diamond y = y \diamond x = I$.

If $\forall\ x,\ y \in G,\ x \diamond y = y \diamond x$, Γ is called an *abelian* group.

A *finite field* is an algebraic system $\{F, \oplus, \odot\}$ consisting of a set F containing a fixed number of elements and two binary operations, $\oplus$ and $\odot$ on F, satisfying the following axioms:

1. Elements 0 and 1 $\in F$.
2. F is an abelian group respect to operation $\oplus$.
3. F - $\{0\}$ is an abelian group respect to operation $\odot$.
4. $\forall\ x, y, z \in F$, $x \odot (y \oplus z) = (x \odot y) \oplus (x \odot z)$ and $x \oplus (y \odot z) = (x \oplus y) \odot (x \oplus z)$.

The order of a finite field is the number of elements in that field. It has been showed [8] that exists a finite field of order q if and only if q is a prime power. In addition, if q is a prime power, the finite field of order q is unique. A finite field, also known as *Galois Field*, is denoted as F_q or $GF(q)$.

2.2.2 Modular arithmetic

The operation,

$$a \bmod n = z \tag{2.1}$$

means that z is the remainder when a is divided by n, the remainder is an integer in the range $[0, n-1]$. This operation is called modular reduction, and it is used in cryptographic schemes mainly for two reasons: 1) operations like logarithms and square roots module n are hard problems and 2) the space of values is restricted to a fixed group of numbers. In cryptography applications, a, z and n are large integer numbers. Another common notation for equation 2.1 is to say that a and z are equivalent or a is congruent to $z \mod n$, which is written as

$$a \equiv z \pmod{n} \tag{2.2}$$

Modular arithmetic is commutative, associative and distributive. The common integer operations +, *, and - in modular arithmetic are defined as follows:

1. $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
2. $(a - b) \bmod n = ((a \bmod n) - (b \bmod n)) \bmod n$
3. $(a * b) \bmod n = ((a \bmod n) * (b \bmod n)) \bmod n$
4. $a * (b + c) \bmod n = (((a * b) \bmod n) + ((a * c) \bmod n)) \bmod n$

Another important modular operation is the *inversion*. a^{-1} is the inverse mod n of a number a if equivalence in equation 2.3 is true.

$$a * a^{-1} \equiv 1 \pmod{n} \tag{2.3}$$

For a given number a, a^{-1} is the unique solution only if a and n are *relatively primes* [4]. If n is a prime number, every number in the range $[1, n-1]$ is relatively prime to n and has exactly one inverse $\pmod{n}$. To calculate a module inverse, two algorithms are commonly used: The Extended Euclidean Algorithm and the Fermat's Little Theorem. These algorithms are described in next chapter.

2.2.3 Prime and binary finite field

The *prime finite field* has been long used in cryptography. It is denoted as F_p, where p is a prime number. F_p consists of the elements $\{0, 1, 2, \ldots, p-1\}$. The operations $\oplus$ and $\odot$ are performed as the ordinary integer operations sum and multiplication respectively applying reduction $\pmod{p}$. These operations are defined as follows:

1. $\oplus$: $(a + b) \bmod p$, a,$b \in F_p$,
2. $\odot$: $(a * b) \bmod p$, a,$b \in F_p$,
3. $\forall\ a \neq 0 \in F_p$, a^{-1} is the inverse of a if $a * a^{-1} = 1 \pmod{p}$.

The *binary finite field* is denoted as F_{2^m} (also known as *two-characteristic field* or the *Galois field*) and can be viewed as a m-dimension vectors space on $\{0, 1\}$. As a vectorial space, a basis exist in F_{2^m}. The set $\{\alpha_0, \alpha_1, ..., \alpha_{m-1}\}$, $\alpha_i \in F_{2^m}$, is called a basis of F_{2^m} if exist $m-1$ elements a_i in $\{0, 1\}$ such as every element $a \in F_{2^m}$ can be expressed as in equation 2.4.

$$a = a_{m-1}\alpha_{m-1} + a_{m-2}\alpha_{m-2} + ... + a_1\alpha_1 + a_0\alpha_0 \tag{2.4}$$

If such basis exist, each element in F_{2^m} can be represented as the binary m-vector $(a_0, a_1, a_2, \ldots, a_{m-1})$. There are different basis for F_{2^m}, most common used are: *polynomial*, *normal* and *dual*. The arithmetic for binary operations $\oplus$ and $\odot$ changes lightly according to the basis employed. When implementing binary field arithmetic, polynomial basis is preferred because of the sum of two elements is a simple xor operation. Details about normal basis can be found in [8].

In polynomial basis, each m-vector $a \in F_{2^m}$, is viewed as a polynomial,

$$a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0 \tag{2.5}$$

The binary field F_{2^m} is generated by an *irreducible polynomial* $F(x)$ of grade m of the form

$$x^m + f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + ... + f_1x + f_0 \tag{2.6}$$

where $f_i \in \{0, 1\}$. The polynomial is named irreducible because it can not be expressed as the multiplication of two other polynomials (it is as a prime number in integer arithmetic). The polynomial needs to be irreducible, otherwise the math does not work [4].

Let be $a,\ b \in F_{2^m},\ a = (a_{m-1}, a_{m-2}, ..., a_1, a_0)\ b = (b_{m-1}, b_{m-2}, ..., b_1, b_0)$. The arithmetic operations $\oplus$ and $\odot$ in the finite field F_{2^m} are defined as follows:

1. $\oplus$: $a \oplus b = c$, $c = (c_{m-1}, c_{m-2}, ..., c_1, c_0)$, where $c_i = a_i$ xor b_i.

2. $\odot$: $a \odot b = c$, $c = (c_{m-1}, c_{m-2}, ..., c_1, c_0)$, where $c(x) = a(x)b(x) \bmod F(x)$.
   $c(x) = c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + ... + c_1x + c_0$
   $a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0$
   $b(x) = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + ... + b_1x + b_0$

For cryptographic applications, the irreducible polynomial $F(x)$ is a trinomial of the form $x^m + x^i + 1$ or it is pentanomial of the form $x^m + x^i + x^j + x^l + 1$, where i, j and l are positive integers. The use of trinomials or pentanomials leads to efficient software and hardware implementations.

2.2.4 Security of public key cryptosystems

Different to symmetrical cryptography, where data transformation is carried out by permutations and transpositions, in public key cryptography, almost all methods transform data by executing complex arithmetic operations on a finite field. This is because the security of these cryptosystems is based on conjectured very difficult problems, so, in order to break the cryptosystem, it is necessary to solve the underlying problem. The more difficult the problem on which one cryptosystem is based on, the higher the security offered by the cryptosystem. In practice, two problems have been taken to build on them public key cryptosystems: the large integer factorization problem and the logarithm discrete computation. The complexity of these two problems is a function of the order of the finite field used. While cryptosystems based on integer factorization are considered currently secure using 1024 bit operands, the cryptosystems based on logarithm discrete problems offer an equivalent security level using 160-bit operands.

The logarithm discrete problem is defined on a finite group $\Gamma = \{S, \diamond\}$. For a group element $g \in S$ and a number n, let g^n denote the element obtained by applying operation

$\diamond$ n times to g ($g^2 = g \diamond g$, $g^3 = g \diamond g \diamond g$, and so on). The discrete logarithm problem is as follows: given an element $g \in S$ and another element $h \in S$, find an integer x such that $g^x = h$.

The most widely public key cryptosystem, the RSA, is based on the integer factorization problem. Other public key cryptosystems like ElGammal, Diffie-Hellman key exchange and DSA are based on the logarithm discrete problem defined on the multiplicative group modulo p, p is a large prime number.

For both, the integer factorization and the logarithm discrete problems exist algorithms for solving them running in sub-exponential time. This means that these problems are considered hard, but not as hard as those problems which admit only fully exponential time algorithms. In the following section, another hard mathematical problem is presented: the discrete logarithm problem defined on the group additive of elliptic curve points.

2.3 Elliptic Curve Cryptography (ECC)

Elliptic curves, as geometric algebraic entities have been studied since the second half of nineteen century, initially, without any cryptographic purpose. In 1985, application of elliptic curves in public key cryptography was independently proposed by Neals Koblitz [24] and Victor Miller [25].

Koblitz and Miller proposed to use an elliptic curve (see definition in next section) defined on a finite field, and to define a point addition operation such that the points of the elliptic curve and the point addition operation formed an abelian group. On this group, the discrete logarithm problem, called the elliptic curve discrete logarithm problem (ECLDP), can be defined and so, a cryptosystem could be built on this problem. The advantages of this elliptic curve cryptosystem is that the ECLDP is more difficult to solve than that defined on the multiplicative group F_p. The best algorithm known for solving the ECDLP is fully exponential, the Pollar-Rho method [26].

ECC can offer a similar security level than other public key criptosystems using shorter length keys, which implies less space for key storage, time saving when keys are transmitted and modular computations less costly. ECC's security has not been proved; its strength is based on the inability to find attacks.

2.3.1 The elliptic curve group

An elliptic curve is a set of points (x, y) that satisfies an equation. Although the curve can be defined on any domain, for cryptographic applications, it must be defined on a

finite field F_q. If it is defined on F_p the elliptic curve equation is

$$y^2 = x^3 + ax + b \tag{2.7}$$

When the elliptic curve is defined on the binary field, the equation is

$$y^2 + xy = x^3 + ax^2 + b \tag{2.8}$$

In both cases, a, b are in the finite field. Because all arithmetic is performed as defined in the underlying finite field, x, y are finite field elements too. Let $EC(F_q)$ denote the points of the elliptic curve defined on the finite field F_q. Because a F_q is finite, $EC(F_q)$ is also a finite set. In order to build a group from $EC(F_q)$, a closure operation on $EC(F_q)$ must be defined. The operation is the sum of points and its definition has a geometrical interpretation. The sum can involve equal or different points. For equal points, the sum is known as DOUBLE and for different points, it is known as ADD. Each of these operations is carried out in different way and its definition depends on the finite field on which the elliptic curve is defined.

In the case of the finite field F_p, ADD and DOUBLE operations are defined as follows:

Let a, $b \in F_p$ satisfying $4a^3 + 27b^2 \neq 0$. Let $P, Q, R \in E(F_p)$, $P = (x_1, y_1)$ $Q = (x_2, y_2)$ $R = (x_3, y_3)$. The system $\{E(F_p) \cup \{O\}, +\}$ forms an abelian group respect to $+$ operation, defined as

1. $P + O = O + P = P$, $\forall P \in E(F_p)$
2. If $P \neq \pm Q$, ADD(P, Q) $= P + Q = R$, where
   $x_3 = \lambda^2 - x_1 - x_2$
   $y_3 = \lambda(x_1 - x_3) - y_1$
   $\lambda = (y_2 - y_1)/(x_2 - x_1)$
3. If $P = Q$, DOUBLE(P) $= P + Q = 2P = R$, where
   $x_3 = \lambda^2 - 2x_1$
   $y_3 = \lambda(x_1 - x_3) - y_1$
   $\lambda = (3x_1^2 + a)/2y_1$

So, an ADD operation requires one inversion, two multiplications, one squaring operation and six sums, all of them operation on F_p. A DOUBLE operation requires one extra squaring and two extra sums.

For the case of the F_{2^m} finite field, let a, $b \in F_{2^m}$. Let $P, Q, R \in E(F_p)$, $P = (x_1, y_1)$ $Q = (x_2, y_2)$ $R = (x_3, y_3)$. The system $\{E(F_{2^m}) \cup \{O\}, +\}$ forms an abelian group respect to $+$ operation, defined as

1. $P + O = O + P = P$, $\forall P \in E(F_{2^m})$

2. If $P \neq \pm Q$, ADD(P, Q) $= P + Q = R$, where
   $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$
   $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$
   $\lambda = (y_2 + y_1)/(x_2 + x_1)$

3. If $P = Q$, DOUBLE(P) $= P + Q = 2P = R$, where
   $x_3 = \lambda^2 + \lambda + a$
   $y_3 = x_1^2 + \lambda x_3 + x_3$
   $\lambda = x_1 + y_1/x_1$

In this case, ADD requires one inversion, two multiplications, one squaring and eight additions. For DOUBLE operation, it requires five additions, two squaring, two multiplications and one inversion, all of them, operations on F_{2^m}.

Either if the elliptic curve is defined on the prime F_p or binary F_{2^m} finite fields, the sum of points is the group operation where the ECDLP is defined. The ECDLP consist on given two points P, $Q \in E(F_q)$, to find the positive integer k such as $Q = kP$. As commented in section 2.2.4, this problem is of exponential complexity. On the contrary, knowing the scalar k and the point P, the operation kP is relative easy to compute. The operation kP is the main and most time consuming operation in all cryptographic schemes built on elliptic curves. The operation kP is computed as accumulative sum operation of point P with itself. These sum operations can be either ADD or DOUBLE operations, the efficient computation of these operations impacts the execution time of the scalar multiplications and also, the execution of the cryptographic schemes. It has been shown in the literature that the finite field used do no affect the security of the elliptic curve cryptosystems but their performance. It is considered that elliptic curves defined on the binary field F_{2^m} leads to efficient implementation of the field arithmetic. There exist curves on which the ECDLP is easy to solve [8], the so-called *supersingular* curves have to be avoided. These curves have an equal number of points that the finite field on which they are defined. The National Institute of Standards and Technology (NIST) has emitted several recommendations and has proposed several elliptic curves over both F_p and F_{2^m} for cryptographic applications. Different to other cryptosystems, the security of ECC not only depends on the length of the key but also in other parameters like the elliptic curve being used.

2.3.2 Cryptographic schemes

An elliptic curve cryptosystem consist in a tuple. If the elliptic curve is defined on F_p, the cryptosystem domain parameters are the six-tuple $D_{F_p} = (p, a, b, G, n, h)$, p is the

a big prime number, a and b define the elliptic curve on F_p, G is a generator point of $EC(F_p)$, n is the order of G, that is, the smaller integer such that $nG = \emptyset$ (infinity point). h is called the co-factor and is equal to $\#EC(F_p/n)$, this value is optionally used in some cryptographic schemes based on elliptic curves.

If the curve is defined on the binary field F_{2^m}, the domain parameters consist on a seven-tuple $D_{F_{2^m}} = (m, F(x), a, b, G, n, h)$. m defines the order of the finite field. $F(x)$ is the irreducible required in the field arithmetic; all other values have similar definition that in the case of F_p.

In both cases, the domain parameters specify an elliptic curve, a generator point and the order of this point. How to generate such domain parameters is not concern of this thesis neither the validation process. There are proposed methods for both operations that can be reviewed in [8], [27] and [28].

The algorithms implemented in this thesis are: the Elliptic Curve Digital Signature Algorithm (ECDSA) and the Elliptic Curve Integrated Encryption Scheme (ECIES). The ECDSA is the elliptic curve analogue of DSA for digital signatures. It has been standardized by several international organizations like ISO, IEEE and ANSI. In this thesis, the specification of ECDSA in the ASI X9.62 document [28] is referred.

The ECIES scheme, for public-key bulk encryption is the most promising scheme to be standardized. It has been considered in some drafts and it is currently recommended by the Standards for Efficient Cryptography Group (SECG) [27]. In this thesis, this specification of ECIES is referred.

ECDSA and ECIES are listed in next section. In both, it is supossed that two entities A and B share either the domain parameters $D_{F_{2^m}}$ or D_{F_p}. It is also supossed that $\{d_A, Q_A\}$ are the private and public keys for entity A and that $\{d_B, B\}$ are the ones for entity B.

ECIES scheme

The ECIES scheme employs two algorithms: a symmetric cipher E and a MAC (Message Authentication Code) algorithm. The keys for the symmetrical and the MAC algorithms are generated from a secret shared value between A and B by a key derivation function (KDF). Assume that $k_{Slength}$ is the key's length for the symmetrical algorithm and $k_{MAClength}$ is the one for the MAC algorithm. A sends an encrypted message D to B executing the following steps:

1. Select a random number k from $[1, n-1]$
2. Compute $(x, y) = kQ_B$ and $R = kG$
3. Derive a $(S + M)$-bit key k_{KDF} from x according to [27].

4. Derive a S-bit key k_S from K_{KDF} and encrypt the message. $C = E(m, k_S)$

5. Derive a M-bit key k_M from K_{KDF} and compute the m's MAC value. $V = MAC(m, k_M)$

6. Send (R, C, V) to B

To recover the original message, B does the following:

1. If R is not a valid elliptic curve point, fail and return.

2. Compute $(x', y') = d_B R$

3. Derive a $(S + M)$-bit key k_{KDF} from x' according to [27].

4. Derive a S-bit key k_S from K_{KDF} and decrypt the message C. $m_1 = E(C, k_s)$

5. Derive a M-bit key k_M from K_{KDF} and compute the m_1's MAC value. $V_1 = MAC(m_1, k_M)$

6. Accept message m_1 as valid if and only if $V = V_1$

SEC-1 [27] mentioned only two MAC algorithms supported, HMAC-SHA-1-160 and HMAC-SHA-1-80. In the firt case the key used is 160-bits long and produces an 80-bit or 160-bit output. The second case is different only in that the key is 80-bits long. Both MAC schemes are described in ANSI X9.71 based on the hash function SHA-1 described in FIPS 180-1 [29]. For symmetrical encryption, SEC mentioned that at this moment, only two symmetrical algorithms can be used: 3-Key TDES in CBC [28] mode and XOR encryption. The XOR encryption scheme is the simplest encryption scheme in which encryption consists of XORing the key and the message, and decryption consists of XORing the key and the ciphertext to recover the message. The XOR scheme is commonly used either with truly random keys when it is known as the 'one-time pad', or with pseudorandom keys as a component in the construction of stream ciphers. The XOR encryption scheme uses keys which are of the same length as the message to be encrypted or the ciphertext to be decrypted. 3-key TDES in CBC mode is designed to provide semantic security in the presence of adversaries launching chosen-plaintext and chosen-ciphertext attacks. The XOR encryption scheme is designed to provide semantic security when used to encrypt a single message in the presence of adversaries capable of launching only passive attacks. The KDF key derivation function only supported at this moment in ECIES is defined in ANSI X9.63.

ECDSA scheme

To sign a message D, entity A does the following:

1. Select a random number k from $[1, n-1]$
2. Compute $R = kG = (x, y)$ and $r = x \bmod n$. If $r = 0$ go to step 1.
3. Compute $s = k^{-1}(H(D) + d_A r) \bmod n$, H is the hash value of the message.
4. The digital signature on message D is the pair (r, s)

Entity B can verify the digital signature (r, s) on D performing the following steps:

1. Verify r and s are integers in $[1, n-1]$, if not, the digital signature is wrong. Finish and reject the message.
2. Compute $w = s^{-1} \bmod n$ and $H(D)$, H is the hash value of the message.
3. Compute $u_1 = H(D)w \bmod n$ and $u_2 = rw \bmod n$
4. Calculate $R' = u_1 G + u_2 Q_A = (x', y')$
5. Compute $v' = x' \bmod n$, accept the digital signature if and only if $v' = r$

The ECDSA scheme requires an elliptic point addition, scalar multiplications, modular arithmetic and the hash value of the message. The hash function recommended in ANSI X9.62 is SHA-1.

2.3.3 ECC related work

All the reported hardware architectures for elliptic curve cryptography are related with the field and elliptic arithmetic required, and none of them seems to implement an ECC algorithm such as digital signature. Although software implementations of elliptic curve schemes have been reported, arithmetic operations are slower than hardware solutions and are unattractive if the cryptosystem needs to perform many cryptographic operations, for example, in a secure web server. Next section presents a brief review of the related work both in hardware and software.

Hardware implementations

Hardware architectures for ECC can be divided into processor or coprocessor approaches. In the former, there exist a number of specialized instructions the processor can support, some of them are for finite operations or elliptic curve points. In the latter, there

are no such instructions because the algorithms are implemented directly on specialized hardware.

Some processors reported for ECC are found in [30], [31], [32] and [33]. In [30], Orlando and Paar reported an FPGA-based processor while in [31] Satoh and Takano have reported an ASIC-based processor. In the work performed by Orlando and Paar, scalar multiplication can be performed in 0.21 ms, working over the binary field $F_{2^{167}}$ with polynomial basis representation and representing elliptic points in projective coordinates. The architecture exploits the benefits of reconfigurable computing and incorporates pipelining and concurrence techniques in order to have an optimized hardware that can support several elliptic curves and finite field orders. A prototype of this processor was implemented in a XCV400E-8BG432 Xilinx FPGA. The processor reported by Satoh and Takano in [31] can support both prime and binary finite field for arbitrary prime numbers and irreducible polynomials. This characteristic is achieved by introducing a dual field multiplier. Scalar multiplication can be performed in 0.19 ms in the binary field $F_{2^{160}}$.

Some coprocessors reported for ECC are found in [34], [35] and [36]. In [34] Okada *et al.* reported a coprocessor for ECC both in a FPGA device and in an ASIC platform. In the case of the FPGA implementation, scalar multiplication takes 80 ms for random curve over the finite field $F_{2^{163}}$. The critical part of the coprocessor is a bit-parallel multiplier that can operate with different irreducible polynomials. In [35] Ernest *et al.* reported a reconfigurable implementation of a coprocessor for ECC. The coprocessor is based on two scalable architectures for finite field computations. One of them is a combinatorial Karatsuba multiplier, the other is a finite field coprocessor that implements field multiplication, addition and squaring in hardware. ECC software implementation performance can be improved by using these two finite field coprocessors. According to the authors, if only the first architecture is used, scalar multiplication can be performed in 184 ms. If both architectures are used, the required time reduces to 10.9 ms. In [36], Ernest *et al.* reported a hardware architecture to perform scalar multiplication over the binary field $F_{2^{270}}$ using projective coordinates. Field elements are represented in normal basis, for that reason a Massey-Omura multiplier is used. Coprocessor architecture consist of three basic blocks: a register file with sixteen 270-bits registers, a finite state machine implementing the *add and double* method to perform scalar multiplication and an arithmetic unit performing field computations. The design was synthesized for an FPGA device equivalent to 180,000 logic gates. The coprocessor occupies 82% of the available CLBs while performing scalar multiplication in 6.8 ms.

Software implementations

Some software implementations of ECC have been reported in [37], [38] and [39]. As reported in [39], there are two libraries which offer primitives for EC cryptography: LiDIA [40] and MIRACL [41]. Both are efficient, general purpose, number-theoretic libraries implemented in C++ that offer a vast number of cryptographic primitives. Although there seem to exist implementations of EC cryptographic protocols (e.g., ECDSA) and EC generation methods (CM method) based on these libraries, these implementations are either not publicly offered (LiDIA), or are partly provided (MIRACL).

In [37], Hankerson *et al* present a C implementation on a Pentium II 400 MHz workstation for performing the scalar multiplication on F_{2^m} and optimized implementation of the elliptic curve operation $u_1G + u_2Q_A$ performed in ECDSA verification. This optimization is based on the fact that the point G is known *a priori*, so some precomputation can be done. NIST random and Koblitz elliptic curves over the binary field were used. They implement several algorithms for field arithmetic scalar multiplication. The best of their results is achieved by pre-computing values in the operation kP. They conclude aspects about implementing algorithms for inversion. In this implementation, for $m = 163$, inversion can be achieved in 30.99 μs for the extended euclidean algorithm, 42.49 μs for the almost inverse algorithm and 40.26 μs for the modified almost inverse algorithm. Multiplier implemented was the Karatsuba method (3.92 μs) and the shift and add method (16.36 μs). Scalar multiplication is executed in 9.178 ms applying the binary method and using affine coordinates. If projective coordinates are used, multiplication can be done in 4.71 μs. If values are pre-computed, scalar multiplication is achieved between 1.683 and 3.44 ms. Assuming a m-bit value for the hash value previously computed, ECDSA verification can be done in 5.005 ms for $m = 163$.

In [38], an ECC implementation on a Palm OS Device is presented. A NIST (random and Koblitz) curve over the finite field $F_{2^{163}}$ was used and projective coordinates were selected. Code was written in C and executed on a Handspring Visor with 2 MB of memory. For the random curve, the scalar multiplication takes 3.5 sec using the Montgomery algorithm [42]. For the Koblitz curve, the scalar multiplication takes 2.4 sec.

In [39] an ANSI C software library for ECC is presented. The GNU Multiple Precision arithmetic library is used to perform the arithmetic operations over the prime finite field F_p. The library was tested on a Pentium III 933 MHz 256 Mbytes workstation using prime finite fields $F_{p^{175}}$, $F_{p^{192}}$ and $F_{p^{224}}$; for each of these finite fields, timing results for performing scalar multiplication were 13.6 ms, 15.7 ms and 19.5 ms respectively. In this work, ECDH, ECES and ECDSA protocols were implemented to test the proposed library. For the finite field $F_{p^{192}}$, timing results are as follows: assuming MAC

value previously computed, ECES encryption takes 36.5 ms, ECDSA digital signature generation takes 22.7 ms and 28.3 ms for verification.

Because of elliptic curve cryptographic algorithms perform a high amount of mathematical operations with large numbers in a finite field; implementing ECC with the available instruction set of a general purpose processor is inefficient, as mentioned in the software implementations. For this reason, and the advantages of a hardware implementation, this last solution seems to be better suited for ECC algorithms.

2.4 Summary of the Chapter

In this Chapter, the basic background of this research work was exposed. Some theoretical aspects of lossless compression, cryptography and elliptic curves were described. In the following Chapter, the architecture of the system in presented based on the best decision made according to the delimitations of the design.

Chapter 3

Hardware design of the system

This chapter discusses the architecture of the proposed system. An unified architecture is designed to support both echemes ECDSA and ECIES adding data compression. The system is composed of two main architectures: one for processing data on the fly and other one for arithmetic operations.

Data processing involves data compression and cryptographic operations. Cryptpgraphic operations can be either MAC or hash computations depending on what scheme, ECIES or ECDSA, is executed. The arithmetic units perform either elliptic curve arithmetic or integer modulo n arithmetic.

Architectures were designed by identifying shared components and applying parallelism at algorithm level and pipeline techniques. The architecture is designed so it can only perform data compression or only executes ECDSA or ECIES without data compression.

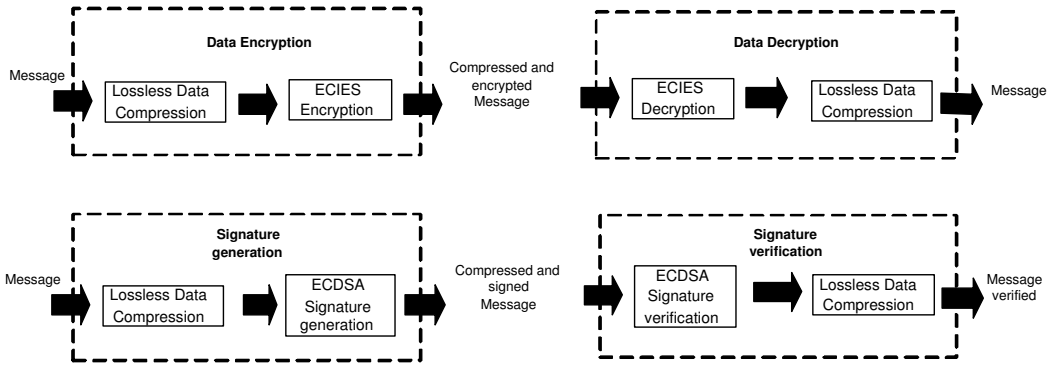

Figure 3.1: Block diagram of the system

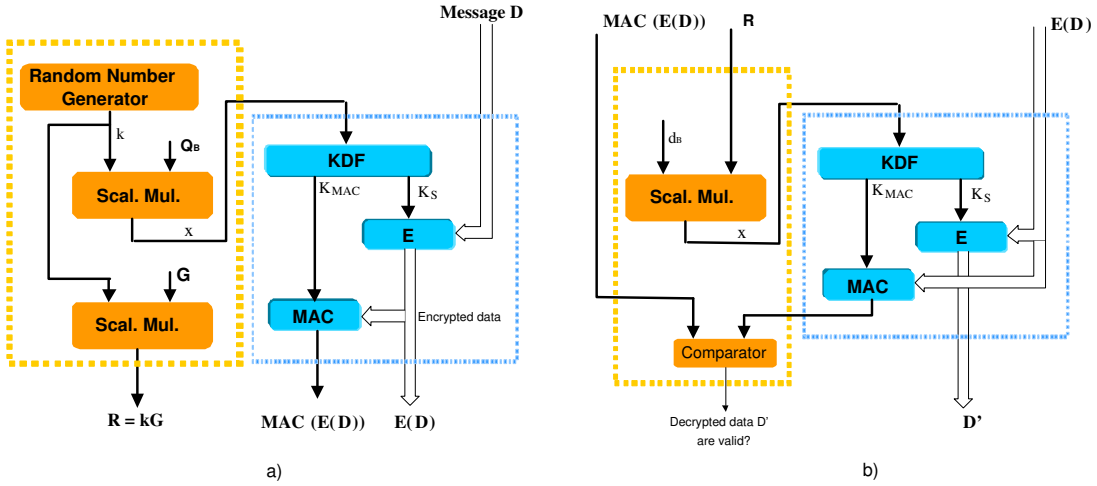

Figure 3.2: ECIES scheme data flow. a) encryption, b) decryption

3.1 The system

The hardware architecture of the system is a customized implementation of the elliptic curve cryptographic schemes ECIES and ECDSA, described in section 2.3.2, and a compression module.

The system executes the cryptographic operations of data encryption/decryption and digital signature generation/verification. When data are encrypted (ECIES) or digitally signed (ECDSA), they are previously compressed. When data are decrypted or a digital signature is verified, data are not compressed, but decompressed after the cryptographic operations. These operations are shown as black box in figure 3.1.

When compression is required, the system makes transparent the process of encryption and compression. Different of performing the compression of all data and then encrypting them, like in a sequential implementation, the system processes data produced by the compressor module on the fly. When all input data have been compressed, the cryptographic schemes have been also applied.

3.1.1 Adding data compression to the ECIES scheme

Figure 3.2 shows the data flow in both, the encryption and decryption operations in ECIES, described in section 2.3.2. When encrypting, two scalar multiplications are computed; kQ_B and kG. k is a random number, Q_B is the public key of the receiver and G is the shared point in the tuple T. The x-coordinate of the point kQ_B is used by the key derivation function **KDF** to generate the keys K_{MAC} and K_S. The symmetrical cipher **E** encrypts incoming data using the key K_S and the result is authenticated by the message authentication code algorithm **MAC** K_{MAC}. The result of the encryption

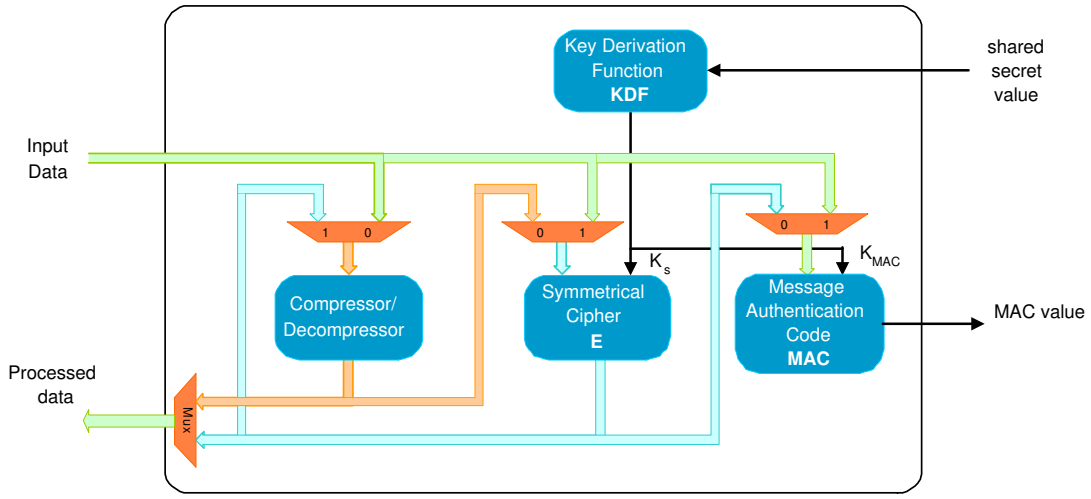

Figure 3.3: Processing data for ECIES: addition of a data compression module

process is the point $R = kG$, the MAC value $MAC(E(D))$ and the encrypted data $E(D)$.

In a decryption operation only one scalar multiplication is computed; $d_B R$. d_B is the private key of the receiver an R is part of data to be decrypted. From the x-coordinate of point $d_B R$, KDF produces the keys K_{MAC} and K_S. The incoming data are assumed in a compressed form; they are authenticated and decrypted by the MAC and E algorithm respectively. The deciphered data are passed to the decompressor to recover the data set D'. If the current computed MAC value MAC(E(D)) is equal to the incoming one, then the decompressed data D' are accepted as the original message D sent by the emisor.

In figure 3.2 a) and b), two main modules framed in dashed lines are identified: one that just involves arithmetic operations and other involving data processing. In the first one, the main operation is the scalar multiplication. The second one involves the coordinated execution of the modules KDF, E and MAC. The data source for MAC is different depending on which operation, encryption or decryption, is performed.

Data compression should be added to the processing data module. That is, data are compressed, encrypted and authenticated. In a decryption operation, the incoming data are decrypted and authenticated. The resulting decrypted data is then decompressed. This is shown in the block diagram in figure 3.3. The data source for every data procesing unit is controlled by multiplexers.

An external shared secret value is used to derive the keys K_s and K_{MAC}, which are used by E and MAC respectively. The output of the processing data module is either compressed and encrypted data or decrypted and decompressed data. In an encryption operation, the MAC value is computed and sent with the encrypted data.

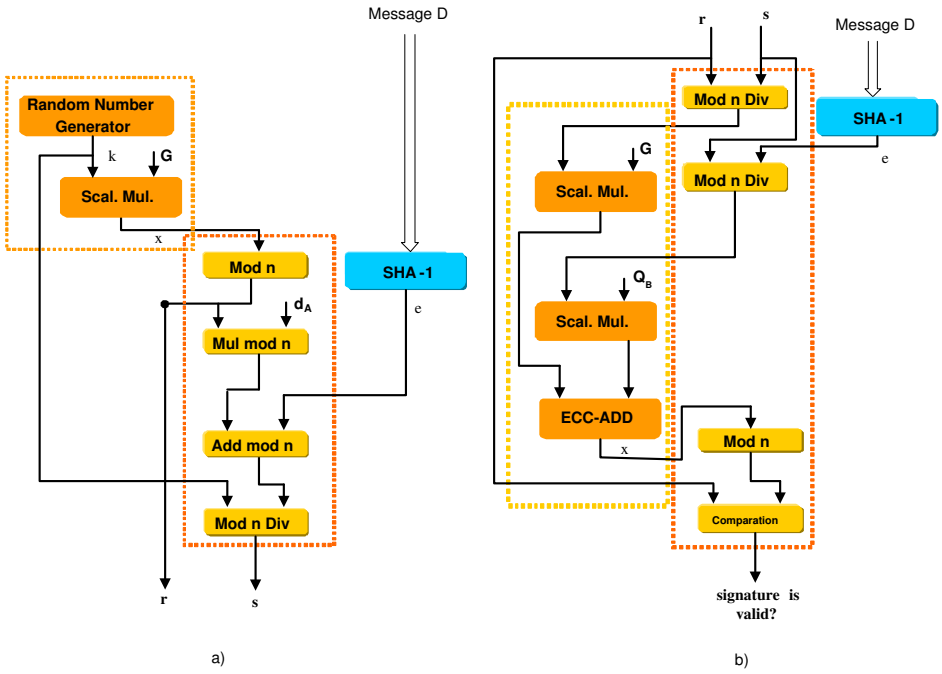

Figure 3.4: Data flow for signature generation/verification in the ECDSA scheme

In a decryption operation, the MAC value on the incoming data is used to authenticate.

3.1.2 Adding data compression to the ECDSA scheme

Figure 3.4 shows the data flow for a) signature generation and b) signature verification in the ECDSA scheme described in section 2.3.2. For signature generation, only the scalar multiplication kG is computed; being k a random number and G the shared point in tuple T. The first component of the signature, r, is computed by applying modular reduction to the x coordinate of kG. The second part of the signature, s, is obtained from modulo n operations that involves the r value, the hash of input data e, the random number k and the private key d_A.

In a signature verification, two scalar multiplications are computed. The two scalars are derived from modulo n operations involving the incoming signature under test and the hash value of incoming data. The points are the shared point G and the public key Q_A of the emisor. Different to signature generation, a sum of points is additionally required. The x-coordinate of this elliptic point sum is used to determine whether the signature is valid or not.

As in ECIES, the two modules for arithmetic operations and data processing are present. The compression module for ECDSA should have to compress the incoming data before hashing in a signature generation. Data must be decompressed after data

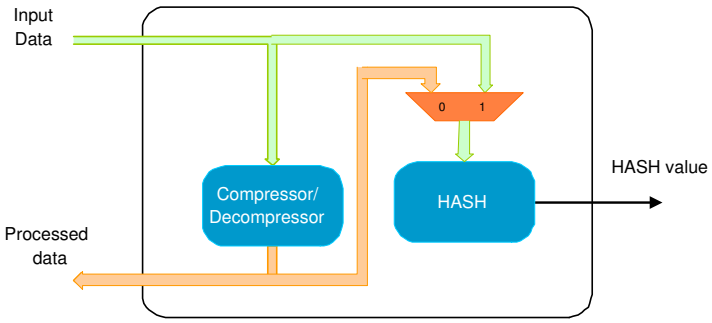

Figure 3.5: Data compression added to ECDSA

hashing when signature is verified. This is shown in figure 3.5. Although the data processing is simpler than ECIES, the arithmetic unit is more complex.

3.2 System specifications

Before discussing the architecture of the system, the algorithms for data processing to be considered in this work are showed.

For ECDSA, the SHA-1 algorithm is used for hash computation in both, signature generation and verification. SHA-1 is iterative, one-way hash function that produces a condensed 160-bit representation, called a message digest of data of any size. SHA-1 can be described in two stages: *preprocessing* and *hash computation*. Preprocessing involves padding the message, parsing the padded message into 512-bit blocks, and setting initialization values to be used in the hash computation. The purpose of the padding is to ensure that the padded message is a multiple of 512 bits. Padding consist in adding the bit '1' and the length of D in bits as a 64-bit number. To fill a 512-bit block, zeros are added. Then, the padded message is parsed into NB 512-bit blocks, $D^{(1)}, D^{(2)}, ..., D^{(NB)}$. The initial hash value, $H^{(0)}$, consist of the following five 32-bit words, in hexadecimal:

$$H_0^{(0)} = \text{0x67452301}$$
$$H_1^{(0)} = \text{0xefcdab89}$$
$$H_2^{(0)} = \text{0x98badcfe}$$
$$H_3^{(0)} = \text{0x10325476}$$
$$H_0^{(0)} = \text{0xc3d2e1f0}$$

From $H^{(0)}$, every hash value $H^{(i)}$ of block $D^{(i)}$, $1 \leq i \leq NB$ is used to compute the next hash value $H^{(i+1)}$ corresponding to the next block $D^{(i+1)}$.

The hash computation generates a *message schedule* from the padded message and

Algorithm 2 HASH computation: SHA-1 core

Require: $D^{(i)}$ a 512-bit block

Ensure: The HASH value corresponding to $D^{(i)}$ from $H^{(i-1)}$

1: Prepare the message schedule, W_t from block $D^{(i)}$:

$$Wt = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \leq t \leq 79 \end{cases}$$

2: Initialize the five working variables A, B, C, D, and e, with the $(i-1)$st hash value:

$A \leftarrow H_0^{(i-1)}$
$B \leftarrow H_1^{(i-1)}$
$C \leftarrow H_2^{(i-1)}$
$D \leftarrow H_3^{(i-1)}$
$E \leftarrow H_4^{(i-1)}$

3: **for** t from 0 to 79 do **do**

4: $E \leftarrow D$
$D \leftarrow C$
$C \leftarrow ROTL^{30}(B)$
$B \leftarrow A$
$A \leftarrow ROTL^5(A) + f_t(B, C, D) + E + k_t + W_t$

5: **end for**

6: Compute the ith intermediate hash value $H^(i)$:

$H_0^{(i)} \leftarrow A + H_0^{(i-1)}$
$H_1^{(i)} \leftarrow B + H_1^{(i-1)}$
$H_2^{(i)} \leftarrow C + H_2^{(i-1)}$
$H_3^{(i)} \leftarrow D + H_3^{(i-1)}$
$H_4^{(i)} \leftarrow E + H_4^{(i-1)}$

uses that schedule, along with functions, constants, and word operations to iteratively generate a series of hash values, one for every 512-bit block. Every 512-bit block is processed in 80 iterations and the result of the final block becomes the message digest.

The hash computation is applied to every block message $D^{(i)}$ according to algorithm 2. After processing the block $D^{(NB)}$, the resulting 160-bit message digest of the data set D, is $H_0^{(NB)}||H_1^{(NB)}||H_2^{(NB)}||H_3^{(NB)}||H_4^{(NB)}$, where $||$ means a bit-string concatenation. In algorithm 2, operation $ROTL^j(A)$ means a left circular shift of value A j positions.

The definition of function $f_t(B, C, D)$ is

$$f_t(B, C, D) = \begin{cases} (B \wedge C) \oplus (\neg B \wedge D) & 0 \leq t \leq 19 \\ (B \oplus C \oplus D & 20 \leq t \leq 39 \\ (B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D) & 40 \leq t \leq 59 \\ (B \oplus C \oplus D & 60 \leq t \leq 79 \end{cases}$$

The four 32-bit constants k_t are defined as

$$k_t = \begin{cases} 0x5a827999 & 0 \leq t \leq 19 \\ 0x6ed9eba1 & 20 \leq t \leq 39 \\ 0x8f1bbcdc & 40 \leq t \leq 59 \\ 0xca62c1d6 & 60 \leq t \leq 79 \end{cases}$$

The key derivation function **KDF** considered is this work is specified in ANSI X9.63. **KDF** uses a shared secret Z to derive a bit-string of arbitrary *length* by executing the SHA-1 algorithm $\lceil length/160 \rceil$ times. The bit-string is formed by concatenating the hash result in each iteration. Algorithm 3 lists the steps of KDF.

Algorithm 3 Key derivation function KDF

Require: An octet string Z which is the shared secret value.
Require: An integer *keydatalen* which is the length in octets of the keying data to be generated.
Ensure: The keying data Key which is an octet string of length *keydatalen* octets, or 'invalid'.
1: Check that $Z + 4 < hashmaxlen$. If not, output 'invalid' and stop.
2: Initialize a big endian 4-octet string *counter* as 00000001_{HEX}
3: **for** i from 1 to $\lceil keydatalength/hashlen \rceil$ **do**
4: $k_i \leftarrow HASH(Z||Counter)$
5: $counter \leftarrow counter + 1$
6: **end for**
7: Key is the bit-string $K_1||K_2...K_{\lceil keydatalength/hashlen \rceil}$ trunctated to *keydatalength* octects.

The hash value computed in step 5 in algorithm 3 is on fixed size data. In every iteration, only the value of the counter changes, so, the 512-bit block can initially be padded and then the part corresponding to the counter can be updated.

For message authentication code **MAC**, the HMAC-SHA-1-160, specified in ANSI X9.71 is selected. It uses a 160-bit K_{MAC} key and the SHA-1 algorithm for hashing. HMAC ensures that secure data is not corrupted in transit over unsecure channels (like the internet) and is used in ECIES operations. HMAC generates a Message Authentication Code using the following formula: HMAC(D)= $HASH[(K_{MAC} \oplus opad)||HASH[(K_{MAC} \oplus ipad)||D]]$,
where

D	=	Message
H	=	Underlying Hash function
K_{MAC}	=	Secret Key
$opad$	=	$0x36$, repeated as needed
$ipad$	=	$0x5C$, repeated as needed
$\|\|$	=	concatenation operator
$\oplus$	=	XOR operation

SHA-1 computes the hash value on two different data sets. First, SHA-1 is applied to the bit string $D_1 = (k_{MAC} \oplus ipad) \parallel D)$. The second data set is $D_2 = (k_{MAC} \oplus opad) \parallel$ SHA-1(D_1).

HMAC(D) is then sent as any typical MAC(D) in a message transaction over insecure channels. HMAC has all of the general properties of a MAC function and it is computationally very fast and very compact.

The symmetrical encryption algorithm **E** is the XORing encryption. This kind of encryption consist in a XOR operation between the key K_S and data, so, K_S should be of be the same as the input data. So, **KDF** needs to generate a 160-bit K_{MAC} key and a K_S key of the same length that the message to be encrypted/decrypted.

Based on the previous specifications, next section describes an unified architecture that gives support to ECIES and ECDSA for data processing.

In both ECIES, and ECDSA, the elliptic curve is defined on the binary field F_{2^m} using polynomial basis. Based on the literature reviewed, this field leads to an efficient hardware implementation. Algorithms for performing these operations and their architectures are described in the last section of this chapter.

3.3 Unified architecture for data processing and data flow

The architecture for data processing consists of four main modules:

- A **data compressor module** that implements a variant of the first algorithm LZ to compress/decompress data.
- A **KDF** module that implements the KDF algorithm to generate the keys K_{MAC} and K_S.
- A **simmetrical cipher** E module that encrypts and decrypts data by XORing data with key K_S.
- An **HMAC** module that implements the HMAC algorithm for message authentication code generation. In ECDSA operations, this unit computes not the MAC but the HASH value of input data.

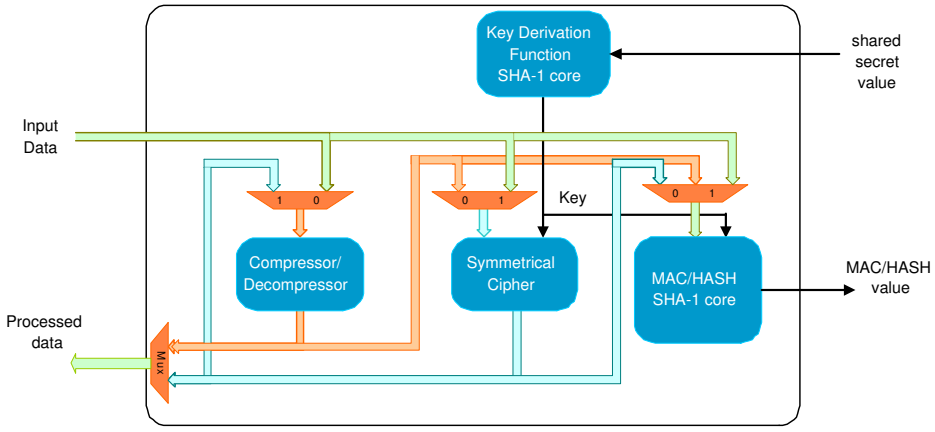

Figure 3.6: Data processing for ECDSA and ECIES

The block diagram for the unified data processing is shown in figure 3.6. The architecture is similar to the architecture for ECIES. The main difference is that the initial MAC module can also perform hash computation.

The four modules for data processing receive data from different sources, depending on which operation (data encryption, data decryption, signature generation, signature verification) is executed. The approach followed in this work was to design every module so that it could receive data from any external device and at the same time it provides an interface to send data to another entity. This was achieved by providing similar data input/output for data communication. Any module does not know where data is coming from but only processes them when ready. Under this approach, data and control signals can be switched by multiplexers without altering the internal functionality of each module. So, the compressor, symmetrical cipher E and HMAC modules include the data control signals listed in table 3.1.

Table 3.1: Common control signals for internal data communication

Signal	Type	Description
`dat_in`	input	Input data to be processed
`dat_out`	output	Data processed (compressed, encrypted, decrypted)
`v_dat`	inpt	Indicates data in `dat_out` bus is valid
`m_dat`	output	Indicates new source data must be put in the the `dat_in` bus
`eod`	input	Indicates the end of input data

Every module can send a request for data by enabling signal `m_dat`. Data will be considered as valid when `v_dat` is valid, and it is responsibility of the external entity. When module is not able to receive new data, line `m_dat` is disabled. Signal `eof` indi-

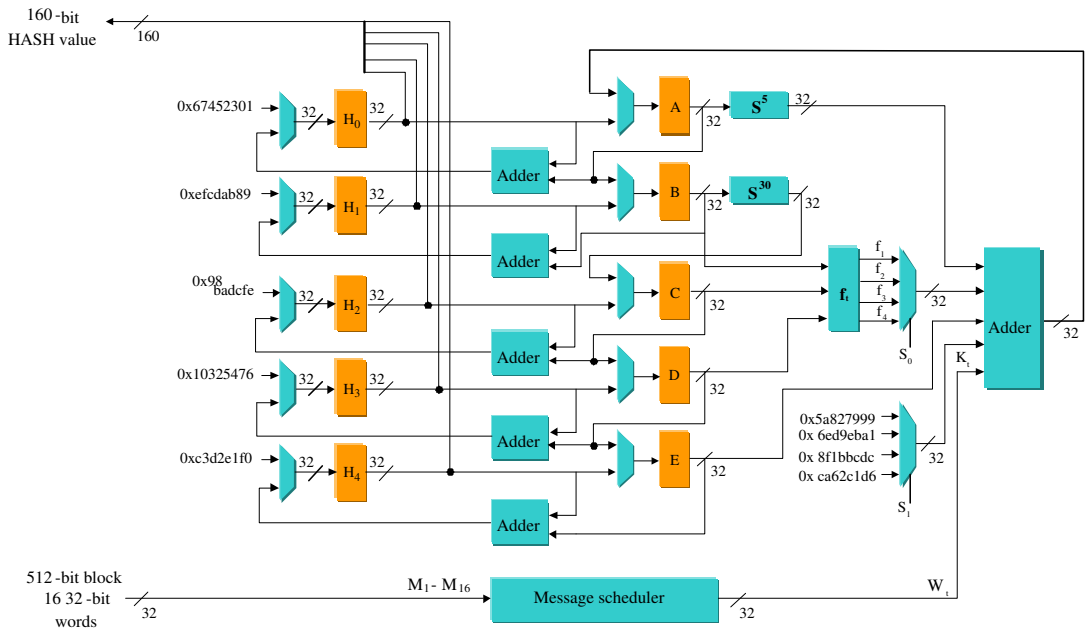

Figure 3.7: SHA-1 core

cates when data is finished. In this work, data produced by the compressor is always processed, so the signal `m_dat` is ignored by the compressor module. Detailed description of the four main modules is presented in following sections.

3.3.1 Architecture for the SHA-1 core operation

Figure 3.7 shows the hardware design for algorithm 2 which computes the hash value of a 512-bit block. $H_0 - H_1$ are registers that keep the current hash value for a given block. For every new block, registers $A - E$ are loaded with current values in registers $H_0 - H_1$, that are updated at the end of the hash computation. The control of the 80 iterations and correct values assignment for f_t and k_t is performed by a Finite State Machine.

The 512-bit block is serially given to the SHA-1 core as sixteen 32-bit words M_0 to M_{15}. For every iteration, a scheduler message produces every W_t value as specified in SHA-1 algorithm. The Message Scheduler is based in the work reported by Grembowsky in [43]. He used 16 32-bit registers connected in cascade, one 32-bit 1-to-2 multiplexer, one 4-input 32-bit XOR and a 32-bit circular shifter. A diagram of the scheduler is shown in figure 3.8.

The first sixteen W_t words are the values M_0-M_{15}. These values initially fill the 16 registers and allow to compute the next 64 words. After W_{15}, every new W_t is given by the result of the circular shift that takes previous values of W_t. So, the 80 values W_t

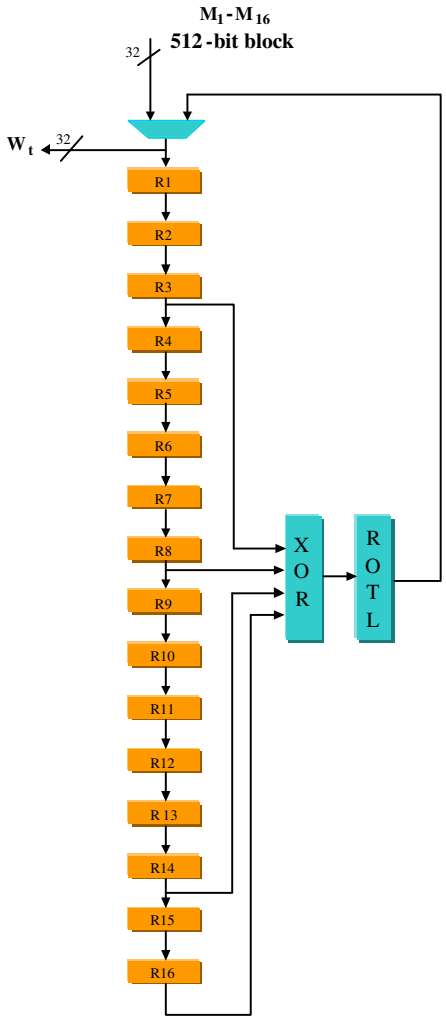

Figure 3.8: Message Scheduler for SHA-1 core

are generated at every iteration using only 16 registers.

3.3.2 HMAC module

The HMAC module consist of a pre-processing unit, the SHA-1 core module and a control unit. The preprocessing unit is close related to the SHA-1 core block. Data to be authenticated or hashed are received by the pre-processing unit, which is responsible to perform the padding and parsing as specified in algorithm SHA-1. The control unit is a finite state machine that makes the initialization of registers $H_0 - H_1$ and implements the necessary tasks to either compute the MAC or the HASH value. General architecture for HMAC is shown in figure 3.9. In a message authentication code computation, once the key k_{MAC} is ready, the control unit sends the bit-string $D_1 = (k_{MAC} \oplus ipad)$ to

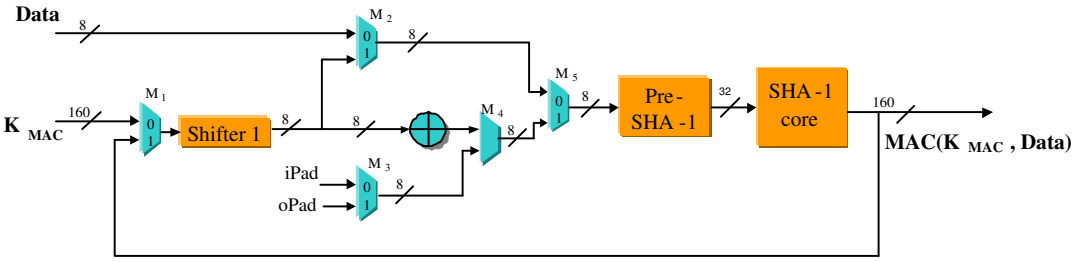

Figure 3.9: HMAC algorithm

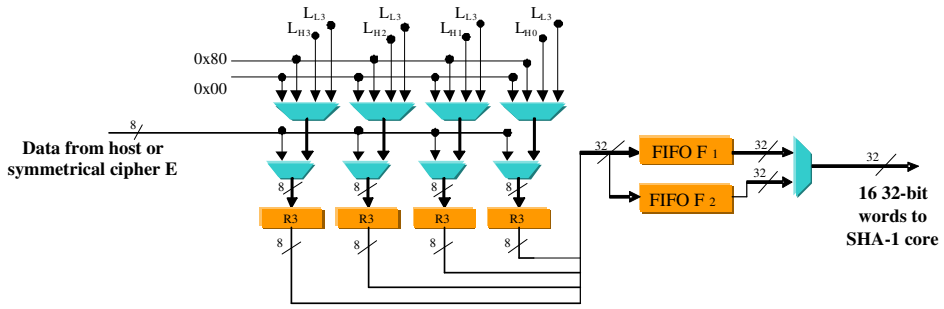

Figure 3.10: Prepocessing module for SHA-1

the pre-processing unit and then turns the control signal and data to the external host. Once all data are entered and the first hash is computed, the control unit re-initialize the SHA-1 core module and sends the string $(k_{MAC} \oplus opad)||HASH(D_1)$ for final hash computation and then the MAC value. In a hash operation, the control unit is not used, and input data for the pre-processing module are given directly by the external host.

Figure 3.10 shows the preprocessing unit for SHA-1. Incoming data are enqueued in one of two 512-bit FIFO memories F_1, F_2, arranged as 16 32-bit words. Initially F_1 is the current FIFO where data coming from the compressor is enqueued. When the current FIFO is ready, a 512-bit block can be processed according the SHA-1-core algorithm. Data in the current FIFO is dequeued serially during the first 16 iterations of SHA-1 core, in this period of time, data produced by the compressor is enqueued in the other 512-bit FIFO memory F_2, that becomes the current FIFO. F_1 and F_2 switches as more data are processed by the compressor, when the compressor finishes, the padding in performed on the current memory block. If necessary, a new block is generated as is specified in SHA-1 algorithm using the free FIFO memory.

3.3.3 KDF

The architecture for key derivation function consists of the SHA-1 core block, a 16x32 RAM memory and a control unit. In this case a preprocessing unit is not necessary

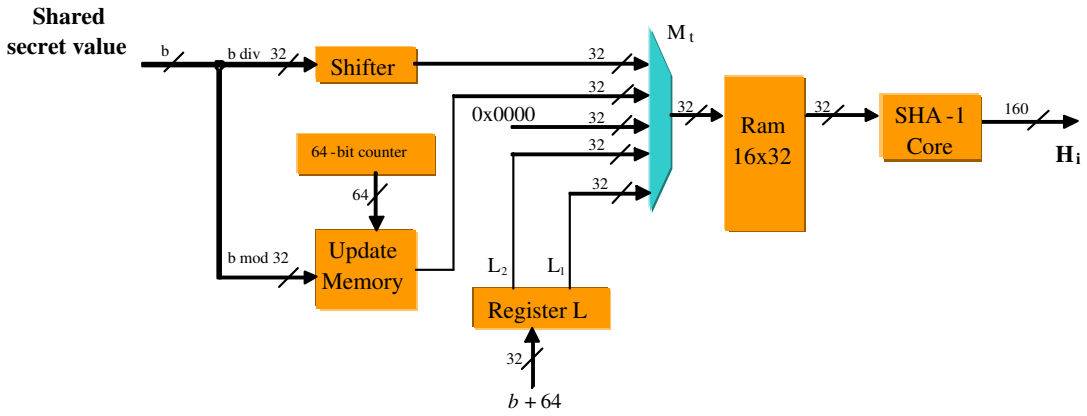

Figure 3.11: Key derivation function KDF

because SHA-1 is applied to a fixed size of data. KDF block consists of the shared secret value and a 32-bit counter. The RAM memory becomes the 512-bit block being processed by the SHA-1 core block. According to the current state of the art, keys no longer than 233 bits are necessary in ECC. So the key's length and the counter fits in only one 512-bit block and so only one memory is used.

KDF waits a shared secret value Z of size b bits. If b is not a multiple of 8, Z is padded with $'0'$ so it converts in an octet string. When the secret is ready, the RAM memory is set by storing the bit-string $Z||counter$ with the padding as a 512-bit block ordered in 16 32-bit words.

Every time a new 160-bit result of KDF is requested, the content of the memory is updated only in the part corresponding to the counter and then the computation is performed.

For every new hash computation, the counter is incremented and only the locations corresponding to the counter in the memory are updated. Then, the hash value is computed from the current content of the memory. The diagram of KDF implementation is shown in figure 3.11. The shifter module takes the $b/32$ less significant bits of the secret shared value Z. Every 32-bit word of Z is stored in the 32x16 memory block. The $(bmod32)$ remainder bits, jointly with the initial value of the 32-bit counter, are stored as the following 32-bit words in the memory. If it is supposed that b is less than 416, what occurs in practical ECIES implementation, then the padding and the required zeros are stored in the consecutive memory locations according SHA-1 algorithm. In the last two words of memory, the size of the input puls the size of the counter are stored as a 64-bit value. For every new iteration in KDF, the counter is incremented and the memory is updated. Once memory has been updated, the SHA-1 core operation starts by reading the current 512-bit block stored at memory as 32 bit words.

3.3.4 The data compressor: a systolic array approach

After reviewing the literature related to hardware implementation of the first LZ algorithm, it was decided to implement it by using a systolic array approach. Under this approach, hardware implementation occupies fewer resources and clock rate is high. These two aspects should be taken into account in order to achieve a low complexity system.

The systolic array was derived by studying the data dependence in the LZ algorithm. Reconsidering the algorithm 1 listed in section 2.1.2 to find the largest string in buffer Y, being the prefix in buffer X. The inner loop controlled by variable j can be redefined using an extra variable *match*. This makes the inner loop independent of the current data being tested. The new algorithm becomes as listed in algorithm 4.

Algorithm 4 Step two in the LZ algorithm, redefined

Require: X, Y: searching and coding buffer
Ensure: A codeword $C = \{pointer, maxLength\}$

```
1:  maxLength ⇐ 0, pointer ⇐ 0
2:  for i from 0 to N − 1 do
3:     length ⇐ 0, match ⇐ 1
4:     for j from 1 to M do
5:        if X[i + j] = Y[j] and match = 1 then
6:           length ← j
7:        else
8:           match ← 0
9:           length ← length
10:       end if
11:    end for
12:    if length > maxLength then
13:       maxLength ← length
14:       pointer ← i
15:    end if
16: end for
```

The outer loop finds the largest string from every pointer in buffer X. For every pointer i, the inner loop increments the variable *length* for every consecutive success comparison and keeps with the last value at the first comparison failure. When the inner loops finishes, *length* is the maximum length of the string that starts at pointer i and is the prefix in buffer Y. The variable *maxLength* and *pointer* are updated for every new pointer tested, when all pointers have been tested, *maxLength* and *pointer* indicate the largest string that matches in the buffer Y.

Data dependence is studied by applying loop unrolling. In figure 3.12 they are defined basic cells representing main statements in algorithm 4. Lines 5-9 are represented

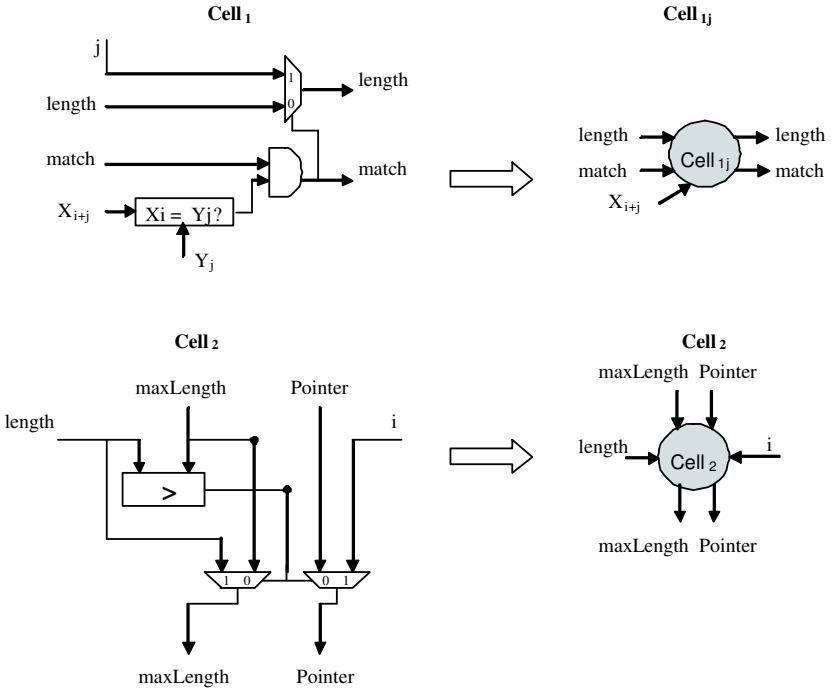

Figure 3.12: Basic cells

by the cell $cell_1$; lines 12-15 are represented as $cell_2$. $Cell_{1j}$ represents the main body of the inner loop for a fixed value of j. The $cell_{1j}$ updates the variables *length* and *match* that will be the inputs for following cell $cell_{1j+1}$. The output *length* of the cell $cell_{1M}$, which is the max length for the current pointer i is the input to $cell_2$, which compares it with the global variable *maxLength*. If current *length* is greater, *length* becomes the *maxLength* and the *pointer* is updated to i. Otherwise, previous values of *maxLengh* and *pointer* remain.

To illustrate the data dependence, it is considered an example for a searching buffer X of size 5 and a coding buffer Y of size 3. Let the buffers X and Y be $\{X_1, X_2, X_3, X_4, X_5\}$ and $\{Y_1, Y_2, Y_3\}$ respectively. Figure 3.13 shows the unrolled inner loop for every pointer i, using the basic cells previously defined. In figure 3.13, let define the cell $cell_{i,1j}$ and $cell_{i,2}$ as the $cell_{1j}$ and $cell_2$ corresponding to pointer i. The computations in every row corresponding to pointer i are performed serially; $cell_{i,1j}$ requires the results from $cell_{i,1j-1}$. In the graph, zz input to $cell_{i,1j}$ indicates that entry is not valid. After executing every cell at every row, the final codeword is given by cell $cell_{5,2}$.

In figure 3.13, computations are performed from $i = 1$ to $i = 5$. It does not matter if the computations are performed in the reverse order, from pointer $i = 5$ to $i = 1$. The only change is the data flow for $cell_2$, now initial values for *maxLengh* and *pointer* are

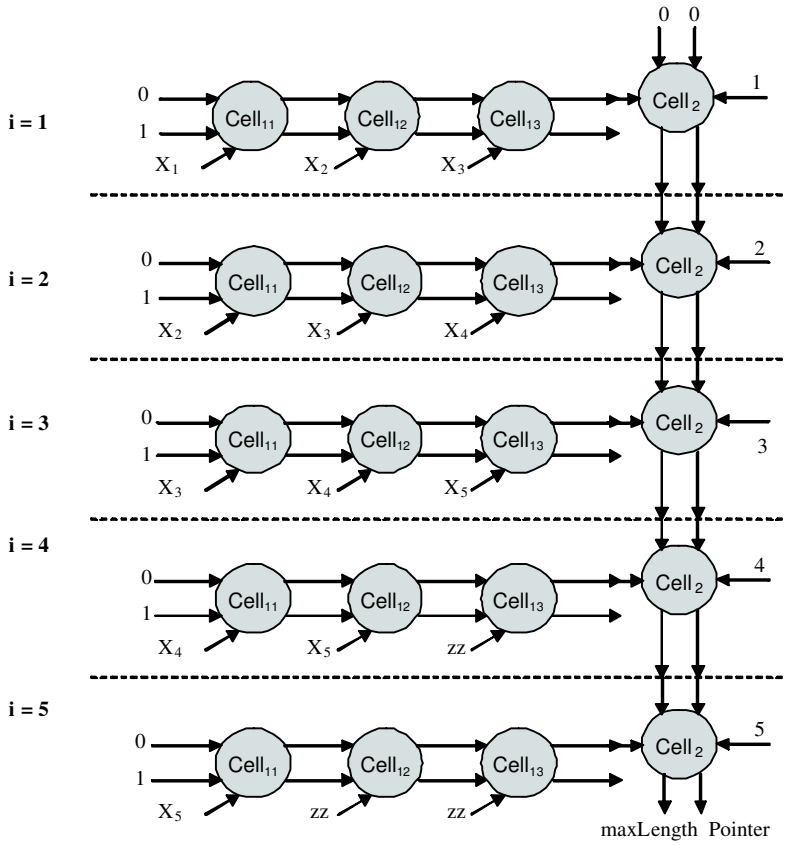

Figure 3.13: Unrolling the LZ algorithm

given in $cell_{5,2}$ and results are passed to $cell_{4,2}$ and so on. So, the final codeword will be given by cell $cell_{1,2}$ when the row $i = 1$ finishes.

From figure 3.13, it is seen that symbols processed by cells $cell_{i,1j}$ are used by cells $cell_{i-1,1j+1}$, that is, the symbol used in one cell $cell_{1j}$ will be used for the following cell $cell_{1j+1}$. Instead of doing all computations for $i = 5$ and then all computations for $i = 4$, and so on, processing data can be done in a pipelined form. In table 3.2, a timing schedule for computations of figure 3.13 is given. All computations in figure 3.13 are listed in table 3.2. The pipeline is accomplished by adding registers in each cell in order to keep the required signals for the other cells. In table 3.2, the *length* and *match* values produced by cell $cell_{1j}$ are registered and then, they are used for the following cell $cell_{1j+1}$ at the next unit of time. The symbol X_i processed by cell $cell_{1j}$ is delayed two units of time for the next cell $cell_{1j+1}$. The $cell_2$ computes a valid comparison receiving the length of every pointer every following unit time from $time = 4$. This cell will keep the results in registers to use them in the next unit of time.

In this example, all computations can be done using only three cells $cell_{1j}$ and one cell $cell_2$. Symbols from the searching buffer X enter to $cell_{11}$ every unit of time and

Table 3.2: Timing space scheduling

Time	$cell_{11}$	$cell_{12}$	$cell_{13}$	$cell_2$
1	X_5-Y_1	zz-Y_2	zz-$Y3$	zz
2	X_4-Y_1	zz-Y_2	zz-$Y3$	zz
3	X_3-Y_1	X_5-Y_2	zz-$Y3$	zz
4	X_2-Y_1	X_4-Y_2	zz-$Y3$	5
5	X_1-Y_1	X_3-Y_2	X_5-$Y3$	4
6	zz-Y_1	X_2-Y_2	X_4-$Y3$	3
7	zz-Y_1	zz-Y_2	X_3-$Y3$	2
8	zz-Y_1	zz-Y_2	zz-$Y3$	1

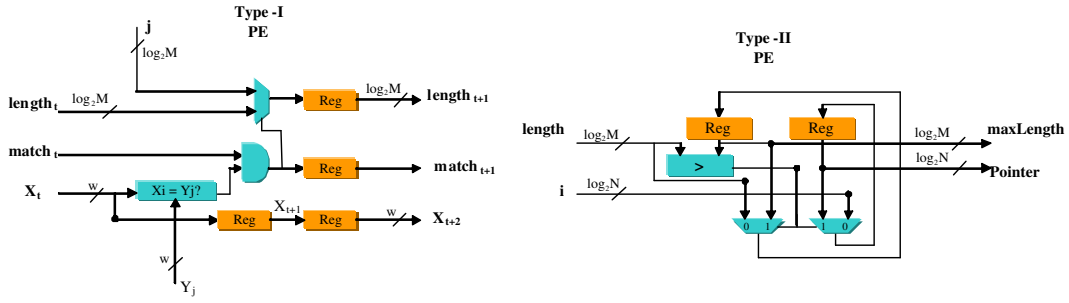

Figure 3.14: Processing elements

they are propagated to the other internal cells delayed two units of time. In the example of figure 3.13, for a searching buffer X of size 5 and a coding buffer Y of size 3, it was necessary 8 units of time to get the codeword. It can be generalized that following the schedule in table 3.2, for an M-size searching buffer X and an N-size coding buffer Y, the codeword is found in $N + M$ units of time, and M $cell_{1j}$ and 1 $cell_2$ cells are required.

By inserting the required registers in cells depicted in figure 3.12, $cell_{1j}$ and $cell_2$ become the type-I and type-II processing elements respectively, both depicted in figure 3.14. The systolic array is composed of M type-I processing elements connected in cascade and one type-II processing element placed at the end of the array. The type-I processing element consist of the following elements:

- 1 w-bit equal comparator (w is the size in bits of the symbols in the buffers),
- 1 $log_2 M$ multiplexer
- 1 2-input AND gate.
- 1 $log_2 N$-bit register

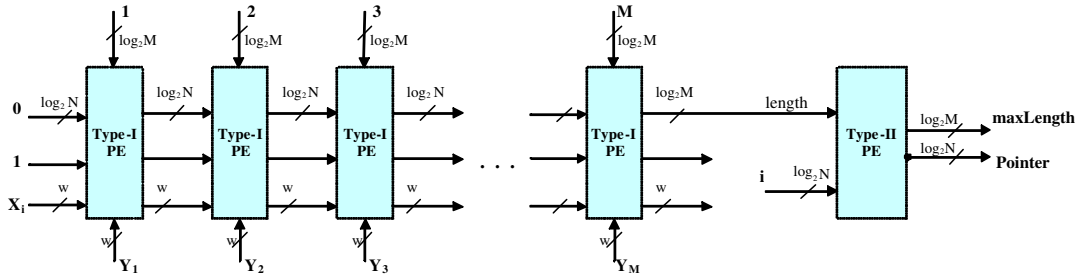

Figure 3.15: The systolic array

- 1 $log_2 M$-bit register,
- 1 w-bit registers and
- 1 flip-flop.

Type-II processing element consists of

- 1 $log_2 N$-bit multiplexer
- 1 $log_2 M$-bit multiplexer
- 1 $log_2 N$-bit register
- 1 $log_2 M$-bit register
- 1 $log_2 M$-bit greater-comparer.

The systolic array is depicted in figure 3.15. Initial values for the first type-I PE inputs, *length* and *match*, are 0 and 1 respectively as specified in algorithm 4. The values of the coding buffer are accessed in parallel, the value for input j in type-I PE is constant and can be considered as an index, indicating the position of the PE in the array. The input i for the type-II PE can be a decreasing counter starting from N.

3.3.5 The searching and coding buffers

For the design of the buffers in the LZ algorithm, it is necessary consider two aspects. The first one is that the design must facilitate the input of new symbols according to the first step of the LZ algorithm, giving the effect of a sliding window. The second one is that it must allow the serial input of symbols X_i in the searching buffer to the systolic array while keeping the searching buffer values (keep the dictionary).

The design for the buffers in this work is as shown in figure 3.16. Both, the searching and the coding buffers are built by connecting w-bit registers R in cascade. A load signal En_1 is shared by all registers in the searching and coding buffers. By

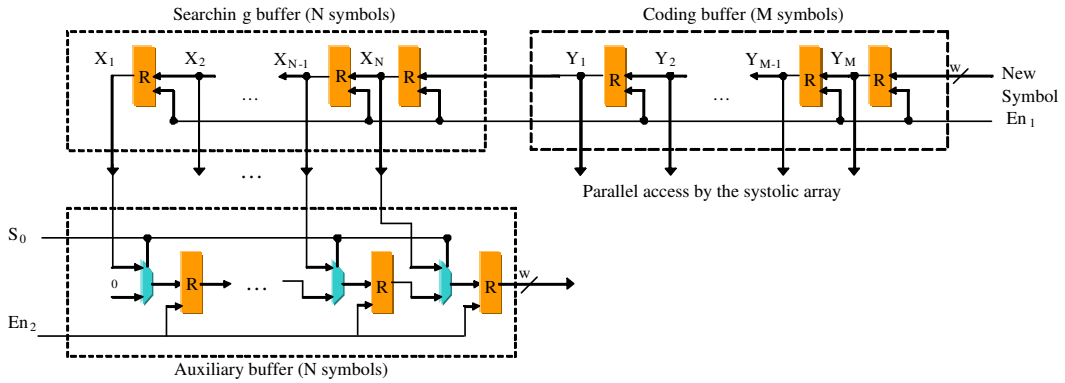

Figure 3.16: Buffers for LZ compression

enabling this signal, new symbols present at input *New Symbol* enter to the coding buffer. This leads to perform one shift operation to the left of the current values of both buffers, as required in the first step of the LZ algorithm. The auxiliar buffer is used to keep the values in the searching buffer while producing one X_i value to the systolic array at every clock cycle. After entering new symbols to the coding buffer, the content of the searching buffer is copied to the auxiliary buffer. One symbol X_i, from $i = N$ downto 1 is produced at every clock cycle by enabling signal En_2. The area complexity of the buffer is $(2N + M)$ w-bit registers and N w-bit multiplexers. The first step in the algorithm is executed by controlling the signal En_1. The second step in the algorithm is carried out by updating the auxiliar buffer and enabling N cycles the signal En_2. At the same time, registers initialization in the systolic array PEs must take place. Once all X_i symbols have been entered to the systolic array, after maximum M clock cycles, the resulting codeword is given by the outputs of the systolic array.

3.3.6 Preventing expantion

LZ algorithm can produce expansion instead of reduction. A codeword is composed of two fields, one is the length of the current matched string, and other is the pointer in the searching buffer where the string starts. The codeword is $(log_2 N + log_2 M)$ long. A codeword replaces the current string matched, when this match length is less than the size of the codeword, the output stream grows. One way to prevent this [21] is to only replace the string with the codeword if the length field is greater than the size of the codeword. Following this approach, the output stream of compressed data will consist of either codewords or no-codified symbols from the source. In this work one-bit flag is used to distinguish between codified and no-codified symbols.

When the codeword is produced by the systolic array, a decision module compares the length field with a threshold, if the length field is greater, the codeword is emitted

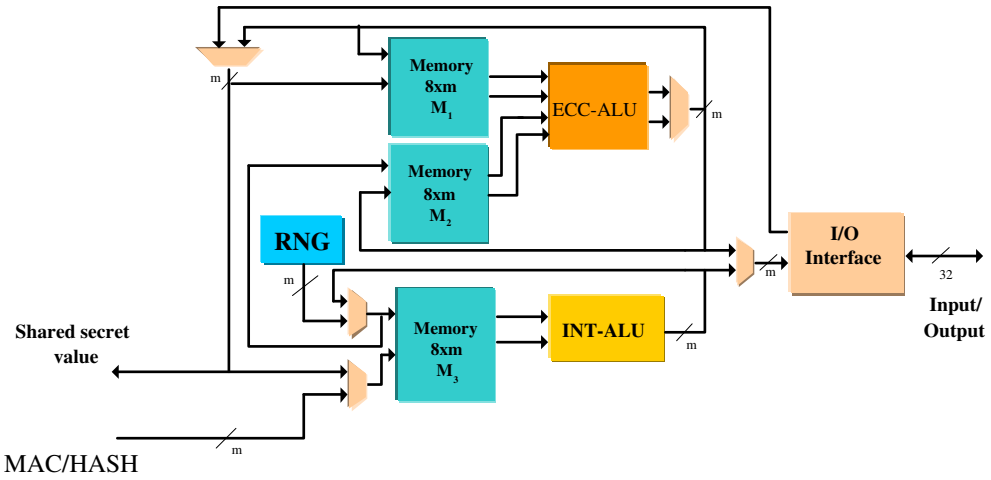

Figure 3.17: Architecture for arithmetic support

as a codified symbol. If not, the first two symbols of the coding buffer are taken as no-codified symbols to the output stream.

3.4 Modulo n and elliptic curve arithmetic support

Special hardware for arithmetic operations is presented in this section. Two main arithmetic units were implemented; one for elliptic curve arithmetic (ECC-ALU) and other for large modulo n integer arithmetic (INT-ALU). The general diagram of hardware for arithmetic support to ECIES and ECDSA is depicted in figure 3.17.

The random number generator provides the random number in the interval [1, n-1] as required in both ECIES and ECDSA. The arithmetic unit ECC-ALU computes either a scalar multiplication or a sum of points in the elliptic curve. It uses two dual-port 8xm memories to store field elements going to be used. The points are stored in memory M_1 and scalars are stored in memory M_2. Memory M_1 stores the public point of both the receiver and emisor. In a decryption operation, the point R is stored in this memory. The memory M_2 stores the scalars involved in the multiplications and also, a point when a signature is verified. Scalars can be either a random number, the private key of the emisor in a signature generation or values given by the INT-ALU. The coordinated execution of each ALU is accomplished by a control unit, which is implemented as a finite sate machine.

The arithmetic unit INT-ALU computes any of the operations required by ECDSA. It uses an 8xm RAM memory M_3 to store temporal values or values coming from either an external host or the ECC-ALU. This memory stores also the MAC value in a ECIES operation for comparison.

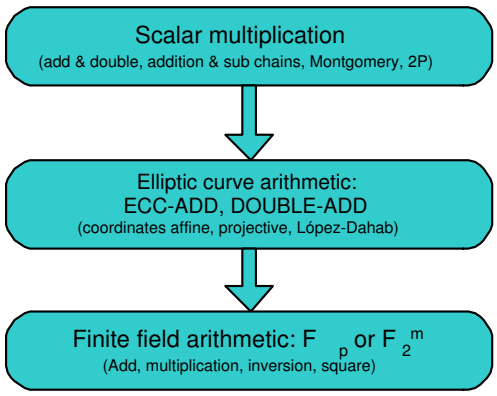

Figure 3.18: Scalar multiplication levels

Detailed implementation of ECC-ALU and INT-ALU are presented in the next sections.

3.4.1 Elliptic curve arithmetic unit

A scalar multiplication is performed in three different stages, as shown in figure 3.18. At the top level, the method for computing the scalar multiplication must be selected. Some methods have been proposed, the most commonly used are the binary method, Add and Subtraction chains, and Montgomery. In the second level, the coordinates to represent elliptic points must be defined. From this representation, the add operation is defined. Possible coordinates are: affine, projective, Jacobeans and López-Dahab. The lower level, but the most important, involves the primitive field operations on which the curve is defined. Basic field operations are sum, multiplication, squaring and division. How these operations are performed depends on the finite field and its efficient implementation impacts the ALU performance to compute the scalar multiplication.

In this work, the binary finite field F_{2^m} was selected because according to literature it leads to efficient hardware implementations compared to the prime finite field [8]. Basic operations in F_{2^m} were defined in chapter 2. Sum operation in F_{2^m} is a simple xor operation. Squaring is a special case of multiplication, although it can be done efficiently if customized hardware is developed. For multiplication, three types of implementation are been reported: serial, parallel and digit-serial. The serial multiplier is simpler and requires m iterations to perform the multiplication. On the contrary, parallel multipliers compute the multiplication in one clock cycle at expenses of higher area resources. Digit-serial multipliers combine the two approaches and are better preferred for implementation. Division is commonly implemented as a composed operation: one inversion is computed and then a field multiplication. To compute the inverse of a field element, two main methods have been commonly used: Fermat's theorem or the

Extended Euclidean Algorithm.

In this work, the binary method is used to compute the scalar multiplication. This algorithm is listed in algorithm 5.

Algorithm 5 Binary method: right to left

Require: $P = (x, y)x, y \in F_{2^m}$ and $k = (k_{m-1}, k_{m-2}, ..., k_0)$
Ensure: $R = kP$
1: $R \Leftarrow (0, 0)$
2: $S \Leftarrow P$
3: **for** i from 0 to $m - 1$ **do**
4: **if** $k_i = 1$ **then**
5: $R \Leftarrow R + S$
6: **end if**
7: $S \Leftarrow 2S$
8: **end for**

The binary method scans every bit of scalar k and, depending on its value, 0 or 1, it performs an *ECC-DOUBLE* operation or both a *ECC-DOUBLE* and an *ECC-ADD* operation. In this work, the points of the elliptic curve are considered in affine coordinates. Algorithm 5 scans every bit of k from right to left. This allows to perform the operations *ECC-DOUBLE* and *ECC-ADD* in parallel. The binary method is implemented as a finite state machine that orchests the execution of two module: one for ECC-ADD and other for ECC-DOUBLE.

For an elliptic curve defined on F_{2^m} using affine coordinates, the operations *ECC-ADD* and *ECC-DOUBLE* are performed according to algorithms 6 and 7 respectively.

The operation *ECC-ADD* requires one inversion, two multiplications, one squaring and eight additions. The operation *ECC-DOUBLE* requires five additions, two squaring, two multiplications and one inversion, all of them, operations on F_{2^m}.

Algorithm 6 ECC-ADD: Sum of diferent points

Require: $P = (x_1, y_1)$,$Q = (x_2, y_2)$,$x_1, y_1, x_2, y_2 \in F_{2^m}$
Require: $a \in F_{2^m}$, a is the constant in the elliptic curve
Ensure: $R = (x_3, y_3) = P + Q$
1: **if** $P = O$ or $Q = O$ **then**
2: $R \leftarrow O$
3: return
4: **end if**
5: $\lambda \leftarrow (y_2 + y_1)/(x_2 + x_1)$
6: $x_3 \leftarrow \lambda^2 + \lambda + x_1 + x_2 + a$
7: $y_3 \leftarrow \lambda(x_1 + x_3) + x_3 + y_1$
8: return

For the ECC-ALU unit, the binary method for computing the scalar multiplication

Algorithm 7 ECC-DOUBLE: Double of a point P

Require: $P = (x_1, y_1),, x_1, y_1 \in F_{2^m}$
Require: $a \in F_{2^m}$, a is the constant in the elliptic curve
Ensure: $R = (x_3, y_3) = 2P$
1: **if** $P = O$ **then**
2: $R \leftarrow O$
3: return
4: **end if**
5: $\lambda \leftarrow x_1 + y_1/x_1$
6: $x_3 \leftarrow \lambda^2 + \lambda + a$
7: $y_3 \leftarrow x_1^2 + \lambda x_3 + x_3$
8: return

kP, described in algorith 5, was implemented as a finite sate machine. The finite state machine coordinates two dedicated units that perform the operations ECC-ADD and ECC-DOUBLE according to algorithms 6 and 7 respectively. So, the performance of the ECC-ALU depends strongly on the performance of these dedicated units.

Three different architectures were designed for ECC-ADD and ECC-DOUBLE algorithms in order to select the better ones for the ECC-ALU. The main differences were in the implementation for field arithmetic. The following sections describe each one of these architectures.

Architecture 1

The first architectures for ECC-ADD and ECC-DOUBLE are depicted in figures 3.19 and 3.20 respectively. They are based on a field serial multiplier and a field inverter. Squaring is performed as a multiplication and division is computed as two consecutive operations, one inversion and then a multiplication. All internal buses are m-bit wide.

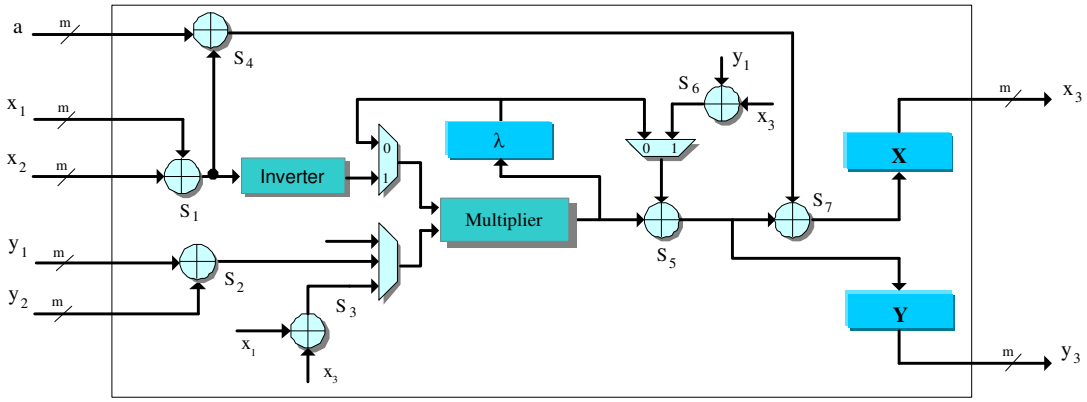

Figure 3.19: Sum of different points, ECC-ADD

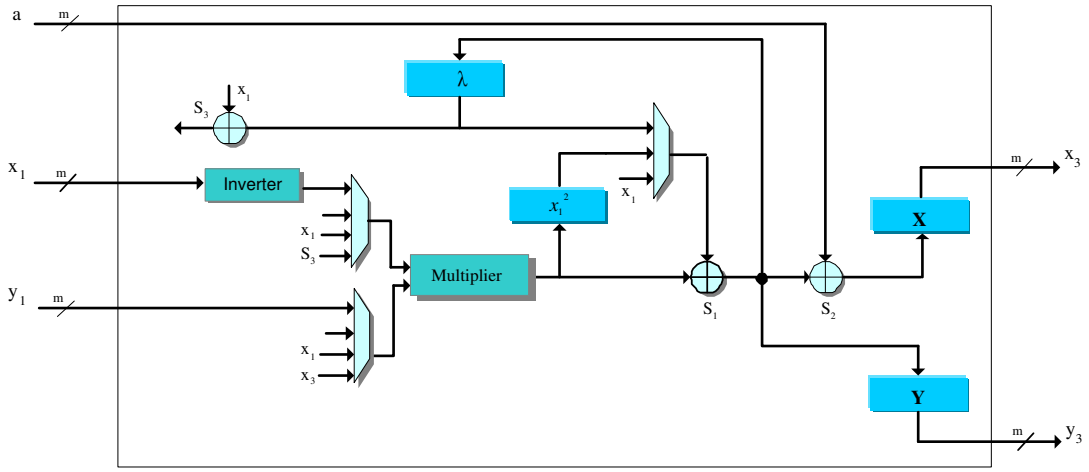

Figure 3.20: Sum of equal points, ECC-DOUBLE

The multiplier implements the serial multiplier in F_{2^m} as described in algorithm 8. The algorithm performs m iterations. In every step, the multiplication $C(x)x$ is performed and the resulting polynomial is reduced modulo $F(x)$. These two consecutive operations can be performed in parallel so, the multiplication is performed in m iterations.

Algorithm 8 Shift and Add method: Multiplication in F_{2^m}

Require: $A(x)$, $B(x) \in F_{2^m}$, $P(x)$ the irreducible polynomial of degree m
Ensure: $C(x) = A(x)B(x) \bmod P(x)$
1: $C(x) \Leftarrow 0$
2: **for** i from $m-1$ down to 0 **do**
3: $\quad C(x) \leftarrow C(x)x + A(x)b_i$
4: $\quad C(x) \leftarrow C(x) + c_m P(x)$
5: **end for**
6: return $C(x)$

The block diagram of the serial multiplier is shown in figure 3.21.

Field inversion is an implementation of the Modified Almost Inversion Algorithm listed in algorithm 9. Latency of the MAIA algorithm is at most ($2m$-1) iterations. In this case, all internal buses are $m+1$ bits long because of the irreducible polynomial is $m+1$-bit long. MAIA is a variant of the Extended Euclidean Algorithm, commonly used to compute inverses in the integer numbers. This algorithm was selected because it is considered less complex than other variants of the Extended Euclidean Algorithm that can be found in [37]. The hardware implementation of algorithm 9 is depicted in figure 3.22.

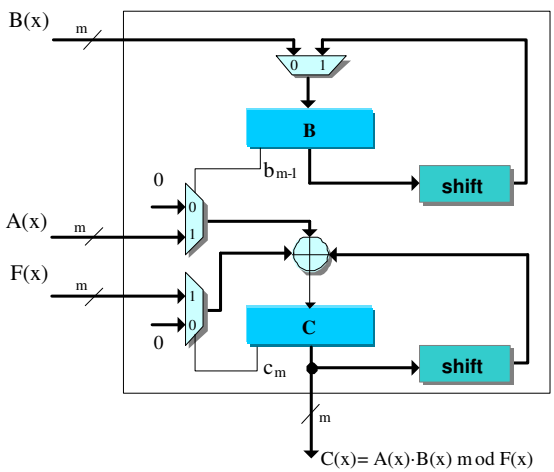

Figure 3.21: Serial multiplier

Algorithm 9 Modified Almost Inverse Algorithm: Invesion in F_{2^m}

Require: $A(x) \in F_{2^m}$, $A(x) \neq 0$ and $P(x)$ the irreducible polynomial of degree m
Ensure: $C(x) = A(x)^{-1} \bmod P(x)$

1: $B(x) \leftarrow 1$
2: $C(x) \leftarrow 0$
3: $U(x) \leftarrow A(x)$
4: $V(x) \leftarrow P(x)$
5: **loop**
6: **while** $U(0) = 0$ **do**
7: $U(x) \leftarrow U(x)x^{-1}$
8: $B(x) \leftarrow (B(x) + x_0 P(x))x^{-1}$
9: **end while**
10: **if** $U(x) = 1$ **then**
11: return $B(x)$
12: **end if**
13: **if** $grade(U(x) < grade(V(x))$ **then**
14: $U(x) \leftrightarrow V(x)$
15: $C(x) \leftrightarrow B(x)$
16: **end if**
17: $U(x) \leftarrow U(x) + V(x)$
18: $B(x) \leftarrow B(x) + C(x)$
19: **end loop**

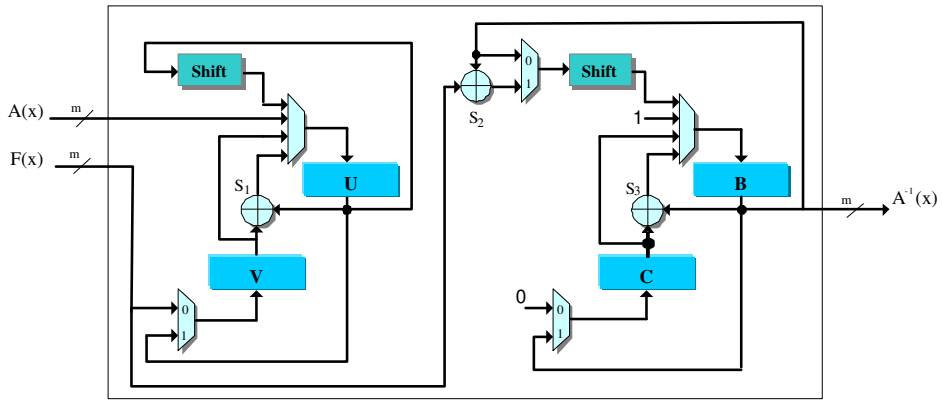

Figure 3.22: Interter for F_{2^m}

Architecture 2

For the second developed architecture, the multiplier was implemented as a digit serial multiplier [44]. The inversion operation was implemented according to the Fermat's algorithm.

The digit serial multiplier is designed according to the following facts: an element $a \in F_{2^m}$, in polynomial basis, is viewed as the polynomial $A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0$. For a positive number $g < m - 1$, the polynomial $A(x)$ can be grouped so that it can be expressed as:

$A(x) = x^{(s-1)g}B_{s-1}(x)+...+x^gB_1(x)+B_0(x)$, where $s = \lceil m/g \rceil$ and $B_i(x)$ is defined as follows:

$$B(x) = \begin{cases} \sum_{j=0}^{g-1} b_{ig+j}x^j & 0 \leq x \leq s-2 \\ \sum_{j=0}^{m \bmod g-1} b_{ig+j}x^j & i = s-1 \end{cases} \tag{3.1}$$

If x^g is factored in equation 3.1, the resulting expression is

$$B(x) = x^g(x^g(\cdots(x^g(x^gB_{s-1}(x) + B_{s-2}(x)) + \cdots) + B_1) + B_0 \tag{3.2}$$

So, using the representation shown in equation 3.2, the field multiplication can be computed according to algorithm 10.

According to [44], the operation 1 and 2 in algorithm 10 can be computed as follows:

$$V_1(x) = x^g \sum_{i=0}^{m-g-1} c_ix^i \tag{3.3}$$

Algorithm 10 Digit-Serial multiplication: Multiplication in F_2^m

Require: $A(x), B(x), F(x) \in F_2^m$
Ensure: $C(x) = A(x)B(x) \bmod F(x)$
1: $C(x) \leftarrow B_{s-1}(x)A(x) \bmod F(x)$
2: **for** k from $s-2$ downto 0 **do**
3: $\quad C(x) \leftarrow x^g C(x)$
4: $\quad C(x) \leftarrow B_k(x)A(x) + C(x) \bmod F(x)$
5: **end for**

$$V_2(x) = x^g \sum_{i=m-g}^{m-1} c_i x^i \bmod F(x) \tag{3.4}$$

$$V_3(x) = B_k(x)A(x) \bmod F(x) \tag{3.5}$$

$$P(x) = V_1(x) + V_2(x) + V_3(x) \tag{3.6}$$

Using the property that $F(x) = x^m + x^d + \cdots + 1$ provides the equivalence $x^m \equiv x^d + \cdots + 1$, $V_2(x)$ becomes

$$V_2(x) = (x^d + \cdots + 1)(c_{m-1}x^{g-1} + \cdots + c_{m-g+1}x + x_{m-g}) \bmod F(x) \tag{3.7}$$

Polynomial $V_1(x)$ consists of the least significant $m-g$ bits of $C(x)$. The operands of $V_2(x)$ have degree d and $g-1$. Often, the irreducible polynomial used is such that $d+g < m$ so modular reduction is not necessary. $V_2(x)$ and $V_3(x)$ are then computed in parallel. The diagram of the architecture for multiplication in F_2^m according to the algorithm 10 is depicted in figure 3.23.

The two main blocks are two combinatorial multipliers, one for computing $V_2(x)$ and the other for $V_3(x)$. The architecture of a combinatorial multiplier is shown in figure 3.24. As mentioned before, the logic to perform modular reduction is not necessary in the computation of $V_2(x)$. The digit-serial multiplier speeds-up the serial multiplier performing a multiplication in F_{2^m} in m/d iterations, where d is the size of the digit. This size affects the latency of the multiplier and also the area complexity of the multiplier.

Different to architecture I, the squaring operation is performed by a customized architecture instead of using a multiplier. The squarer is implemented with pure combinatorial logic so it can be computed in only one clock cycle. The square of an element a represented by $A(x)$ involves a polynomial multiplication and then the reduction modulo

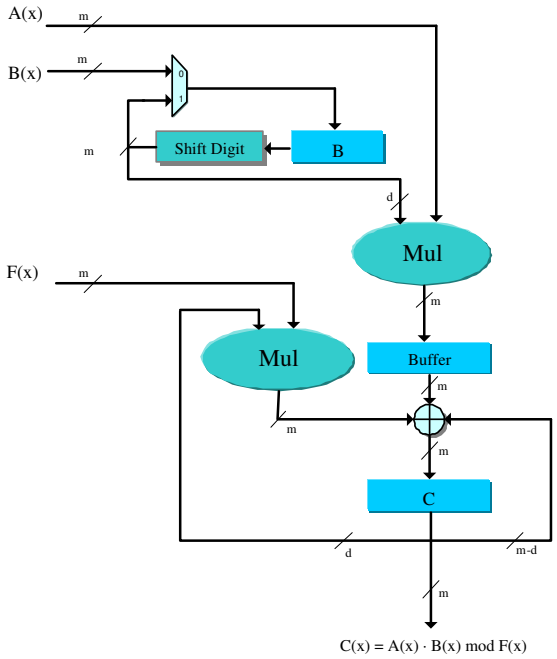

Figure 3.23: Digit serial multiplier for F_{2^m}

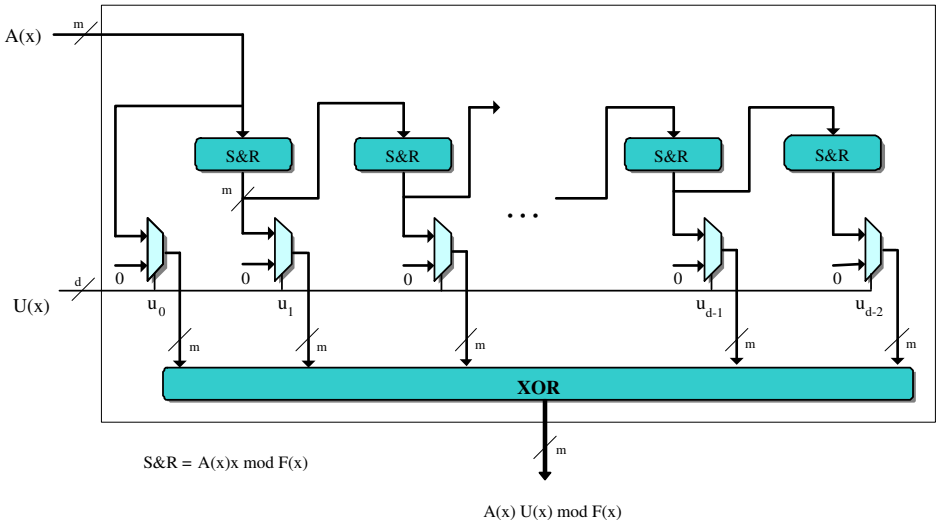

Figure 3.24: Main reduction module for the digit-serial multiplier

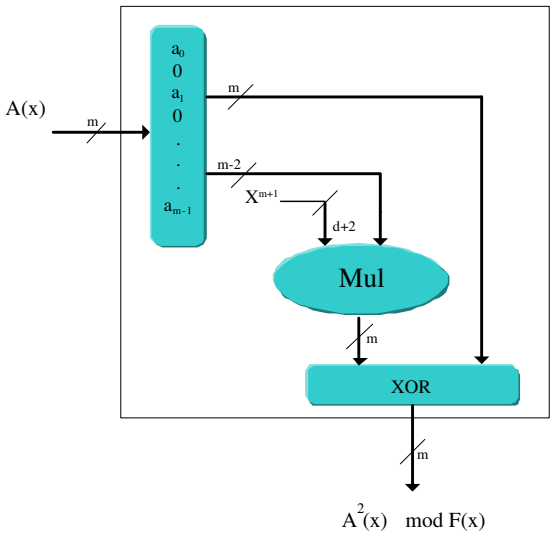

Figure 3.25: Squarer for F_{2^m}

$F(x)$. $A(x)A(x) = A^2(x)$ is given in equation 3.8.

$$A^2(x) = a_{m-1}x^{2m-2} + \cdots + a_2x^4 + a_1x^2 + a_0 \tag{3.8}$$

By factoring x^{m+1}, equation 3.8 can be written as $A^2(x) = A_h(x)x^{m+1} + A_l(x)$, where

$$A_h(x) = a_{m-1}x^{m-3} + \cdots + a_{(m+3)/2)}x^2 + a_{(m+1)/2}$$
$$A_l(x) = a_{(m-1)/2}x^{m-1} + \cdots + a_1x^2 + a_0$$

The degree of $A_l(x)$ is lower than m and reduction is not necesary. The product $A_h(x)x^{m+1}$ may have degree as high as $2m-2$. By multiplying both sides of the field equivalence $x^m = x^d + \cdots + 1$ by x, it is deduced that $x^{m+1} = x^{d+1} + \cdots + x$. So

$$A_h(x)x^{m+1} = A_h(x)(x^{d+1} + \cdots + x)$$

This operation is performed using the architecture for combinatorial multiplication in the digit-serial multiplier. Here, the size of the digit is $d+2$. The diagram of the squarer is depicted in figure 3.25.

The operation $A^2(x)$ results is a polynomial of degree as high as $2m-2$. It is a polynomial with interleaved insertion of '0'. Once $A^2(x)$ is computed, a combinatorial multiplier computes $A_h(x)x^{m+1} \mod F(x)$. Final result is obtained by adding the polynomial $A_l(x)$ to the result of the polynomial multiplication.

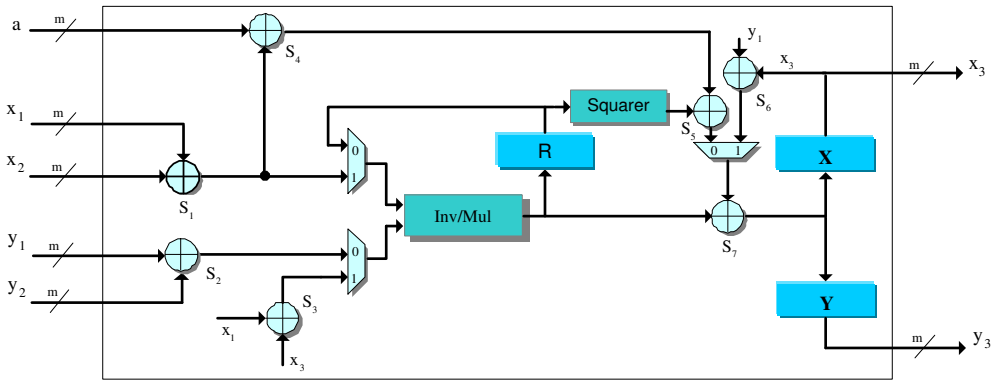

Figure 3.26: Architecture for add operation, ECC-ADD

The inversion operation is performed according the Fermat's algorithm listed in algorithm 11. It states that for every $a \in F_2^m$, $a^{2^m-2} \equiv a^{-1}$.

Algorithm 11 Fermat's Algorithm: Inversion in F_2^m

Require: $a \in F_2^m$
Ensure: $b = a^{-1}$
1: $b \leftarrow a$
2: **for** i from 1 to $m-2$ do **do**
3: $\quad b \leftarrow b^2 a$
4: **end for**
5: $b \leftarrow b^2$

Hardware for algorithm 11 is built on the digit-serial multiplier and the squarer described previously. A finite state machine orchests the correct execution of these two main blocks and computes an field inversion in $m-2$ fixed iterations, each one consuming m/d cycles.

New architectures for ECC-ADD and ECC-DOUBLE using previous basic units are shown in figures 3.26 and 3.27 respectively.

Architecture 3

The third architecture for scalar multiplication keeps the squarer and multiplier in architecture 2, but implements a new division algorithm. Instead of computing the division as two consecutive operations, inversion and the multiplication, division is computed directly. The new algorithm was proposed in [45] and is listed in algorithm 12.

The spatial complexity of algorithm 12 is similar to the one in the Modified Almost Inverse Algorithm implemented in the first architecture. The advantage of the new algorithm is that the multiplication is not necessary, saving a considerable number of

Algorithm 12 Division algorithm: Division in F_{2^m}

Require: $X_1(x), Y_1(x) \in F_{2^m}$, $X_1(x) \neq 0$ and $F(x)$ the irreducible polynomial of degree m

Ensure: $U(x) = Y_1(x)/X_1(x) \bmod P(x)$

1: $A(x) \leftarrow X_1(x)$
2: $B(x) \leftarrow F(x)$
3: $U(x) \leftarrow Y_1(x)$
4: $V(x) \leftarrow 0$
5: **while** $A(x) \neq B(x)$ do **do**
6: **if** x divides to $A(x)$ **then**
7: $A(x) \leftarrow A(x)x^{-1}$
8: **if** x divides to $U(x)$ **then**
9: $U(x) \leftarrow U(x)x^{-1}$
10: **else**
11: $U(x) \leftarrow (U(x) + F(x))x^{-1}$
12: **end if**
13: **else if** x divides to $B(x)$ **then**
14: $B(x) \leftarrow B(x)x^{-1}$
15: **if** x divides to $V(x)$ **then**
16: $V(x) \leftarrow V(x)x^{-1}$
17: **else**
18: $V(x) \leftarrow (V(x) + F(x))x^{-1}$
19: **end if**
20: **else if** grade of $A(x)$ is greater than grade of $B(x)$ **then**
21: $A(x) \leftarrow (A(x) + B(x))x^{-1}$, $U(x) \leftarrow U(x) + V(x)$
22: **if** x divides to $U(x)$ **then**
23: $U(x) \leftarrow U(x)x^{-1}$
24: **else**
25: $U(x) \leftarrow (U(x) + F(x))x^{-1}$
26: **end if**
27: **else**
28: $B(x) \leftarrow (A(x) + B(x))x^{-1}$, $V(x) \leftarrow U(x) + V(x)$
29: **if** x divides to $V(x)$ **then**
30: $V(x) \leftarrow V(x)x^{-1}$
31: **else**
32: $V(x) \leftarrow (V(x) + F(x))x^{-1}$
33: **end if**
34: **end if**
35: **end while**

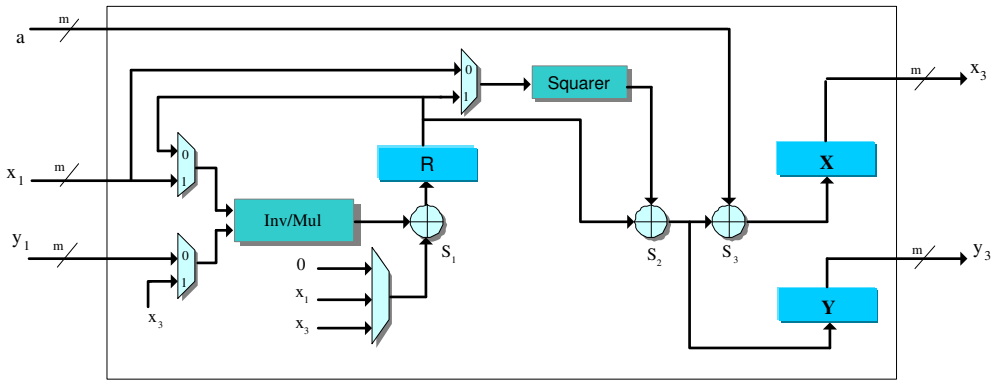

Figure 3.27: Architecture for double operation, ECC-DOUBLE

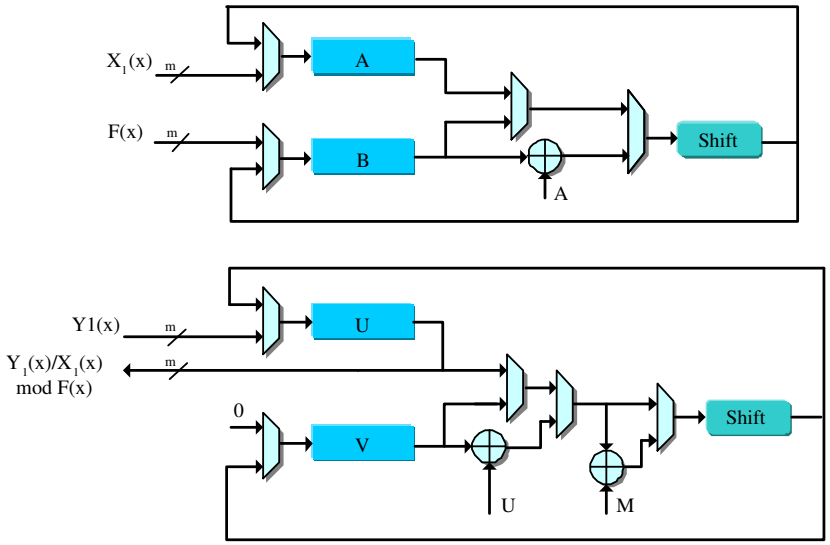

Figure 3.28: Architecture for division in F_{2^m}

cycles in each ECC-ADD or ECC-DOUBLE operation. The implementation of this divider is shown in figure 3.28.

With this new architecture for the division operation, architectures for ECC-ADD and ECC-DOUBLE becomes as depicted in figures 3.29 and 3.30.

The three previous architectures were synthesized and simulated. Architecture 3 lets better timing for computing scalar multiplication. In next chapter such results are reported.

In next section the basic units for the large integer arithmetic modulo n are presented.

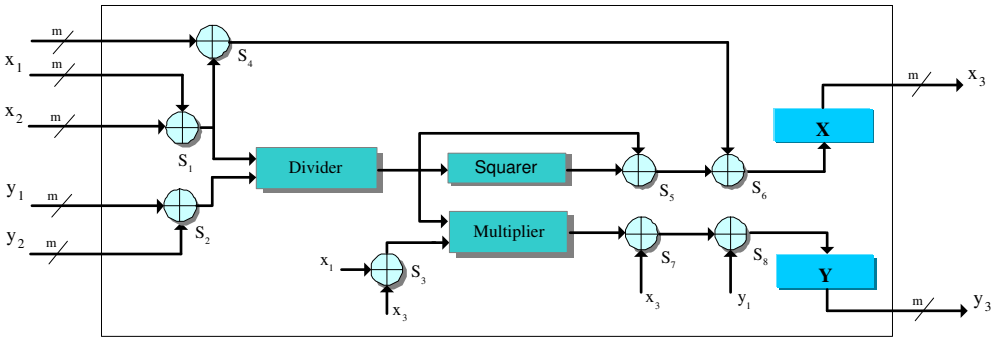

Figure 3.29: Architecture for add operation, ECC-ADD

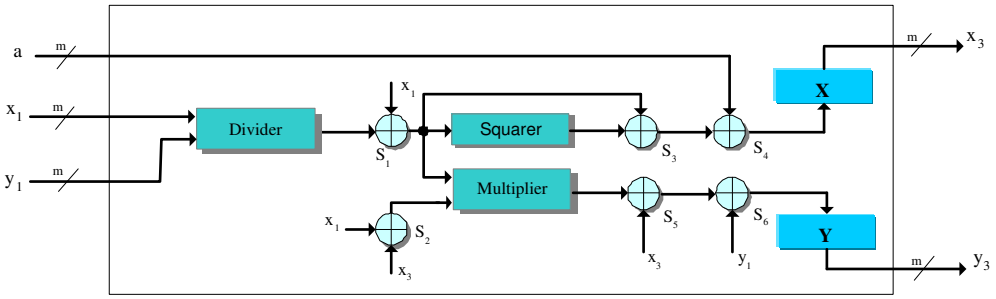

Figure 3.30: Architecture for double operation, ECC-DOUBLE

3.4.2 Big integer arithmetic ALU

Modulo n arithmetic is necessary in ECDSA scheme. Four operations are required: integer inversion, multiplication, sum, and reduction modulo n. However, comparison is required in ECDSA signature verification and ECIES decryption. So, the comparison operation is also included in the arithmetic unit design.

Two main unit form the INT-ALU: A unit to perform multiplication, sum and reduction modulo n, and a dedicated unit for division.

The first unit is an implementation of the Blakley method listed in algorithm 13.

Algorithm 13 Blakley Method: Multiplication for ECDSA

Require: a, $b \in F_2^m$ and n the order of the G point
Ensure: $c = ab \bmod n$
1: $r \leftarrow 0$
2: **for** i from 1 to $m-1$ do **do**
3: $\quad r \leftarrow 2r + a_{k-1-i}b$
4: $\quad r \leftarrow r \bmod n$
5: **end for**
6: return r

The Blakey method performs serial multiplication, that is a variant of the shift and

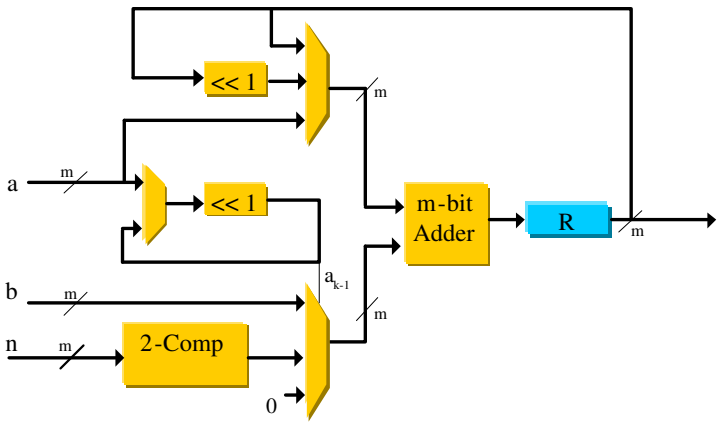

Figure 3.31: Modular multiplier

add scholar method. It computes a multiplication modulo n in m clock cycles with a direct implementation, as it was done in this work. However, the latency is high because of the subtractor and adders that need to be implemented, mainly for performing reduction modulo n. Better clock frequency was achieved by re-using components and separating the modular reduction operation in a separate step. The design for the multiplier is as shown in figure 3.31. By adding extra logic, this architecture can compute a sum or reduction modulo n operation. The multiplication is performed in maximum $3m$ cycles, while the sum and reduction modulo n can be computed in maximum 3 cycles.

The modular divisor is the algorithm 12 applied to integer numbers. This is listed in algorithm 14. The architecture of the divisor is more complex because the modular reduction is implemented by subtracting n to the result.

Two hardware designs were made for algorithm 14. The first one is a direct implementation that performs the division at most in $(2m - 1)$ cycles. In the second architecture, the critical path is reduced. The reduction modulo n is computed in 2 or three cycles, so latency increments in a factor of 3. This improved architecture spends more cycles to compute an integer division modulo n but increments the clock frequency, an important aspect when INT-ALU is integrated with the elliptic curve ALU.

The second architecture for algorithm 14 is shown in figure 3.32. A division operation is completed in maximum $6m - 3$ cycles.

3.5 Chapter summary

The architectures for data processing and arithmetic support to execute the operation in both ECDSA and ECIES were presented, incorporating data compression. Data

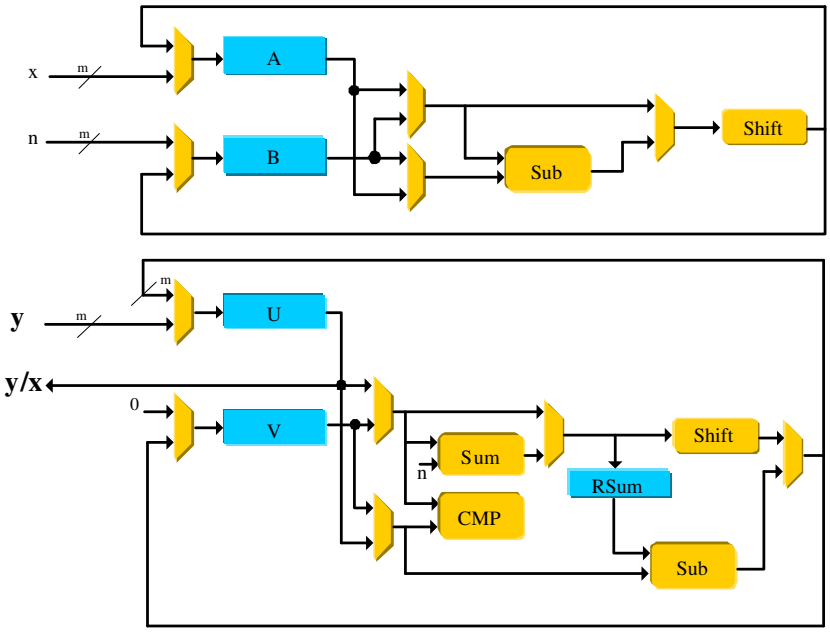

Figure 3.32: Integer divisor *mod n*

processing occurs on the fly, so temporal storage is not necessary. Main modules for data processing can operate receiving data from different sources by implementing a simple communication protocol based on handshaking signals. Arithmetic units give support for both ECDSA and ECIES. The scalar multiplication, which is the most time consuming part in elliptic curve cryptosystems is computed by executing operations ECC-ADD and ECC-DOUBLE in parallel. The architectures for field and large integer arithmetic are generic architectures that can be parameterized for any field order m.

Following chapter presents the synthesis results for architectures presented in this chapter and the performance of the proposed system.

Algorithm 14 Modular Integer Division

Require: $x, y \in [1, M-1]$ and M
Ensure: U, where $y = Ux \pmod{M}$

```
1: A ← x, B ← M, U ← y, V ← 0
2: while A ≠ B do do
3:    if A is even then
4:       A ← A/2
5:       if U is even then
6:          U ← U/2
7:       else
8:          U ← (U + M)/2
9:       end if
10:   else if B is even then
11:      B ← B/2
12:      if V is even then
13:         V ← V/2
14:      else
15:         V ← (V + M)/2
16:      end if
17:   else if A > B then
18:      A ← (A − B)/2, U ← U − V
19:      if U < 0 then
20:         U ← U + M
21:      end if
22:      if U is even then
23:         U ← U/2
24:      else
25:         U ← (U + M)/2
26:      end if
27:   else
28:      B ← (B − A)/2, V ← V − U
29:      if V < 0 then
30:         V ← V + M
31:      end if
32:      if V is even then
33:         V ← V/2
34:      else
35:         V ← (V + M)/2
36:      end if
37:   end if
38: end while
39: Return U
```

Chapter 4

Implementation and Results

This chapter presents the synthesis results for the system implementation and shows comparison results. The architecture was synthesized for a Xilinx VirtexII XC2V6000-4ff1176 FPGA, using the Xilinx ISE 6.x synthesis tools and simulated in Active-HDL 6.2 environment.

The following section shows the synthesis results for the developed system. Firstly the results for the data processing blocks are presented, then the results for the arithmetic units. The parameters of the architectures presented in the previous chapter for data processing were selected from software implementation results. The parameters for the arithmetic units, like finite field, irreducible polynomial and elliptic curve were taken from the recommendations by NIST and SEC group.

4.1 Implementation: data processing blocks

4.1.1 Compression module

The architecture described in Chapter 3 for data compression is mainly dependent on the size of the buffers. The values for the size of the coding and searching buffers determine the size of registers, multiplexers, comparators and counters in the processing elements, and also, the compression ratio and latency. These sizes were selected taking into account the results of a software implementation of the compression algorithm. In this work, the compressor was tested using the large Calgary corpus [46]. The corpus is a data set commonly used for evaluating lossless data compression methods, it is composed of 18 files, which range in size from 21 KB to 768 KB, from source code, object code, among others. The size and data-type of the files are listed in table 4.1, all file sizes in bytes.

The software version of the LZ algorithm was tested using different values for the coding and searching buffers. From this software implementation, it was measured the

Table 4.1: Calgary corpus

Name	Category	Size (bytes)
bib	Bibliography (refer format)	111,261
book1	Fiction book	768,771
book2	Non-fiction book (troff format)	610,856
geo	Geophysical data	102,400
news	USENET batch file	377,109
obj1	Object code for VAX	21,504
obj2	Object code for Apple Mac	246,814
paper1	Technical paper	53,161
paper2	Technical paper	82,199
pic	Black and white fax picture	513,216
progc	Source code in "C"	39,611
progl	Source code in LISP	71,646
progp	Source code in PASCAL	49,379
trans	Transcript of terminal session	93,695

compression ratio, symbols processed in every codification step and timing. In figure 4.1, the average compression ratio on the Calgary corpus is shown.

As shown in figure 4.1, the coding buffer does not need to be large for achieving good compression ratios, but the searching buffer does. The size of the searching buffer limits the compression ratio to an specific value, which is reached as the coding buffer grows. In the graph, the average compression ratio of the Calgary Corpus is reduced by half only if the length of the searching buffer is bigger than 2KB. Although a large searching buffer allows to achieve better compression ratios, it increases the latency of the systolic array. For the compressor design of this work, the number of clock cycles required to obtain a codeword is in the interval $[M,\ M+N]$, where M is the size of the searching buffer and N is the one for the coding buffer.

Other important measurement is the throughput the compressor can achieve. Most of the compression methods use the following equation to estimate it.

$$Throughput(Mbps) = clk\frac{8L}{N+M} \tag{4.1}$$

where clk is the frequency at which the compressor works(cycles per second) and L is the longest match found in every codification step. As commented before, $(N+M)$ is, in the worst case, the latency for finding a codeword. Equation 4.1 gives an estimation of the compression throughput valid only for the best case that happens when the length of the matched strings is equal to the maximum value L. Often, this does not occur.

Figure 4.2 shows the average symbols processed in every LZ codification step for

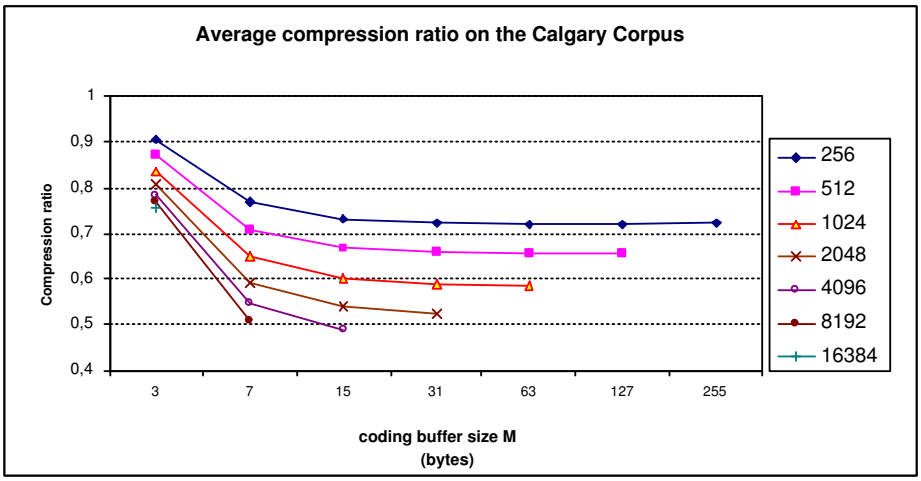

Figure 4.1: Software results of LZ compression

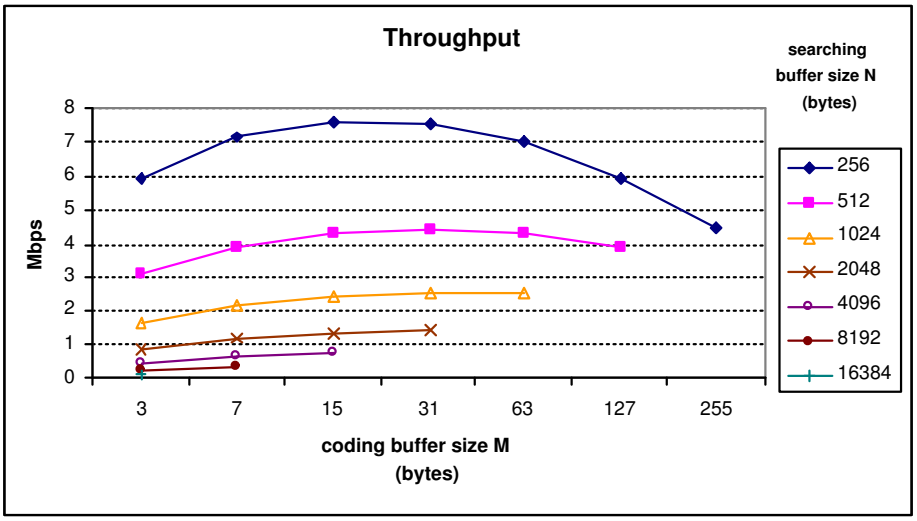

Figure 4.2: Software LZ symbols processed per step in software implementation

Table 4.2: Compressor post-synthesis results

Module	Slices	Max.Delay (ns)
Dictionary	8331(24%)	6.18
Systolic array	316 (1%)	10.88
Type-I PE	5 (1%)	6.46
Type-II PE	5 (1%)	9.49
Control	67 (1%)	9.24
Out interface	35 (1%)	5.87

different sizes of the searching and coding buffers. That is, figure 4.2 shows the average length L of the matched strings in every codification step of the LZ algorithm.

The biggest value for matched strings occurs for a searching buffer of size $N = 4$KB and a coding buffer of size $M = 4$. Just in this case the compressor reaches its maximum throughput.

Based on results showed in previous graphs, it was decided to use a searching buffer of size 1024 bytes long and a coding buffer of 15 bytes. With these values, the compressor can achieve a compression ratio around 0.6 while processing 3.8 symbols at every codification step on average. This leads to a hardware implementation of a systolic array of 14 type-I, 1 type-II processing elements and a buffer of size 1039, 8-bit registers and 1024 8-bit multiplexers. Post-synthesis results for the compressor using these size values are listed in table 4.2.

The compressor occupies 9700 slices what is equivalent to 188,904 logic gates. The dictionary, composed of the searching and coding buffers, is the most area consuming component. The maximum delay in the design is 12.13 *ns* so the compressor works at 82 MHz approximately.

At this frequency, and supposing it is almost constant for different buffer sizes, graph 4.3 shows the throughput of the developed compressor. Every curve in the graph is the evaluation of equation 4.1 taking values for L from figure 4.2.

From figure 4.3, it can be seen that better throughput is achieved if the searching buffer is smaller, opposite to the compression ratio case. For a searching buffer of any size in figure 4.3, throughput reaches its maximum value at an specific value size of the coding buffer. For example, for $N = 256$ and $M = 15$, compressor reaches the maximum throughput of 7.60 Mbps. Throughput decreases as the searching buffer grows. While a selection of $N = 4096$ and $M = 15$ leads to the best compression ratio in average for the Calgary Corpus, the same selection leads to a throughput of 0.74 Mbps which is the worst value in figure 4.3.

These were the results of the compressor synthesized alone. In following section the

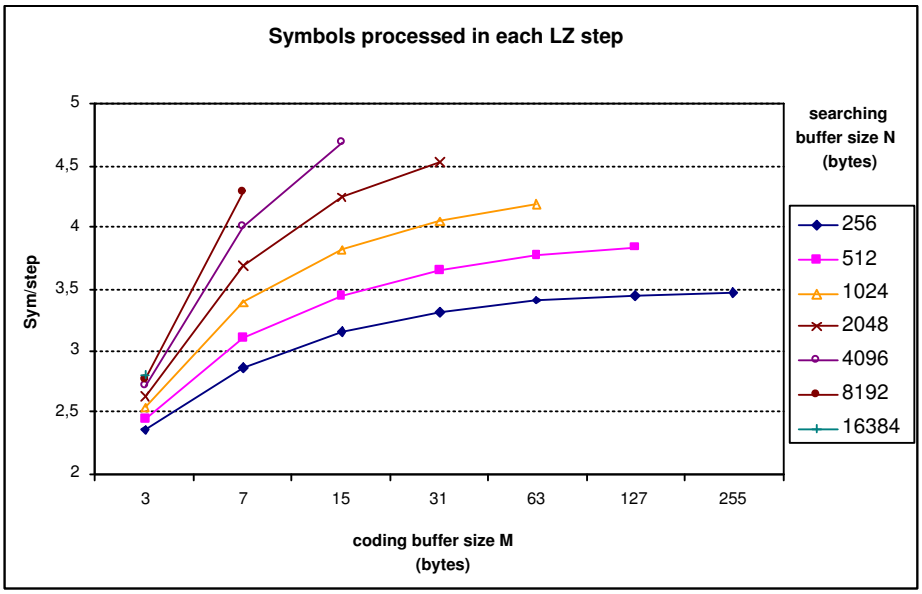

Figure 4.3: Software LZ symbols per step processed

Table 4.3: Synthesis results for HMAC, KDF and E modules

Module	Slices	Max. Delay(ns)
SHA-1 core	676 (2%)	18.47
Pre-processor-SHA	218 (1%)	8.29
KDF	946 (2%)	25.93
HMAC	1339 (3%)	16.28
ENC	203 (1%)	15.61

synthesis results of every module involved in data processing architecture are presented, including the compressor module.

4.1.2 Compressor, HMAC, KDF and E modules

Table 4.3 shows the synthesis results of the main system data processing blocks. The FIFO memories used by the pre-processing module for the SHA-1 core and the 16x32 RAM memory used in KDF, were mapped to block RAM memories (BRAMs) in the FPGA.

The entire architecture for data processing, considering the compressor and the cryptographic primitives, occupied 3 BRAMs and 12,305 slices of 33,792 available (36%). This is equivalent to 441,641 logic gates. The minimum period was 20.273 *ns*, so the design works at 49 MHz approximately.

Table 4.4: Synthesis results for three different ECC-ALUs

Architecture	Slices	Max delay (ns)	Time for kP (ms)
Arch. 1	8,132 (24%)	26.74	1.41
Arch. 2	6,987 (20%)	23.41	26.80
Arch. 3	6,080 (17%)	25.80	4.79

Up to here the synthesis results for data processing were presented. Following section presents synthesis results for the arithmetic units and the random number generator.

4.1.3 Synthesis results for the arithmetic units

For synthesizing the arithmetic units, the $7-tuple\ T = (m, F(x), a, b, G, n, h)$ was parameterized as follows

m	=	163					
$F(x)$	=	$x^{163} + x^8 + x^7 + x^3 + 1$					
G	=	(x, y)					
x	=	0x3	69979697	AB438977	89566789	567F787A	7876A654
y	=	0x4	035EDB42	EFAFB298	9D51FEFC	E3C80988	F41FF883
n	=	0x3	FFFFFFFF	FFFFFFFF	FFFF48AA	B689C29C	A710279B
a	=	0x7	B6882CAA	EFA84F95	54FF8428	BD88E246	D2782AE2
b	=	0x7	13612DCD	DCB40AAB	946BDA29	CA91F73A	F958AFD9
h	Not necessary in ECIES and ECDSA						

These values correspond to a random curve defined on the finite field $F_{2^{163}}$. They were taken from that recommended by the SEC group. A random curve was selected because the arithmetic units were designed to work on any elliptic curve. The order of the field, $m = 163$, does not obeyed to an specific reason, it was just selected because it provides a security level as required according the current state of the art.

The architectures supporting arithmetic operations in both, F_{2^m} and F_n finite fields are parameterized in the order of such fields. n is part of the 7-tuple of the cryptosystem and it is always a prime number minor than the order of the binary field, m. So, all internal buses in the arithmetic units are m-bit wide.

As described in chapter 3, there were designed three different architectures for elliptic curve arithmetic. Table 4.4 shows the synthesis results for the ECC-ALU using these three different architectures. Architecture 3 was the smallest in area and also, the faster to compute a scalar multiplication kP.

Similarly, in table 4.5 are the results for two different modular integer arithmetic

Table 4.5: Synthesis results for two different INT-ALUs

Architecture	Slices	Max delay (ns)
INT-ALU 1	2932 (8%)	57.047
INT-ALU 2	3331 (9%)	48.301

units. The difference is the divider used; ALU1 uses the direct implementation of algorithm 14 which computes the division at most in $(2m-1)$ cycles. The second ALU uses the divider in figure 3.32 which reduces the critical path in algorithm 14 but increases both, latency and area. In ALU2, the division is performed in at most $(6m-3)$ cycles.

ALU2 for integer arithmetic was selected because it is desired a fast arithmetic unit. ALU2 and ARCH3 for elliptic curve arithmetic were integrated in a single architecture. Synthesis results for the final architecture that integrates both, elliptic curve and modular integer arithmetic unit occupies 15 block RAMs, 9,347 slices and it has a critical path of 46.82 *ns*. This is equivalent to 1,136,653 logic gates working at 21.3 MHz. The frequency is slower if compared with results obtained for the processing data architecture, but because they do not share logic, each one can work within its own frequency.

4.2 Timing results

This section presents execution timing results for the cryptographic schemes ECIES and ECDSA adding data compression. The results were obtained by simulating the overall system using the post-synthesis clock frequencies. The architecture was tested using files of the Calgary corpus as data sets. The files were encrypted and decrypted; also signatures were generated and verified. The results were compared with results of the software implementation.

Tables 4.6 and 4.7 show the timing for data processing in encryption and signature generation respectively. In these two operations, data were compressed before the cryptographic operations.

In encryption/signature generation the time for computing the MAC/hash value is similar to the required for compression. According to these results, both the MAC and HASH computations are almost transparent to the compression one. This shows how the proposed architecture outperforms the sequential approach of compressing the entire data and then computing the MAC or HASH value. The extra time given in the last column of table 4.6 indicates the difference in time between the complete

Table 4.6: Timing (ms) for computing the MAC value in data encryption

File	MAC (Encryption)	Time for Compression	Compression ratio	Extra time
obj1	101.866	101.860	0.665	0.006
paper1	247.211	247.205	0.635	0.006
paper3	239.950	239.944	0.680	0.006
paper4	63.948	63.807	0.652	0.141
paper5	54.526	54.519	0.632	0.007
paper6	168.693	168.685	0.618	0.007
progc	160.689	160.683	0.591	0.006
progp	142.607	142.600	0.478	0.006

Table 4.7: Timing (ms) for hashing data in signature generation

File	HASH	Time for Compression	Compression ratio	Extra time
obj1	101.862	101.860	0.665	0.002
paper1	247.207	247.205	0.635	0.002
paper3	239.945	239.944	0.680	0.002
paper4	63.944	63.807	0.652	0.137
paper5	54.522	54.519	0.632	0.003
paper6	168.688	168.685	0.618	0.003
progc	160.685	160.683	0.591	0.002
progp	142.602	142.600	0.478	0.002

Table 4.8: Timing (ms) in a sequential implementation for compressing and encrypting

File	Time for compression	MAC (Encryption)	Total	Time saved
obj1	101.860	1.175	103.035	1.168
paper1	247.205	2.762	249.967	2.756
paper3	239.944	2.591	242.534	2.584
paper4	63.807	0.716	64.523	0.575
paper5	54.519	0.626	55.145	0.619
paper6	168.685	1.931	170.617	1.924
progc	160.683	1.917	162.600	1.911
progp	142.600	1.936	144.537	1.930

Table 4.9: Timing (ms) in a sequential implementation for hashing data

File	Time for compression	HASH	Total	Time saved
obj1	101.860	1.169	103.029	1.167
paper1	247.205	2.756	249.961	2.754
paper3	239.944	2.585	242.528	2.583
paper4	63.807	0.710	64.517	0.573
paper5	54.519	0.620	55.139	0.617
paper6	168.685	1.925	170.611	1.922
progc	160.683	1.911	162.594	1.909
progp	142.600	1.930	144.531	1.929

operation and timing for data compression. In encryption, that time is spend for final SHA-1 computations in the HMAC algorithm after all data are compressed. In signature generation, the same occurs. Extra time is slightly less because no other SHA-1 iteration needs to be performed after data are all read, as occurs in a MAC computation.

Tables 4.8 and 4.9 show the timing for encrypting and signing the same corpus of files but in a sequential implementation. That is, data are compressed first and then, they are encrypted or digital signed. These results demonstrate the transparency of cryptographic operations.

Note how timing for sequential implementation is greater than the approach in this work of processing compressed data on the fly. The difference in time is almost the same that required for encryption/digital signature.

Tables 4.10 and 4.11 show the timing for data processing in decryption and signature verification respectively. The same table shows the timing when data were not compressed in the previous operation (encrypted or signature) and also, the percentage

Table 4.10: Timing (ms) for computing the MAC value in data decryption

File	MAC c/Comp.	MAC s/Comp.	% Time saved
obj1	1.175	1.765	0.666
paper1	2.762	4.346	0.636
paper3	2.591	3.806	0.681
paper4	0.716	1.093	0.655
paper5	0.626	0.984	0.637
paper6	1.931	3.117	0.620
progc	1.917	3.241	0.591
progp	1.936	4.036	0.480

Table 4.11: Timing (ms) for hashing data when verifying a digital signature

File	HASH c/Comp.	HASH s/Comp.	% Time saved
obj1	1.169	1.759	0.664
paper1	2.756	4.340	0.635
paper3	2.585	3.800	0.680
paper4	0.710	1.087	0.653
paper5	0.620	0.978	0.634
paper6	1.925	3.111	0.619
progc	1.911	3.235	0.591
progp	1.930	4.030	0.479

of time saved.

MAC and hash values are computed faster when decrypting or verifying a signature on compressed data. Results showed in tables 4.10 and 4.11 indicate that the percentage of time saved is similar to the compression ratio achieved by the compressor. In the case of a signature verification or data decryption, the system processes on average, 1 byte every 4.5 clock cycles, that is equivalent to process 10 Mbps.

Results shown in the previous tables demonstrate two of the objectives stated in chapter 1. When encrypting data or generating a signature, all the time to process data is spent by the compressor, which is the same time either data is encrypted or not. So, time for encrypting or signing is almost null. When data are decrypted or a signature on them is verified, the improvement in the cryptographic operations is given in the same ratio the data is compressed.

The timing results for cryptographic operations in ECIES and ECDSA are shown in table 4.12. This table assumes a message of 64 bytes and no data compression is

Table 4.12: Timing results (ms) for cryptographic schemes

Operation	Timing
ECIES encryption	5.152
ECIES decryption	2.579
ECDSA signature generation	2.581
ECDSA signature verification	5.161

Table 4.13: Timing comparison for scalar multiplication (co-processors)

Reference	F_q	Platform	Time (ms)
This work	$F_{2^{163}}$	Xilinx XC2V6000	1.41
[34]	$F_{2^{163}}$	Altera EPIF10K250	80
[35]	$F_{2^{113}}$	Amtel AT94K40	10.9
[36]	$F_{2^{270}}$	Xilinx XC4085XLA	6.8
[47]	$F_{2^{163}}$	Xilinx XCV1000	12.3
[48]	$F_{2^{191}}$	Xilinx XCV1000	2.27

applied. It was done in order to compare the results of ECIES and ECDSA with related work. The size of 64 was selected because it is the minimum data block that can be processed in both ECIES and ECDSA.

4.3 Comparison results

Since other similar implementations joining data compression and elliptic curve cryptography have not bee reported, it was not possible to compare the results of this work with similar works. However, the results of the elliptic curve arithmetic unit are compared with some works reported about that. A performance comparison of hardware implementations against each other is not straight forward because of different key size and FPGA technology used for their implementation. In table 4.13, the scalar multiplication timing results are compared with some hardware implementations mentioned earlier in this paper.

Also, the results of ECDSA and ECIES execution against reported software implementations are compared, in all them, the scalar multiplication dominates the execution time. In the reported work, specialized techniques were used, for example, inline functions, loop-unrolling and assembler code.

Table 4.14: Timing comparison for scalar multiplication (processors)

Reference	F_q	Platform	Time (ms)
This work	$F_{2^{163}}$	Xilinx XC2V6000	1.41
[32]	$F_{2^{163}}$	XC2V2000E-7	0.14
[31]	$F_{2^{160}}$	ASIC	0.19
[30]	$F_{2^{167}}$	Xilinx XCV400E-8	0.21
[49]	$F_{2^{155}}$	Xilinx XCV300-4	6.8
[33]	$F_{2^{176}}$	Xilinx XCV300-4	6.9

Table 4.15: Timing (ms) comparison of cryptographic operations with software implementations

Reference	F_q	Encryp.	Decryp.	Sig. Gen.	Sig. Ver.	Platform
This work	$F_{2^{163}}$	5.152	2.579	2.581	5.161	XC2v6000
[38]	$F_{2^{163}}$			2400	900	Palm OS Device
[39]	F_{175}	28.8	13.3	19.1	24.5	PIII 933 MHz
[50]	$F_{2^{163}}$	6.67	4.69	2.64	6.46	PII 400 MHz
[51]	$F_{2^{163}}$			5.7	17.9	Zaurus 206 MHz

Chapter 5

Concluding remarks

This chapter reviews this work including the objectives and contributions and, also the directions of future work in order to improve the performance of the system.

5.1 Summary and contributions

This work discussed the development of a system combining lossless data compression and public key cryptography. For data compression, a dictionary-based method was selected due to the low hardware requirements for implementation and its application to any kind of data. Data compression was integrated to two elliptic curve algorithms: ECDSA and ECIES. The first one has been standardized by ISO, NIST, IEEE, ANSI and other international standard organizations. ECIES is one of the most promising schemes for data encryption/decryption going to be standardized. In this work, in order to outperform a software implementation, parallelization at algorithmic level was applied. Compression was implemented using a systolic array approach; the compression ratio, latency and area complexity depend strongly on the size of the buffers used. The architectures for data processing in both, ECIES and ECDSA were unified in one unit, which computes on the fly the HASH/MAC value of incoming compressed data.

The complete system was described in VHDL and prototyped to a Xilinx VIRTEX II FPGA device. The design is parameterized on the size of the finite filed F_{2^m}, and designed in a modularized manner. This leads to easy further modification, for example, updating specific field or long-integer arithmetic units.

A unified architecture was designed to support the execution of both ECDSA and ECIES algorithms. The same architecture can perform the four cryptographic operations: encryption, decryption, signature generation and signature verification. The system is composed of two independent modules, one for data processing and other for field and long-integer arithmetic. These two units are related by two values; the secret

value, given by the arithmetic module to the data processing one for ECIES execution, and the HASH/MAC value, given by the data processing module to the arithmetic module for ECDSA execution. Data processing and arithmetic modules communicate througt a handshaking protocol, so they can work independently using each one a different clock frequency.

Latency and area complexity in the data processing module are dominated by the searching and coding buffers, that currently are implemented using flip-flops of the FPGA device. The maximum delay in this module is given by the SHA-1 core module, which internally performs a sum operation with five 32-bit operands.

The arithmetic units can support any elliptic curve defined on any binary finite field. Most of the area consumed is for supporting the elliptic curve arithmetic. However, the frequency for the overall system is given by the modular integer ALU, which is the slower module of the system running at 34.39 MHz.

The throughput of the entire system is widely dominated by the throughput of the compressor. Although a systolic array approach allows to implement the compressor with low hardware requirements, the latency was high. For this implementation, the maximum delay for finding a 2-byte codeword was 1039 clock cycles, which leads to achieve only a throughput of 2.4 Mbps for bulk data encryption and signature generation. By compressing data, almost 100% of time in cipher module is saved. For decryption and signature verification, throughput improves to 10 Mbps because data compression is not required, saving the 40% of time in the cipher module.

Because no other similar implementations have been reported joining lossless compression and public key cryptography, performance results are not compared.

The timing results to perform arithmetic operation were acceptable compared with other reported implementation, were sophisticated techniques are implemented (advanced multipliers, code optimization, better but complex algorithms, etc). In this work, the scalar multiplication can be performed in 1.4 ms, for small data sets, scalar multiplication dominates the execution time of the cryptographic operations, so it is important to perform this operation in the least possible time.

5.1.1 Contributions summary

The first contribution of this work was the development of a hardware architecture that combines two of the most important techniques in computer network applications: lossless data compression and public key cryptography. Such combination, and its hardware implementation had not been explored before. The results achieved show that it is possible to consider such combination and propose better implementation in order to improve the throughput of the complete system. The second contribution was

to propose a hardware architecture for executing elliptic curve cryptographic schemes. No such architecture has been reported in the literature.

5.2 Objectives review

The thesis objectives stated in chapter one were achieved satisfactory. The general objective of designing and implementing a system joining public key cryptography and lossless data compression in a single chip was successfully reached. It was possible to combine two different algorithms, applying the operation in the correct order to gather the advantages of transmitting more data, in a secure form using public key cryptography instead of traditionally symmetrical encryption and further more, using a public key cryptosystem considered more secure than commonly used.

Specific objectives were reached, according to the results presented in previous chapter,

- The first objective of improving the performance of the cipher module by compressing data was reached, the time saving for encryption or signature generations was almost of the 100%, all time required is only to perform compression, what is the same either data is encrypted or not. When decrypting data or verifying a digital signature, time to perform these operation reduces in the same ratio as the compression ratio achieved.

- The second objective was to better utilize the available bandwidth by compressing data before encryption. This objective was reached; the percentage of time saved is similar to the compression ratio achieved by the compressor.

5.3 Directions

Directions on this work are for timing improvement in critical modules and hence, improving the throughput of the overall system. First, throughput for encryption/signature generation can be improved by improving the compressor's throughput. The compressor can be implemented by using the CAM approach instead of using the systolic one. Although area resources will increase considerably, up to one symbol per clock cycle will be processed, improving of this manner the throughput for data encryption and signature generation.

For decryption and signature verification, timing will improve by optimum implementation of adders in the SHA-1 core module. A 5 32-bit operand adder determines the critical path in this module that can be implemented by using different adders (i.e. carry save adder).

Timing for elliptic curve arithmetic can be improved following several ways: One option is to implement the Itoh-Tsujii inversion algorithm, which is computed in 8 clock cycles at the cost of complex control. The second is to use projective coordinates and keep the binary method. This eliminates the inversion operation that is the most time consuming in both ECC-DOUBLE and ECC-ADD operations. Field multiplication can be improved if the digit in the multiplier is incremented at the cost of more area. The third option is to implement the Montgomery method [42] and also, use projective coordinates.

Timing for modular integer arithmetic can be improved by better implementation of adders and substractors required in the architectures, using techniques reported in [52] in order to reduce the critical path.

The system can support a protocol to establish the 7-tuple necessary to define the elliptic curve cryptosystem.

List of Figures

Bibliography

[1] D. Salomon, *A Guide to Data Compresion Methods*. NY: Springer-Verlag, 2002.

[2] I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. CA: Academic Press, 1999.

[3] W. Stallings, *Cryptography and Network Security*. NJ: Prentice Hall, 2003.

[4] B. Schneier, *Applied Cryptography*. NY: John Wiley & Sons, 1996.

[5] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, May 1977.

[6] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, 1978.

[7] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, November 1976.

[8] R. Dahab and J. López, "An Overview of Elliptic Curve Cryptography," Tech. Rep. IC-00-10, State University of Campinas, Brazil, May 2000.

[9] K. Compton, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, vol. 34, pp. 171–210, June 2002.

[10] T. Wollinger and C. Paar, "How Secure are FPGAs in Cryptographic Applications?." Cryptology ePrint Archive, Report 2003/119, 2003. `http://eprint.iacr.org/`.

[11] HiFn, Inc, "The first book of compression and encryption." Hifn Whitepaper, available at `http://www.hifn.com/docs/a/The-First-Book-of-Compression-and-Encryption.pdf`.

[12] PKWare, Inc, "Pkware Extends ZIP Standard to Support Strong Encryption." Available at `http://www.pkware.com/news_events/press_releases/2003/060203-appnote.html`.

[13] Proginet, Co, "Product Information Guide." Available at `http://www.proginetuk.co.uk`.

[14] R. Friend and R. Monsour, "RFC 2395-IP Payload Compression Using LZS." Available at `http://rfc.sunsite.dk/rfc/rfc2395.html`.

[15] D. Huffman, "A Method for the Construction of Minimum Redundancy Codes," in *Proceedings of de IRE, 40(9)*, pp. 1098–1101, 1952.

[16] H. Park and V. Prasanna, "Area Efficient VLSI Architecture for Huffman Coding," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 40, pp. 568–575, September 1993.

[17] L. Liu *et al*, "CAM-Based VLSI Architectures for Dynamic Huffman Coding," *IEEE Transactions on Consumer Electronics*, pp. 282–288, August 1994.

[18] N. Ranganathan and S. Henriques, "High-Speed VLSI Designs for Lempel-Ziv-based Data Compression," *IEEE Transactions on Circuits and Systems II*, pp. 96–106, February 1993.

[19] B. Jung and W. Burleson, "Efficient VLSI for Lempel-Ziv Compression in Wireless Data Communication Networks," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 6, pp. 475–483, September 1998.

[20] T. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, vol. 17, pp. 8–19, June 1984.

[21] J. Storer and T. Syzmanski, "Data Compression via Textual Substitution," *Journal of the ACM*, pp. 928–951, June 1982.

[22] S. Hwang and C. Wu, "Unified VLSI Systolic Array Design for LZ Data Compression," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 9, pp. 489–499, August 2001.

[23] W. Huang, N. Saxena, and J. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors, IEEE Sym. on Field Programmable Custom Computing Machines, 2000."

[24] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, November 1987.

[25] V. Miller, "Use of Elliptic Curves in Cryptography," in *Proc. of Advances in Cryptology, CRYPTO'85*, (Santa Barbara, CA), pp. 417–426, August 1985.

[26] J. Pollar, "Monte Carlo Methods for Index Computation mod p," *Mathematics of Computation*, vol. 32, pp. 918–924, 1978.

[27] SEC1, "Elliptic Curve Cryptography: Standards for Efficient Cryptography Group." `http://www.secg.org`.

[28] American Bankers Association, "ANSI X9.62-1998: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)."

[29] NIST, "FIPS 180-2: Secure Hash Standard (SHS)." `http://csrc.nist.gov/publications/fips/`.

[30] G. Orlando and C. Paar, "A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m)," in *Proc. of the Second International Workshop on Cryptographic Hardware and Embedded Systems, CHES'2000*, vol. 1965 of *Lecture Notes in Computer Science*, (Worcester, MA), pp. 41–56, Springer, August 2000.

[31] A. Satoh and K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor," *Transactions on Computers*, vol. 52, pp. 449–460, April 2003.

[32] H. Eberle, N. Gura, and S. Chang, "A Cryptographic Processor for Arbitrary Elliptic Curves over GF(2^m)," in *Proc. of IEEE 14th International Conference on Application-specific Systems, Architectures and Processors,ASAP'2003, The Hague, The Netherlands*, pp. 444–454, June 2003.

[33] T. Kerins *et al.*, "Fully Parameterizable Elliptic Curve Cryptography Processor over GF(2^m)," in *Proc. of 12th International Conference on Field Programmable Logic and Application, FPL'2002*, vol. 2438 of *Lecture Notes in Computer Science*, (Montpellier, France), pp. 750–759, Springer, September 2002.

[34] S. Okada *et al.*, "Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2^m) on a FPGA," in *Proc. of the Second International Workshop on Cryptographic Hardware and Embedded Systems, CHES'2000*, vol. 1965 of *Lecture Notes in Computer Science*, (Worcester, MA), pp. 25–40, Springer, August 2000.

[35] M. Ernest *et al.*, "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n)," in *Proc. of the 4th International Workshop on Cryptographic Hardware and Embedded Systems - CHES'2002*, vol. 2523 of *Lecture Notes in Computer Science*, (Redwood Shores, CA), pp. 381–399, Springer, August 2002.

[36] M. Ernest *et al.*, "Rapid Prototyping for Hardware Accelerated Elliptic Curve Public Key Cryptosystems," in *Proc. of 12th IEEE Workshop on Rapid System Prototyping, RSP'2001*, (Monterey, CA), pp. 24–31, June 2001.

[37] D. Hankerson, L. López, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields," in *Proc. of the Second International Workshop on Cryptographic Hardware and Embedded Systems, CHES'2000*, vol. 1965 of *Lecture Notes in Computer Science*, (Worcester, MA), pp. 1–24, Springer, August 2000.

[38] A. Weimerskirch, C. Paar, and S. Chang, "Elliptic Curve Cryptography on a Palm OS Device," in *Proc. of the 6th Australasian Conference, ACISP'2001*, vol. 2119 of *Lecture Notes in Computer Science*, (Sydney, Australia), pp. 502–513, Springer, July 2001.

[39] E. Konstantinou, Y. Stamatiou, and C. Zaroliagis, "Software Library for Elliptic Curve Cryptography," in *Proc. of 10th Annual European Symposium on Algorithms, ESA'2002*, vol. 2461 of *Lecture Notes in Computer Science*, (Rome, Italy), pp. 625–637, Springer, September 2002.

[40] "Lidia. A Cryptographic Library." Technical University of Darmstadt, Germany. Available from `http://www.informatik.tudarmstadt.de/TI/LiDIA/Welcome.html`.

[41] "Miracl: Big number library." Available at `http://indigo.ie/~mscott/`.

[42] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over $gf(2^m)$ without Precomputation," in *Proc. of the First International Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, pp. 316–327, Springer, 1999.

[43] T. Grembowski *et al*, "Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512," in *Proc. of the 5th International Conference on Information Security*, Lecture Notes in Computer Science, pp. 75–89, Springer, 2002.

[44] J. Lutz, "High Performance Elliptic Curve Cryptographic Co-processor," Master's thesis, University of Waterloo, 2003.

[45] Shantz, S. C., "From Euclid's GCD to Montgomery Multiplication to the Great Divide," Tech. Rep. TR-2001-95, Sun Microsystems Laboratories, 2001.

[46] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for Text Compression," *ACM Computing Surveys*, vol. 21, pp. 557–591, December 1989.

[47] Y. Choi *et al.*, "Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^{163})$ for ECC protocols," in *2002 International Technical Conference on Circuits/Systems Computers and Communications*, (Phuket, Thailand), July 2002.

[48] M. Bednara *et al.*, "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms," in *Proc. of 9th Reconfigurable Architectures Workshop, RAW'02*, (Fort Lauderdales, Florida), April 2002.

[49] K. Leung *et al.*, "Fpga Implementation of a Microcoded Elliptic Curve Cryptographic Processor," in *Proc. of IEEE FCCM 2000*, (Napa Valley), pp. 68–76, 2000.

[50] M. Brown *et al.*, "PGP in Constrained Wireless Devices," in *Proc. of the 9th USENIX Security Symposium*, (Denver, Colorado), pp. 247–262, August, 2000.

[51] I. Riedel, "Security in Ad-hoc Networks: Protocols and Elliptic Curve Cryptography on an Embedded Platform," Master's thesis, Ruhr-Universitat Bochum, 2003.

[52] C. Koc, "RSA Hardware Implementation," Tech. Rep. TR-801, RSA LAboratories, Redwood City, May 1996.