A Security Coprocessor for Next Generation IP Telephony: Architecture, Abstraction, and Strategies

by

Mohamed Abdelfattah Fayed

Bachelor of Science, Computer & System Engineering, Al-Azhar University Egypt, 1995

M. A. Sc., Electrical & Computer Engineering, University of Victoria Canada, 2003

A Dissertation Submitted in Partial Fullfillment of the
Requirements for the Degree of

Doctor of Philosophy

in the Department of Electrical and Computer Engineering

A Security Coprocessor for Next Generation IP Telephony: Architecture, Abstraction, and Strategies

by

Mohamed Abdelfattah Fayed

Bachelor of Science, Computer & System Engineering, Al-Azhar University Egypt, 1995

M. A. Sc., Electrical & Computer Engineering, University of Victoria Canada, 2003

Supervisory Committee

Dr. Fayez Gebali, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. M. Watheq El-Kharashi, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin Fun Li, Department Member
(Department of Electrical and Computer Engineering)

Dr. Issa Traoré, Department Member
(Department of Electrical and Computer Engineering)

Dr. Nedjib Djilali, Outside Member
(Department of Mechanical Engineering)

Supervisory Committee

Dr. Fayez Gebali, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. M. Watheq El-Kharashi, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin Fun Li, Department Member
(Department of Electrical and Computer Engineering)

Dr. Issa Traoré, Department Member
(Department of Electrical and Computer Engineering)

Dr. Nedjib Djilali, Outside Member
(Department of Mechanical Engineering)

Abstract

In this dissertation, four approaches to improve Voice over Internet Protocol (VoIP) security is proposed. The first two approaches are aimed at encrypting/decrypting and authenticating VoIP packets, whereas the last two approaches are aimed at key exchange and user authentication.

For the first contribution, a reconfigurable, high throughput hardware implementation for the different block cipher operational modes is proposed. The proposed architecture is unified; and it combines multiple related functions on the same architecture. In other words, it has the ability to encrypt/decrypt a

plaintext/ciphertext efficiently using different operational modes. Moreover, it has the ability to ensure data integrity using different operational modes. The proposed architecture is tested using the most widely used block ciphers: DES, TDES, AES-128, AES-192, AES-256, and IDEA. The proposed architecture implementation is analyzed and evaluated by comparing it against other implementations.

For the second contribution, a high speed, deep-pipelined architecture for AES algorithm based on the composite field approach targeting VoIP applications is proposed. A new algorithm for finding the *isomorphic* mapping matrix to work for any irreducible polynomial, not only the primitive polynomials, is proposed. Moreover, the modified algorithm is used to find the optimum matrix that gives the minimum delay. The matrix is then used to implement the *SubBytes /InvSubBytes* transformation using composite fields, which in turn allows us to design a very high speed deep-pipelined architecture. As a result of using the optimized matrix, a processing throughput of 49.401 Gbps is achieved, which is twice as fast as the fastest design introduced before. Another feature of this architecture is the separation of the encryption circuit from the decryption circuit to allow concurrent encryption and decryption, which facilitates full duplex encryption/decryption for VoIP applications.

For the third contribution, a high speed, low area ALU to perform field operations required for cryptographic applications is proposed. Although the proposed architecture works for any cryptographic application, an ECC implementation for VoIP applications is targeted. A processor array design space exploration for $GF(2^m)$ multiplier is conducted. This exploration results in different processor array configurations. Among these configurations, the fastest one is chosen since VoIP applications are targeted. The multiplier architecture is then modified to work as a squarer. Based on the multiplier architecture, a unified architecture to calculate addition, multiplication, squaring, and inversion is proposed. The overall area is optimized by using three types of processing elements instead of using a regular

processing element everywhere. NIST-recommended irreducible polynomials is used, which makes our design secure and more suitable for cryptographic applications. The proposed architecture is implemented for $GF(2^{163})$, $GF(2^{283})$, and $GF(2^{571})$ on a Xilinx XC2V4000-6 device to verify the proposed architecture and measure its performance. A maximum frequency of 264 MHz is achieved, which allows the architecture to calculate $GF(2^{163})$ multiplication in 640 ns and inversion in 14.357 μs.

As a fourth contribution, a high speed ECC architecture based on a high-radix *scalar* multiplication is proposed. This architecture is optimized for VoIP applications. First, a new high-radix *scalar* multiplication algorithm is proposed. Then, a merged double-and-add elliptic curve ALU based on the proposed algorithm is designed. The merged double-and-add ALU combines point doubling and adding operations on one architecture, which in turn reduces the critical path delay. The ECC processor utilizes the previously proposed field ALU, which implements addition, squaring, multiplication, and division over $GF(2^m)$. A maximum frequency of 253 MHz is achieved, which allows the architecture to calculate $GF(2^{163})$ *scalar* multiplication for radix 2^8 in 9 μs. At a minimum our results for $GF(2^{163})$, show a speedup ranging from 1.5 to 326 times in comparison to previous FPGA implementations and a speedup ranging from 1.1 to 5.6 times in comparison to previous ASIC implementations.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

3GPP	The 3rd Generation Partnership Project
AES	Advanced Encryption Standard
AES-128	Advanced Encryption Standard with 128 bit key length
AH	Authentication Header
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction Processors
CBC	Cipher Block Chaining Ciphering Mode
CODEC	Coder-Decoder
CTR	Counter Ciphering Mode
DES	Data Encryption Standard
DoS	Denial of Service
DSA	Digital Signature Algorithm
ECB	Electronic Code Book
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ESP	Encapsulated Security Payload
FPGA	Field-Programmable Gate Array
GF	Galois Field
$GF(2^m)$	Binary Galois Field with 2^m elements
GPP	General-Purpose Processors
HMAC	keyed-Hash Message Authentication Code
IAX	Inter-Asterisk eXchange protocol
IDEA	International Data Encryption Algorithm
IETF	Internet Engineering Task Force

IKE	Internet Key Exchange
IMS	IP Multimedia Subsystem
IO	Input Output
IP	Internet Protocols
IPSec	Internet Protocol Security
ISP	Internet Service Provider
ITU	International Telecommunication Union
KB	Kilo Byte
LAN	Local Area Network
LCD	Liquid Crystal display
LUT	Look-Up Table
MAC	Message Authentication Code
MCU	Multipoint Control Units
MGCP	Media Gateway Control Protocol
MITM	Man-In-The-Middle
NAT	Network Address Translation
NIC	Network Interface Card
NIST	National Institute for Standards and Technology
OFB	Output Feedback
PBX	Public Branch eXchange
PC	Personal Computer
PDA	Personal Digital Assistant
PE	Processing Element
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RFC	Request For Comment

RIA	Regular Iterative Algorithm
RSA	Rivest Shamir Adleman
RTCP	Real-time Transport Control Protocol
RTP	Real-time Transport Protocol
S/MIME	Securing Multipurpose Internet Mail Extension
S/WAN	Secure Wide Area Network
SA	Security Association
SCCP	Skinny Client Control Protocol
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SP	Security Parameters
SPIT	SPAM over Internet Telephony
SRTP	Secure Real-time Transport Protocol
SRTCP	Secure Real-time Control Transport Protocol
SSH	Secure Shell
SSL	Secure Socket Layer
TDES	Triple Data Encryption Standard
TLS	Transport Layer Security
TTM	Time to Market
UA	User Agent
UAS	User Agent Server
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VoIP	Voice over Internet Protocols
VOMIT	Voice Over Misconfigured Internet Telephones
WEP	Wired Equivalent Privacy

WiFi	Wireless Fidelity
WPA	WiFi Protected Access

Acknowledgment

All praise be to Allah the Most High, "Who teaches by the pen, Teach the man that which he knew not.", (Quran[96:4, 96:5]). All praise be to Allah the Almighty who has given me knowledge, patience, and perseverance to finish my Ph.D. dissertation. I would like to thank my parents who listened to my fears and worries, encouraged me, and gave me their love and support. I would like to thank my wife, my son, and my daughter for their love, continuous support, and for being friends that I cherished.

I would like to express my deepest thanks and appreciation to my supervisor, Dr. Fayez Gebali, who gave me the opportunity to work in his research group and was an example for me to follow. My time at the University of Victoria was a great success because of his personal and technical support. I greatly valued the sincere and generous moral and financial support he has provided me.

I would like to express my deepest thanks to my supervisor Dr. M. Watheq El-Kharashi for his invaluable scholarly advice, inspiration, help, and guidance that helped me through my Ph.D. dissertation works.

I would like to acknowledge the advice and support from my supervisory committee members: Dr. Issa Traoré, Dr. Kin Fun Li, and Dr. Nedjib Djilali to make my dissertation complete and resourceful.

There are no research works in isolation. I would like to give special thanks to Muhammad Nadzir Marsono for his invaluable support and scholarly discussions. I would like to thank my colleagues: Ahmed Awad, Ahmad Abdullah, Abdessalam Amer, Haytham El-Miligi, Khalid Khayyat, Mohamed Yasein, and many others that I am sure I will regret not having them included in this section.

Dedication

To my beloved parents who encouraged me,
and gave me their love and support.

To my loving wife & kids.

Chapter 1

Introduction

Unlike the traditional public switched telephone networks (PSTN) [1–3], Voice over Internet Protocol (VoIP), also known as IP Telephony, Internet telephony, or Digital Phone, is the transmission of voice conversations over IP networks. Although end-to-end VoIP could be established using Personal Computers (PCs), the term VoIP refers to an equipment, which lets the user dial numbers to communicate with another party who has a VoIP phone or traditional PSTN phone [4].

VoIP has become one of the most important emerging trends in telecommunications. Being a new technology, VoIP introduces both merits and problems. The most serious drawback of VoIP, as compared to traditional PSTN phones, is the security risks, since VoIP has a completely different architecture than traditional PSTN networks [5]. VoIP uses shared medium as opposed to a dedicated circuit for each telephone session in PSTN. System administrators might mistakenly assume that since digitized voice travels in packets, they can simply plug VoIP components into their already-secured networks and get stable and secure VoIP networks. Unfortunately, the tools that are used to secure today's computer networks (such as firewalls, network address translation (NAT), and encryption) do not provide

sufficient security for VoIP networks [6]. Moreover, they increase the propagation delay, which in turn affects the VoIP quality [6].

1.1 Research Motivation

The current VoIP security solutions are insufficient to mitigate the convergence of VoIP security threats. The rapid increase of VoIP users increases the demand for securing VoIP terminals by using a dedicated solution specially designed for that purpose.

1.1.1 Future of VoIP

Today, all major phone service providers are incorporating VoIP technology into their networks. British Telecom, the dominant carrier in the United Kingdom, plans to convert its entire infrastructure to VoIP by 2009 [7]. In the USA, the leading VoIP provider, Vonage, has about 600,000 customers with 15,000 new customers every week [7]. It is expected by 2009 that the number of USA residential VoIP customers will be more than 27 million subscribers [8]. Figure 1.1 shows the expected USA residential VoIP subscribers count in the next few years [8]. In Japan, 4 million customers (10% of all homes) are using VoIP phones. Soon, even when calls originate and terminate with traditional telephone technology, they will be carried over the VoIP infrastructure of phone companies. In 10 years, or even sooner, the global telephone system will run largely on Internet technology [7].

1.1.2 VoIP Threats and its Security Requirements

VoIP suffers many threats such as eavesdropping, service theft, and toll fraud [6]. In addition to the well-known threats for data networks, VoIP faces threats

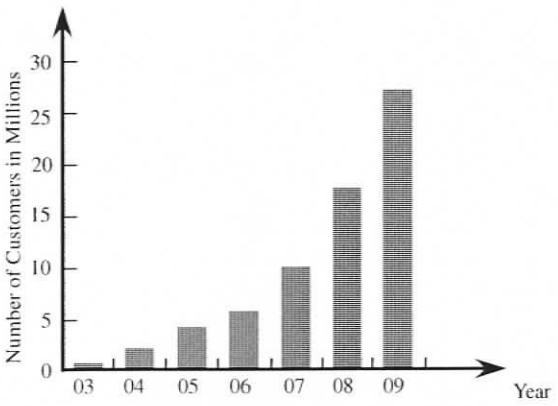


Figure 1.1: USA residential VoIP subscriber.

different than other Internet applications, which trigger unique security and privacy requirements [9]. Therefore, security is one of the essential requirements in VoIP networks. VoIP security is needed to ensure the privacy of conversation, the integrity of transmitted data, and the protection of sensitive data from being accessed by unauthorized recipients [6]. Because VoIP is a new technology, no attacks have been reported so far. It is only a matter of time before we see voice spam on VoIP systems, along with viruses, worms, and other security breaches [7].

Most VoIP solutions do not provide secure transmission [6]. Therefore, it is easy to eavesdrop on VoIP calls and even change their content. There are several tools such as VoIPong or Voice Over Misconfigured Internet Telephones (VOMIT) that collect VoIP packets and reconstruct them in the form of audio files [10–12]. Although VoIP security could be achieved by a proprietary CODEC, it is not applicable because once the CODEC is compromised it becomes insecure. Moreover, using such codes is hard to standardize. Some vendors also use compression to make eavesdropping more difficult. However, real security requires encryption and cryptographic authentication, which are usually not available at the consumer level [6].

One of the most important challenges for securing VoIP is the Quality of Service (QoS). The deployment of security techniques (such as encryption and authentication) can cause a severe deterioration in QoS. These complications range from delaying or blocking calls to encryption delay and delay variation (jitter). Because VoIP is a real-time application, it is delay-sensitive and has low tolerance for disruption and packet loss. Therefore, many security techniques implemented in traditional data networks are not directly applicable to VoIP in their current form [6]. For example, firewalls, intrusion detection systems, and other components must be redesigned for VoIP.

Techniques such as firewalls can help keep intruders from attacking a network. However, firewalls are no defense against an internal attacker. Another layer of defense is necessary at the protocol level to protect the voice traffic. In VoIP, as in data networks, this can be accomplished by encrypting the packets at the IP level using Internet Protocol Security (IPSec), Secure Socket Layer (SSL), and Transport Layer Security (TLS) [13, 14], or at the application level with the secure real-time transport protocol (SRTP) [15]. However, several factors, including the expansion of packet size, ciphering delay, insufficient throughput in the cryptographic engine itself can cause an excessive amount of delay in VoIP packet delivery. This leads to degraded voice quality, again highlighting the tradeoff between security and voice quality, and emphasizing the need for speeding up the cryptographic engine to maintain the required voice quality [1, 6].

We can classify VoIP threats into four classes:

1. Threats related to privacy and confidentiality, which can be eliminated by encrypting VoIP packets using a suitable algorithm such as AES, DES, or TDES.

2. Threats related to data integrity, which can be eliminated by authenticating VoIP packets using suitable hashing algorithms (such as HMAC) or an encryption algorithm (such as AES, DES, or TDES) along with a suitable

message authentication code (MAC) mode such as CBC-MAC.

3. Threats related to user authentication, which can be eliminated by user authentication techniques by using suitable hashing algorithms such as HMAC.

4. Threats related to service disruption, which can be eliminated by using a firewalls. This type of threats will not be discussed this research work.

Table 1.1: VoIP threats and their solutions.

Threat	Solution
Eavesdropping	Encrypt transmitted data using encryption mechanisms like SRTP or IPSec.
Service theft	Use strong user authentication
Toll fraud	Use strong user authentication
Registration hijacking	Use strong user authentication
Call integrity compromise	Use message authentication
Proxy impersonation	Use strong user authentication
Message tampering	Use message authentication
Session tear down	Use message authentication
Denial-of-service (DoS)	Configure devices to prevent such attacks, i.e., using firewalls
SPAM over Internet Telephony (SPIT)	Configure devices to prevent such attacks, i.e., using firewalls

Table 1.1 summarizes the VoIP threats and solutions in order to secure VoIP networks. The result of our study shows that most VoIP threats are due to a lack of encryption/decryption, data integrity, or user authentication. Therefore, to secure VoIP there is a need for encryption/decryption, data authentication, and user

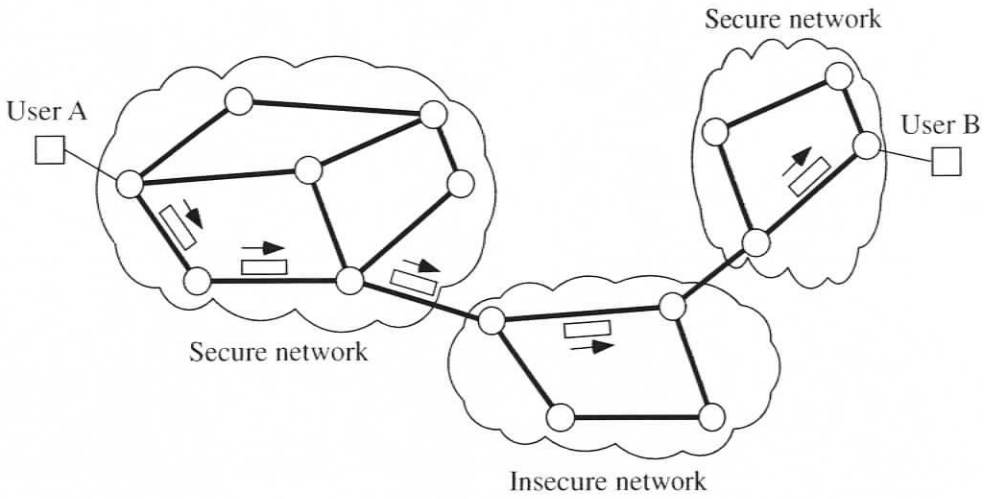


Figure 1.2: A typical VoIP network. Circles represent routers whereas squares represent any system connected to the Internet.

authentication deployment. In order to achieve these requirements a key management approach should be used to share the secret key required for these processes. These requirements should be developed while maintaining the minimum acceptable QoS.

1.1.3 Need to Secure VoIP at Endpoints instead of Routers

Figure 1.2 shows a typical VoIP network, where VoIP packets travel through secure and insecure networks.

Encrypting VoIP packets at routers increases the queuing bottlenecks since routers are built using GPPs, which have limited throughput and limited computational power [16]. Also, as VoIP packets travel through the Internet, they pass through third-party networks, which are not under the control of end users. Therefore, it is highly recommended to perform encryption/decryption solely at endpoints instead of at routers to overcome these problems [6, 17]. One consideration

with this method is that the endpoints must be computationally powerful enough to handle encryption mechanisms. However, endpoints are typically less powerful than gateways. Therefore, a hardware acceleration to increase endpoints computational power is highly required [4]. Moreover, endpoints must have high throughput to handle real-time VoIP encryption. Therefore, there is a need to perform end-to-end security instead of per-hob security, which justifies the demand for fast and computationally powerful IP phones to secure VoIP transmission.

1.1.4 Need for Speed

The two most effective ways to secure networks are firewalls and traffic encryption/decryption. However, they are also two of the greatest contributors to network congestion and delay. Therefore, to reduce the extra delay resulting from firewall and traffic encryption/decryption these processes require speeding up. Because VoIP is a real-time application, the deployment of traditional firewalls and encryption products into a VoIP network is not feasible, particularly when VoIP is integrated into existing data networks. Instead, VoIP security should be implemented by high speed, QoS-aware equipment.

VoIP delay is the time required for a voice transmission to go from its source to its destination. Ideally, delay should be kept as low as possible, but there are acceptable lower bounds on VoIP delay. The International Telecommunication Union (ITU) standard G.114 recommends that the upper bound is 150 ms for one-way traffic [18, 19]. This corresponds to the current delay bound experienced in domestic calls across PSTN lines [18,19]. Therefore, VoIP calls must achieve the 150 ms bound to successfully supply the QoS that PSTN phones provide. This time constraint leaves a very tight margin for error in packet delivery and the amount of security that can be added to a VoIP network. The sources of delay are: voice encoding, packetization,

queuing/buffering, serialization, network delay and jitter buffer. The encoding of voice data can take from 1 to 30 ms [16], whereas voice packet traveling across North America takes up to 100 ms [6, 16], which leaves 20-50 ms for queuing and security implementation. Table 1.2 lists another estimate of these delays for CODEC G.729 according to [20]. The estimates of these delays can tell us the amount of time left for the security implementation. The total of these delays ranges from 121 ms + network transmission delay [20] to 146 ms [19]. Moreover, delay is not only introduced in the endpoints of the system, but also each hop along the network introduces a new queuing delay and processing delay [6]. As a result, VoIP packets have many sources of delay, which leave a very tight margin for security process. Therefore, there is a need for high speed encryption coprocessors to secure VoIP packets without degrading the QoS of VoIP.

1.1.5 Soft-phones Risks

Soft-phone systems, which implement VoIP using an ordinary PC with headsets and special software, should not be used where security or privacy is a concern. Malicious softwares such as worms and viruses are extraordinarily common on PCs connected to the Internet and are very difficult to defend against. These malicious softwares could be easily installed by exploiting the well-known vulnerabilities found in the operating systems and browsers. These vulnerabilities result in unacceptably high risks in the use of soft-phones for most applications [21, 22]. Moreover, soft-phones have a number of challenges regarding the QoS issue (i.e., packet header expansion due to encryption/decryption). These challenges are not all solved at this time [23].

Table 1.2: Delay budget for VoIP packet transmission according to [20].

Delay source (G.729)	Budget (ms)
Device sample capture	0.1
Encoding delay (CODEC G.729)	17.5
Packetization/Depacketization	20
Output queuing delay	0.5
Up link transmission delay	10
Backbone network transmission delay	NTD
Down link transmission delay	10
Input queuing delay	0.5
Jitter buffer	60
Decoder processing delay	2
Device playout delay	0.5
Total	121 + NTD

1.1.6 Importance of Hardware Solutions for VoIP

Traditionally, the low-level networking functions are implemented in hardware to guarantee higher data rates, while high level functions (like those in higher layers) are implemented in software, which provides a high level of flexibility. In general, the most common approach for hardware implementation is designing Application Specific Integrated Circuits (ASICs). For software implementation, the most common approach is using GPPs. However, these two approaches are extremes in terms of speed and flexibility. Although ASICs are very fast, they are also rigid designs. GPPs, on the other hand, provide more flexibility but at the cost of processing speed [24]. Application Specific Instruction Processors (ASIPs) are used to narrow

the gap between the above two extremes (i.e., GPPs and ASICs). ASIPs have some features that make them, overall, preferable over other approaches, including [24]:

Performance: Applications are performed at wire speed because ASIPs are specialized hardware.

Flexibility: Since ASIPs are specialized processors that run software applications, it is easier to adapt to new applications and standards than to design new hardware.

Fast TTM (Time to Market): Since the ASIP hardware is built for specific applications (i.e., security protocols), any modification in these applications need only software modification. As a result, ASIP have fast time to market because design of software is faster and cheaper than hardware design.

Power: ASIPs consume less power compared to other architectures.

Parallelism Since in GPP instructions are executed in a sequential way, a highly parallel architecture can be designed by using ASIPs to achieve higher performance compared to software.

Recently, a kind of ASIP called a VoIP Chip emerged in the VoIP industry. VoIP chips are designed to work as normal phones but using IP networks instead of PSTN. Unfortunately, these VoIP chips do not provide the required security for VoIP applications. The use of ASIPs in VoIP security is of great importance, since security algorithms and protocols need a lot of processing power and, at the same time, needs fast processing because it is a real-time application. Therefore, we have to enhance the security processing without affecting the QoS by utilizing ASIPs.

1.2 Problem Statement

According to the study of the threats and their possible solutions, the following requirements have been identified to secure VoIP transmission:

1. Encryption algorithm,
2. Unified architecture for block cipher operational modes, which can be used to do encryption/decryption and message authentication (i.e., data integrity),
3. User authentication approach, and
4. Key management approach.

VoIP threats have led to a growing demand for effective VoIP security coprocessors, which involves encryption/decryption, message authentication, user authentication, and key management. Both encryption/decryption and message authentication require a block cipher (e.g. AES), which works in different operational modes (e.g. ECB, CBC, and CTR). Therefore, VoIP security coprocessors require a unified architecture to implement the required modes of operation on one architecture, since implementing each mode in a separate architecture will require lots of area and power.

Since VoIP is very sensitive to delay, securing VoIP requires a real-time suitable block cipher, which provides high throughput and low delay and is hard to break. The most attractive and suitable encryption algorithm to secure VoIP applications is the Advanced Encryption Standard (AES) [25]. The National Institute for Standards and Technology (NIST) selected the Rijndael algorithm as the official AES in 2001 [25]. Later, in 2005 NIST recommended securing VoIP using AES [6]. The AES algorithm has a simple structure and can be implemented efficiently on a wide range of general purpose processors. Although a software realization of the AES cipher can achieve

high throughput when compared to other block ciphers, a hardware implementation provides physical security and high speed. Therefore, a hardware implementation is desirable in many special purpose applications where speed and security are important. VoIP phones, video conferencing equipments, and network cryptographic coprocessors are examples where the primary concern is the speed of implementation rather than area and power requirements.

VoIP also requires key exchange and user authentication protocols (e.g. digital signature algorithms (DSA)). These protocols must be suitable for VoIP. Elliptic curve cryptography (ECC) can be used for both key exchange and user authentication [26, 27]. ECC is very attractive because of its smaller key-length and increased theoretical robustness (there is no known sub-exponential algorithm to solve the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is the foundation of ECC [28]). Its relatively small key-size (a 160-bit key provides the same security as a 1024-bit RSA modulus [28–30]) is a major advantage for devices with limited hardware resources, such as smartcards, cell phones, or Personal Digital Assistants (PDAs).

Since ECC algorithm requires basic Galois field operations such as addition, multiplication, squaring, and inversion, there is a high demand to implement these operations on a single field ALU. This field ALU must provide low latency and high throughput to be suitable for VoIP implementations.

Finally, VoIP requires an efficient IP layer security coprocessor, which combines encryption/decryption, message authentication, user authentication, and key management in the same architecture.

1.3 Contributions

After studying VoIP threats carefully, we identified their causes and possible solutions for every threat. Then, we identified the possible algorithms, which could be used as

solutions. For each possible set of solutions we analyzed their performance and chose the best solution out of them.

VoIP security coprocessor requires fast cryptographic modules. To narrow down the scope of this research, a unified architecture for block cipher operational modes, which can be used to do encryption/decryption and message authentication (i.e., data integrity), is required to secure VoIP. To allow fast encryption/decryption, we propose an AES high speed architecture, which is deeply pipelined and utilizes optimum *isomorphic* mapping parameters for *SubBytes* transformation. For fast key exchange and fast user authentication, we propose a high-speed, high-radix architecture for elliptic curve cryptography over $GF(2^m)$. The proposed ECC architecture is based on a proposed processor array field ALU. The contributions of these dissertation are given below.

First, we propose a reconfigurable, high throughput hardware implementation for the different block cipher operational modes. The proposed architecture is unified; it combines multiple related functions on the same architecture. In other words, it has the ability to encrypt/decrypt a plaintext/ciphertext efficiently using ECB, CBC, CFB, OFB, and CTR operational modes. Moreover, it has the ability to ensure data integrity using CBC-MAC, CCM, or RMAC operational modes. Our proposed architecture was tested using the most widely used block ciphers: DES [31], TDES [31], AES-128 [25], AES-192 [25], AES-256 [25], and IDEA [32]. We synthesized these block ciphers using a Xilinx XC2V4000-6-bf957 FPGA [33]. In order to evaluate the proposed architecture and measure its performance, we synthesized an architecture that implements only the ECB. In comparison with the architecture that implements only the ECB mode, our proposed architecture requires extra area ranging from 1.6% for AES-256 to 18.4% for DES and extra delay ranging from 0% for DES to 7% for IDEA. Moreover, the proposed architecture implementation was analyzed and evaluated by comparing it against other implementations [34–36]. This work has

been published in brief in [37], and is submitted in full to [38].

Secondly, we propose a high speed, deep-pipelined architecture for AES algorithm based on the composite field approach targeting VoIP applications. We modified the algorithm in [39] for finding the *isomorphic* mapping matrix to work for any irreducible polynomial, not only the primitive polynomials. Moreover, we used the modified algorithm to find the optimum matrix that gives the minimum delay. We used this matrix to implement the *SubBytes/InvSubBytes* transformation using composite fields, which in turn allows us to design a very high speed, deep-pipelined architecture. As a result of using the optimized matrix, we achieved a processing throughput of 49.401 Gbps, which is twice as fast as the fastest design introduced before [34, 40–44]. Another feature of this architecture is the separation of the encryption circuit from the decryption circuit to allow concurrent encryption and decryption, which facilitates full duplex encryption/decryption for VoIP applications. This work has been published in brief in [45], and is submitted in full to [46].

Thirdly, we propose a high speed, low area ALU to perform field operations required for cryptographic applications. Although the proposed architecture works for any cryptographic application, we target an ECC implementation for VoIP applications. We conducted a processor array design space exploration for $GF(2^m)$ multiplier. This exploration resulted in different processor array configurations. Among these configurations, we chose the fastest one, since we are targeting VoIP applications. We modified the multiplier architecture to work as a squarer. Based on the multiplier architecture, we propose a unified architecture to calculate addition, multiplication, squaring, and inversion. We optimized the overall area by using three types of processing elements (PEs) instead of using a regular PE everywhere. We used NIST-recommended irreducible polynomials, which made our design secure and more suitable for cryptographic applications. The proposed architecture is implemented for $GF(2^{163})$, $GF(2^{283})$, and $GF(2^{571})$ on a Xilinx XC2V4000-6 device

to verify the proposed architecture and measure its performance. We achieved a maximum frequency of 264 MHz, which allowed the architecture to calculate $GF(2^{163})$ multiplication in 640 ns and inversion in 14.357 μs. This work is submitted in brief to [47] and is submitted in full to [48].

Finally, we propose a high-speed ECC architecture based on a high-radix *scalar* multiplication targeting VoIP applications. First, we propose a new high-radix *scalar* multiplication algorithm. Then, we designed a merged double-and-add elliptic curve ALU based on the proposed algorithm. The merged double-and-add ALU combines point doubling and adding operations into one architecture, which in turn reduces the critical path delay. The merged double-and-add ALU calculates $2^wQ + iP$ in a single step. The ECC architecture utilizes the previously proposed field ALU, which implements addition, squaring, multiplication, and division over $GF(2^m)$. We achieved a maximum frequency of 253 MHz, which allows the architecture to calculate $GF(2^{163})$ *scalar* multiplication for radix 2^8 in 9 μs. Our worst case results, for $GF(2^{163})$, showed a speedup that ranges from 1.5 to 326 in comparison to previous FPGA implementations and a speedup ranges from 1.1 to 5.6 in comparison to previous ASIC implementations. This work is submitted in brief to [49] and is submitted in full to [49].

1.4 Dissertation Organization

This dissertation is organized as follows.

In Chapter 2, we review the necessary background, including VoIP systems and VoIP security requirements. Our discussion on the VoIP system includes protocols, architectures, and equipment classifications. Our discussion on VoIP security is divided into two separate topics: VoIP threats and their solutions.

Chapter 3 proposes a novel multi-mode architecture, which supports the

ciphers' specified operation sets in only one hardware architecture. Using a common architecture, the implementation cost is examined for the most widely used block cipher: DES, TDES, AES-128, AES-192, AES-256, and IDEA. The shown experimental results show that the additional implementation costs of all the studied block ciphers are very low for the proposed multi-mode architecture. Therefore, the proposed architecture's additional costs are negligible compared with its huge advantages.

Chapter 4 proposes a high speed deep-pipelined architecture for AES algorithm based on the composite field approach targeting VoIP applications. We modified the algorithm in [39] for finding the *isomorphic* mapping matrix to work for any irreducible polynomial, not just the primitive polynomials. Moreover, we used the modified algorithm to find the optimum matrix that gives the minimum delay. We used this matrix to implement the *SubBytes /InvSubBytes* transformation using composite fields, which in turn allowed us to design a very high speed deep-pipelined architecture.

Chapter 5 proposes a high-speed, low-area ALU to perform field operations required for cryptographic applications. The proposed ALU is processor array based, which is optimized for speed to be suitable for VoIP applications. Although the proposed architecture works for any cryptographic application, we target an ECC implementation for VoIP.

Chapter 6 proposes a novel, area-speed efficient, high-speed ECC architecture over $GF(2^m)$. The critical path delay of the *scalar* multiplication has been reduced by using a high-radix implementation based on a new proposed algorithm and an optimized processor array field ALU. The proposed architecture for $GF(2^{163})$ achieves a maximum of 326 in comparison to previous FPGA implementations, and a maximum speedup of 5.6 in comparison to previous ASIC implementations.

In Chapter 7, we summarize this dissertation, state our contributions, and suggest

directions for future research.

Chapter 2

VoIP Security: Protocols, Threats, and Security Requirements

This chapter reviews necessary background, including VoIP system and VoIP security. Our discussion on the VoIP system includes protocols, architectures, and equipment classifications. Our discussion on VoIP security is divided into two separate topics: VoIP threats and solutions.

This chapter is organized as follows: Section 2.1 discusses VoIP systems protocols, architectures, and equipment classifications. Section 2.2 reviews VoIP threats, risks and solution approaches. Section 2.3 summarizes the chapter.

2.1 The VoIP System

This section reviews the background of the Voice over IP in terms of its protocols, architectures, and equipment classifications. We focus our discussion on H.323 and SIP protocols and their architectures, since they are the most common VoIP protocols.

2.1.1 VoIP Protocols and Architecture

VoIP utilizes two important protocols, which have been derived from various standards and vendors, namely H.323 and Session Initiation Protocol (SIP). Moreover, there is a number of proprietary VoIP protocols such as MGCP, Megaco/H.248, Skinny Client Control Protocol (SCCP), Inter-Asterisk eXchange protocol (IAX) and Skype [21, 50]. In our research we focus on the well known two standard protocols, as the other protocols are derived from these two standards [1, 6].

H.323

H.323 is the International Telecommunication Union (ITU) standard originally proposed for multimedia (audio and video) conferencing over packetized networks. H.323 was proposed by the ITU Study Group 16. The original standard was realized in 1996 and further enhancements are being developed up until now. H.323 works as a wrapper for a set of ITU media control protocols (e.g. H.225, H.235, H.239, H.245, H.450, etc.). Each protocol has a specific role in the call-setup process. H.323 is currently implemented by various Internet real-time applications such as NetMeeting. It has the ability to handle both point-to-point communications and multi-point conferences. Figure 2.1 illustrates the architecture of H.323 networks. H.323 networks consist of four logical components: several endpoints (also called terminals, handsets or IP phones), gateways, gatekeepers and multipoint control units (MCUs) [1,6,51,52]. The description and the function of each element in H.323 network is as follows:

Endpoints: Endpoints are terminating devices built in either hardware or software, which participate in real-time, two-way communications with other H.323 endpoints, gateways, or multi-point control units (MCU). A terminal must support voice communication and may also support video or data or a combination of all three [1, 6]. See Section 2.1.2 for more details.

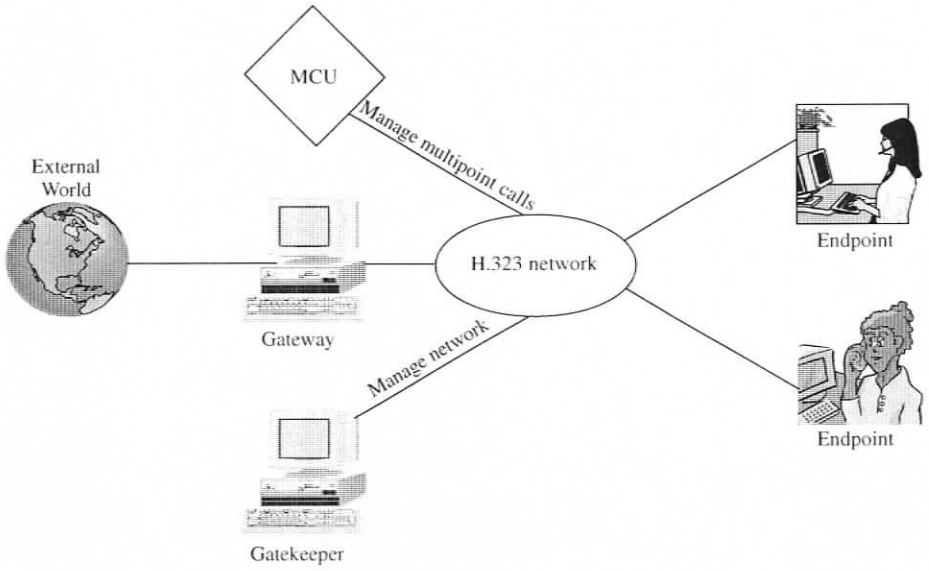


Figure 2.1: H.323 network architecture.

Gateways: Gateways are servers, connecting H.323 networks to the outside world, including SIP networks and traditional public switched telephone networks (PSTNs).

Gatekeepers: Gatekeepers perform two crucial call control functions. The first function is address translation services between the endpoints and gateways. The second function is bandwidth management. For instance, if the maximum number of simultaneous conferences has been reached, the gatekeeper can refuse to make any more connections [1, 6].

Multipoint Control Units (MCUs): MCU is an endpoint that provides the capability for three or more terminals and gateways to participate in a multi-point conference. It controls and mixes video, audio, and data from terminals to create a robust video conference. MCUs can also connect two

terminals in a point-to-point conference that can later develop into a multi-point conference [1, 6].

Session Initiation Protocol (SIP)

SIP is the Internet Engineering Task Force (IETF) proposed standard for initiating, modifying, and terminating an interactive multimedia (video, voice, instant messaging, etc.) session over packetized networks [53]. The first SIP proposed standard was defined in RFC2543 [54]. The protocol was further clarified in RFC3261 and RFC4321 [55, 56]. In November 2000, SIP was accepted as the 3rd Generation Partnership Project (3GPP) signaling protocol and permanent element of the IP Multimedia Subsystem (IMS) architecture [1]. It is one of the leading signalling protocols for VoIP, along with H.323. SIP works in conjunction with several other protocols and is only involved in the signaling portion of a communication session. SIP acts as a carrier for the Session Description Protocol (SDP), which describes the media content of the session, (e.g. which IP ports to use) the CODEC being used, etc. [1, 6]. In typical use, SIP "sessions" are simply packet streams of the Real-time Transport Protocol (RTP). RTP is the carrier for the actual voice or video content itself [53]. SIP was designed to be simpler than H.323 but it became as complex as H.323 as it evolved [6, 52, 55, 56].

Figure 2.2 illustrates the architecture of SIP networks. SIP network architecture differs from H.323 structure. A SIP network consists of endpoints (also called User Agents (UA)), a proxy (also called redirect server), a location server, and a registrar (also called User Agents Server (UAS) [1, 6, 55, 56]. The description and the function of each element in SIP network is as follows:

Endpoints: Endpoints are terminating devices built in either hardware or software, which participates in real-time, two-way communications with another SIP

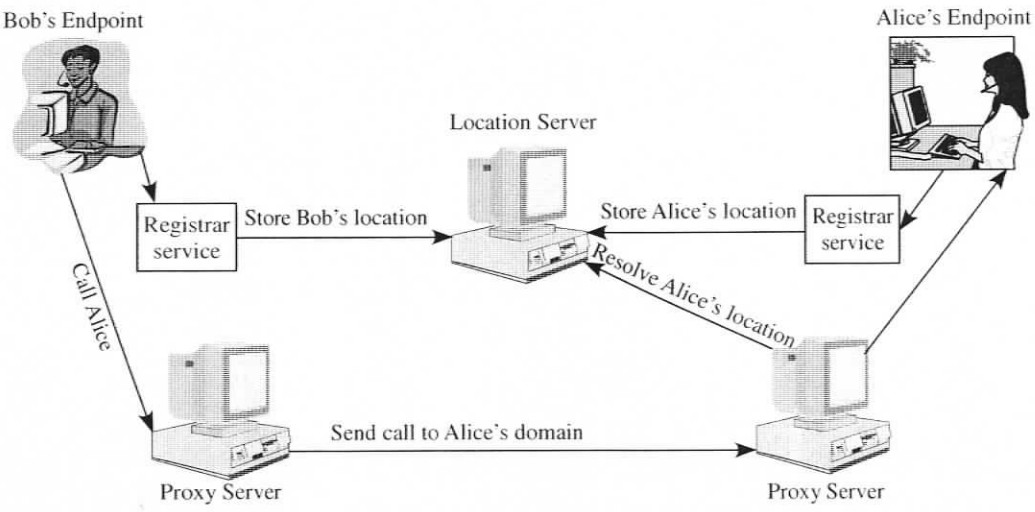


Figure 2.2: SIP network architecture.

endpoint, proxy server, location server, or User Agent Server (UAS). A terminal must support voice communication and may also supports video, or data or a combination of all three [1, 6, 55, 56]. See Section 2.1.2 for more details.

Proxy Servers: A server that helps route requests to the user's current location, authenticate and authorize users for services, implement provider call-routing policies, and provide features to users [1, 6, 55, 56].

Location Servers: A server that stores the location of every user [1, 6, 55, 56].

Registrar or User Agent Server (UAS): A server, which provides a registration function that allows users to upload their current locations for use by proxy servers [1, 6, 55, 56].

In the SIP model, a user does not belong to a specific host. Instead, each user initially reports his location to the registrar, which in turn stores the users location in the location server [55].

2.1.2 VoIP Endpoint Classifications

VoIP endpoints are the end-user devices that support two-way, real-time voice and video communications across packet switched networks. Although VoIP could be established by using a computer along with special software, the term "VoIP" is associated with an equipment, which lets the user dial numbers to communicate with another party on the other end who has a VoIP phone or traditional PSTN telephone [4]. Therefore, VoIP endpoints could be implemented by using two techniques: a computer along with special software (soft-phone) or dedicated hardware especially built for VoIP [6].

Hardware VoIP Phones

A VoIP phone is merely a telephone handset with a built-in CODEC and embedded H.323 or SIP protocol implemented, network interface card (NIC), and IP protocol stack. The VoIP phone is plugged directly into an Ethernet LAN just as a PC or another Ethernet station would be. Usually these products have extra features beyond being simple handset with dial pads. For example, VoIP phones may have small LCD screens and cameras that could be used for conference calls. Some of these VoIP phones are cordless to provide user convenience [6].

Another type of VoIP phones is the mobile (wireless) phones. These mobile VoIP phones utilize the wireless networking infrastructure (802.11). They may suffer additional security rescues, as the Wired Equivalent Privacy (WEP) security features of 802.11b may provide little or no protection [6].

Software VoIP Phones (Soft-Phones)

Software VoIP phones (also called PC phones or Soft-Phones) are merely a computers with special software to build a VoIP phone on a network. For example, NetMeeting is

a software that comes with Microsoft windows to support VoIP services. This option is very similar to the VoIP phone option with one major difference: the PC phone typically has one NIC that will be used for both VoIP packets and data packets. In the case of hardware IP phones, the NICs are dedicated to process VoIP packets only [6].

2.2 VoIP Threats, Risks, and Solutions

This section reviews the background of the VoIP security in terms of its threats, risks, and solutions. We focus on four security techniques, which are encryption/decryption, user authentication, data integrity, and key management. Moreover, we focus on two security protocols, which are IPSec and SRTP [13, 15]. These are reviewed in the following subsections.

2.2.1 VoIP Threats and Risks

VoIP components often use general-purpose operating systems, which tend to have more vulnerabilities than special-purpose operating systems. If not properly secured, each VoIP component (e.g. endpoints) is susceptible to a number of easily anticipated attacks. These attacks are classified into four categories as follows:

1. Attacks related to confidentiality and privacy, such as eavesdropping.
2. Attacks related to user authentication, such as spoofing or phishing (e.g. service theft, toll fraud, registration hijacking, etc.).
3. Attacks related to data integrity, such as call integrity compromise.
4. Attacks related to service availability, such as Denial of Service (DoS) and malicious code (e.g. worms, viruses, and SPAM over Internet Telephony

(SPIT)).

These attacks should be taken into consideration before the deployment of any VoIP system.

Eavesdropping or Voice Tapping

Users expect voice calls to be private. Although some VoIP calls are encrypted, the most are not because encryption degrades VoIP quality. As a result, VoIP packets could be easily captured and converted to audio files by using any common sniffing tools (such as, VoIPong, Voice Over Misconfigured Internet Telephones (VOMIT), etc.) [10–12]. Eavesdropping is one of the most dangerous threats in VoIP environments. Eavesdropping can create destruction to end users of VoIP systems. Eavesdroppers can steal very confidential personal information such as user names, passwords, phone numbers, billing information, credit card numbers, etc. Eavesdropping leads to more dangerous risks such as service theft, toll fraud, and registration hijacking [3, 5, 22, 57, 58].

On the other hand, encryption without strong authentication cannot guarantee privacy, because users cannot be sure an attacker is not accessing the media by performing a Man-In-The-Middle (MITM) attack.

VoIP Service Theft

VoIP service theft occurs when an unauthorized third party obtains a user name and password and uses it to place calls at the expense of the legitimate account holder. Service theft may also occur when a third party gains unauthorized physical access to a VoIP device. Typically, VoIP service theft is used for international calls where rates are higher [3, 57].

Toll Fraud

Toll fraud happens when a third party steals a username and password of a legitimate user and starts behaving as if he is the legitimate account holder. The risk of toll fraud is greatly increased in VoIP networks compared to PSTN systems due to the open nature of VoIP networks and vulnerability to toll fraud via spoofing (also known as phishing or scaming). Unauthorized access to VoIP networks allows attackers to spoof known source or destination addresses of VoIP terminals, creating both privacy and theft-of-service risks. In other words, attackers have the ability to change how his/her number appears on victims' caller ID; leading him to believe it is a bank, credit card company or other trusted authorities calling. Spoofing allows attackers to drain confidential information such as one's social insurance number and credit card information [22, 58].

Registration Hijacking

An attacker can register his/her phone and receive incoming calls intended for a legitimate phone. In registration hijacking, the attacker impersonates a valid User Agent (UA) to a registration server and then registers himself, causing all requests intended for the legitimate UA to be directed to the attacker instead. Registration hijacking can result in toll fraud or service disruption. Massive calls could be generated automatically, resulting in DoS attack against a gateway or Public Branch eXchange (PBX). Attackers can also eavesdrop or monitor calls by redirecting all incoming phone calls or outgoing gateway calls to their phones [21, 59].

Call Integrity Compromise

This attack happens when an attacker alters or modifies packet sequencing and/or corrupts packet contents. This degrades the voice quality of service (QoS) and causes

service disruption or DoS [22].

Proxy Impersonation

Proxy impersonation occurs when an attacker tricks an SIP User Agent Server (UAS) or proxy into communicating with a rough server. If an attacker succeeds in impersonating a UAS, he will have access to all SIP messages and thus complete control of the call. Therefore, all outbound calls to that UAS can be intercepted, manipulated, or recorded [21, 59].

Message Tampering

Message tampering occurs when an attacker intercepts and modifies packets exchanged between SIP components. Message tampering can occur through registration hijacking or proxy impersonation. SIP messages have no built-in means of insuring data integrity. Therefore, manipulating VoIP packets may cause registration hijacking or proxy impersonation [21, 59].

Session Tear Down

Session tear down occurs when an attacker observes the signaling for a call and then sends spoofed SIP "BYE" message to participating User Agent Servers (UAS). Most SIP UASs do not require strong authentication, which allows an attacker to tear down the call by sending "BYE" messages to UASs [21, 59]

Service Disruption or Denial of Service Attacks (DoS)

Service Disruption or Denial of Service DoS attacks are generally the most simple and thus most common attacks faced by IP data networks. VoIP DoS attacks are similar to DoS attacks conducted in IP data networks, in which attackers simply bombard

the call processing/managing server with an excessive amount of simultaneous call requests and registrations. This attack overwhelms servers with requests, causing them to shutdown and make the services they provide unavailable to legitimate users. Calls in process would be suddenly terminated and any attempt to place other calls will not succeed. A successful DoS attack causes risks such as: loss of revenue due to lost sales call volume, negative customer experience resulting from lost support calls, and productivity lost as communications with remote offices drop off [3,21,22,57–59].

SPAM over Internet Telephony (SPIT)

SPIT is the VoIP equivalent of unwanted email messages clogging up a voice mailbox. VoIP SPIT directly hits gateways and degrades the voice quality because of the degradation in bandwidth. Industry observers have cautioned that the open nature of a VoIP phone call makes it easy for spammers to send numerous voice mail messages, therefore causing a reduction in bandwidth and breaks in service much in the same way that e-mail in boxes are spammed. The strain on network resources when millions of 100 KB voice mail messages are transmitted, compared with 5 or 10 KB e-mails, will be considerable. Security specialists, and VoIP vendors for that matter, are treating the prospect of unwanted voice messages as a security threat, much as they have done with e-mail SPAM [58].

2.2.2 VoIP Security Requirements and Solutions

As we learned from Section 2.2.1, VoIP is vulnerable to the traditional IP network attacks as well as specified attacks for voice services. A successful VoIP network requires strong security because non-secure VoIP networks do not maintain the expected levels of quality and reliability [6].

The main goal of securing VoIP systems is to keep communication private and

protect sensitive data from being accessed by unauthorized users [60, 61]. These requirements include:

Protection of privacy of call conversation: It could be provided by encrypting all connections between network elements.

Protection against unauthorized access: It is provided by user authentication, which verifies that a user or client is legitimate.

Protection of data integrity: It is provided by message authentication, which verifies message integrity by using techniques such as AES-CBCMAC.

Protection of encryption key: It is provided by using a key management technique such as RSA or, Elliptic Curve (ECC).

Protection of service disruption: It is provided by using a firewall, which prevents service disruption such as DoS.

2.2.3 VoIP Security Protocols

There are several characteristics and requirements unique to VoIP which make providing security to VoIP much more complex and more difficult than securing a traditional circuit-switched voice network [6].

There are many security protocols used to secure data communication. Examples include SSL, SSH, S/MIME, S/WAN, IPSec, and Kereberos. The most popular and widely-used among these protocols are IPSec and SSL [62, 63]. Out of these security protocols intended for data, only IPSec is suitable for VoIP [6]. Moreover, Secure Real-Time Transport Protocol (SRTP), which is the secure version of Real-Time Transport Protocol (RTP), is intended for securing the transmission of media over IP [62]. Therefore, we limit our discussion to these two protocols, IPSec and SRTP.

The IP Security Protocol (IPSec)

IPSec stands for "IP Security", which is a protocol used to provide security services to the IP layer. These security services include authentication, encryption/decryption, data integrity, and dynamic rekeying, which helps avoid attacks [61]. IPSec requires that all participating network equipment be consistent. This is achieved using a Security Association (SA), which is a state stored at each endpoint of a secure connection. It indicates how to protect traffic, what traffic is to be protected, and with whom protection is performed [13, 14]. Three protocols are used in IPSec implementation:

Authentication Header (AH) This protocol provides packet authentication by attaching strong crypto checksum to the packet. Packet authentication ensures proof-of-data origin, data integrity, and anti-replay protection, but does not provide data confidentiality [13, 14, 61].

Encapsulated Security Payload (ESP) This protocol provides a combined authentication/encryption facility that adds data confidentiality to the functions of AH [13, 14, 61].

Internet Key Exchange (IKE) This protocol organizes the process of exchanging security keys required for encryption and authentication and ensures their privacy [13, 14, 61]. The Diffie-Hellman public key exchange mechanism is used for the purpose of negotiating the shared secret key.

Both AH and ESP support two modes of use: *Transport* mode, which protects only layers above the IP layer and *Tunnel* mode, which protects the IP as well as the above layers [64]. IPSec works on entire packets. The main functions of IPSec include checking the validity of the packet to avoid replay attacks, authenticating the

packet using authentication algorithms, and encrypting and decrypting the payload to ensure data confidentiality [13, 14].

The IPSec protocol utilizes either hashing algorithms (such as HMAC-SHA1, HMAC-MD5, etc.) or block ciphers (AES, TDES, RC5, etc.) in CBC-MAC mode for authentication. On the other hand, the IPSec utilizes a block cipher (AES, TDES, RC5, etc.) in the CBC mode for encryption/decryption [61].

Secure Real-time Transport Protocol (SRTP)

Real-time Transport Protocol (RTP) is commonly used to transmit real-time audio and video on a packet-switched network. RTP does not provide security to voice conversation, which in turn allows eavesdropping and other attacks. The Secure Real-time Transport Protocol (SRTP) defines a profile of RTP to provide confidentiality, message authentication, and replay protection to the data in both unicast and multicast sessions. SRTP was developed by a small team of IP protocol and cryptographic experts from Cisco and Ericsson. It was standardized by IETF and published in March 2004 as RFC 3711 [15, 65].

Since RTP is closely related to RTCP (Real-Time Control Protocol) which can be used to control an RTP session, SRTP also has a sister protocol, called Secure RTCP (SRTCP). SRTCP provides the same security-related features to RTCP, as the ones provided by SRTP to RTP [66, 67].

SRTP provides the following security-related features to RTP [66, 67]:

Encryption/Decryption For the confidentiality of data flow (i.e., encryption and decryption), SRTP (together with SRTCP) standardizes utilization of only a single cipher, AES, which can be used in two cipher operational modes (*counter* mode (CTR) and *F8* mode), allowing the AES block cipher to work as a stream cipher [15, 65]:

1. **Counter Mode (CTR)**: A typical *Counter* mode, allows random access to any blocks, which is essential for RTP traffic running over unreliable network with possible loss of packets. In general, any function can be used in the role of "counter", assuming that this function does not repeat for a long number of iterations. But the standard counter mode utilizes a usual integer incremental counter for the encryption of RTP data. AES running in this mode is used with a default encryption key length of 128 bits (i.e., AES-128) [15, 65].
2. **F8 Mode**: A variation of the Output Feedback mode (OFB), it is enhanced to be seekable and with an altered initialization function. The default encryption key length is the same as for AES in the *Counter* mode [15, 65].

Integrity and replay protection The above-listed encryption algorithms do not secure message integrity themselves, allowing the attacker to either forge the data or at least to replay previously transmitted data. Hence the SRTP standard also provides the means to secure the integrity of data and safety from replay. To authenticate the message and protect its integrity, the HMAC-SHA1 algorithm (defined in RFC 2104 [68]) is used, with a default length of 160 bits for the HMAC-SHA1 authentication. But it does not protect against replay attacks: in replay protection, the receiver maintains the indices of previously received messages and compares them with the index of each new received message and admits the new message only if it was not played before. Such an approach heavily relies on the integrity protection being enabled (to make it impossible to spoof message indices) [15, 65].

MIKEY

MIKEY describes a key management scheme that addresses real-time multimedia scenarios (e.g. unicast, multicast, etc.). SRTP uses a set of negotiated parameters from, which session keys for encryption, authentication, and integrity protection are derived. MIKEY is currently being standardized by IETF. MIKEY is designed for peer-to-peer, simple one-to-many, and small-size interactive group scenarios. MIKEY supports the negotiation of cryptographic keys and security parameters (SP) for one or more security protocols. MIKEY offers independency of a specific communication protocol (SIP, H.323, etc.). Moreover, key material is established within a 2-way handshake. Therefore, MIKEY is best suited for real-time multimedia scenarios [15, 65]. MIKEY has four options for key distribution:

1. Preshared-key
2. Public-key encryption (e.g. RSA, Elliptic Curve (ECC))
3. Diffie-Hellman key exchange protected by public-key encryption
4. Diffie-Hellman key exchange protected with preshared-key and keyed hash functions

2.2.4 Effect of Implementing VoIP Security on the QoS

VoIP QoS ensures voice quality when it is transmitted over an IP network. VoIP transmission must meet the QoS requirements by using different features such as packet classification, queueing mechanisms, traffic shaping, header compression, and congestion avoidance strategies. QoS is a basic requirement for VoIP networks to ensure voice quality [2, 69].

Securing VoIP may cause complications that degrade the QoS. These complications include call setups delaying or blocking by firewalls, encryption-produced delay

and delay variation (jitter). Because VoIP is delay-critical and has a low tolerance for disruption and packet loss, many security techniques implemented in traditional data networks are not applicable to VoIP in their current form [6].

VoIP delay is the time required for a voice transmission to go from its source to its destination. Ideally, delay should be kept as low as possible but there are acceptable lower bounds on VoIP delay. The ITU standard G.114 recommended that the upper bound be 150 ms for one-way traffic. This corresponds to the current delay bound experienced in domestic calls across PSTN lines [18, 19]. For international calls, the ITU considered a delay of up to 400 ms to be tolerable, but since most of the extra time is spent routing and moving the data over long distances, we only consider the domestic case. As a result, VoIP calls must achieve the 150 ms bound to successfully provide the QoS that PSTN phones provide. This time constraint leaves a very tight margin for the amount of security that can be added to VoIP transmission. QoS analysis reduces this bound to 80 ms only to achieve the PSTN voice quality, which in turn reduces the time margin left for security [70]. Figure 2.3 illustrates delay sources for the VoIP packet. These delay sources are: voice encoding, packetization, queuing/buffering, serialization, network delay and jitter buffer. The estimate of these delays tells us the amount of time left for security. The total of these delays ranges from 121 ms + Network transmission delay [20] to 146 ms [19]. These delay amounts leave a very tight margin for security implementations.

Delay is not only introduced in endpoints of the system, but each hop along the network adds queuing delay and processing delay if it is a security checkpoint (i.e., firewall or encryption/decryption point). Also, larger packets tend to cause bandwidth congestion and increased delay. Moreover, adding encryption and authentication to VoIP packets increases the packet length by a range of 11-70%, which in turn increases the amount of processing time required for VoIP packets [16].

In light of these issues, VoIP tends to work best with small packets on a logically

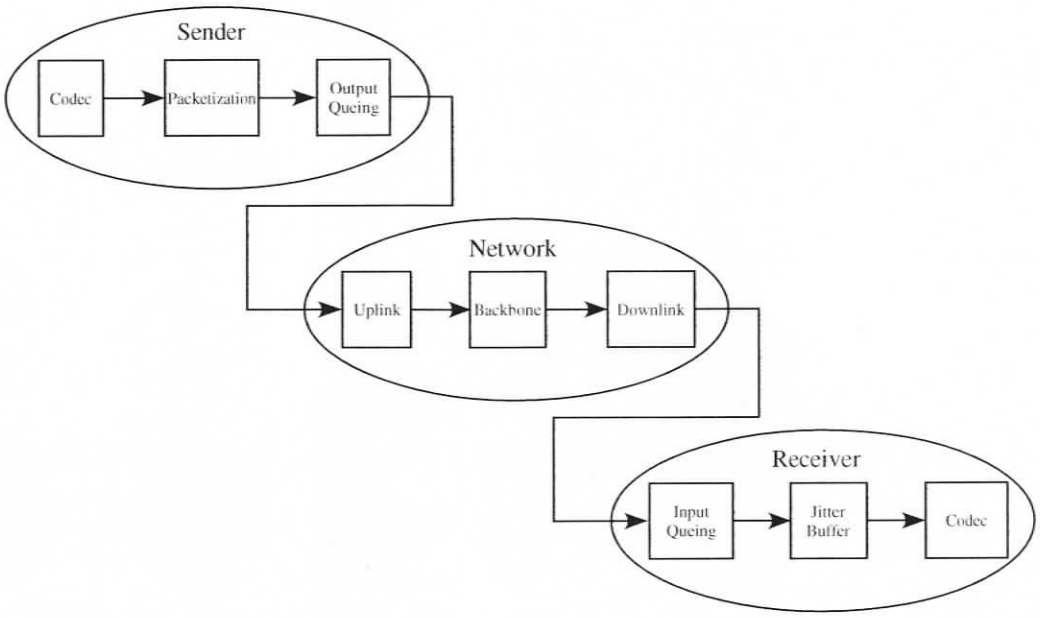


Figure 2.3: Delay budget in VoIP network.

abstracted network to keep delay at a minimum [6].

The Need for Speed

The most two effective ways to secure networks are firewalls and traffic encryption/decryption. However, they are also two of the greatest contributors to network congestion and throughput delay. To reduce the extra delay resulting from firewall and traffic encryption/decryption, we need to speed up these processes. Because VoIP is a real-time application and time sensitive, the deployment of traditional firewalls and encryption products into a VoIP network is not feasible, particularly when VoIP is integrated into existing data networks [6]. Instead, these data-network solutions must be adapted to support security in the new fast paced world of VoIP.

2.2.5 VoIP Products

There are few VoIP processors proposed in the literature by vendors. Most of these processors do not fulfill VoIP security with QoS requirements. For example, VoIP processors in [71–75] do not provide any security for VoIP at all. We can not find any hardware efforts in the literature that provide VoIP security. This is because VoIP evolution follows the phases as Internet evolution, i.e., security comes last. Manufactures do not offer VoIP security until they initially improve service to attract more customers. In other words, manufactures do not offer VoIP security because there is no customer base. Once number of customers is increased, the number of hackers will increase, which in turn justifies securing VoIP.

On the other hand, Cisco is one of the biggest IP phones vendors in the market. The IP phones series 7900 are an example of their products. Most of these series do not provide any VoIP security except for IP phone 7911G. Although it is mentioned in 7911G's data sheet that this IP phone provides security it has a fixed key and one needs to contact Cisco to replace it [76]. Also, this phone does not provide data integrity and key management techniques. Moreover, the user authentication certificate is also fixed [76].

Broadcom has a single-chip VoIP solution such as the BCM1100 series. Although the BCM1113R and the BCM1112 do not also provide security for VoIP, BCM1104 provides some of the required security (i.e. encryption and user authentication), but they lack data integrity and key management [77]. On the other hand, Broadcom produces stand alone security processors (e.g. 5800 series). These processors are designed for data network without taking into consideration the QoS of real-time applications. Moreover, these processors do not provide a complete solution on one chip, (i.e., none of these processors have encryption, message authentication, user authentication, and key management on the same chip) [77].

Intel products such as IXP421 and IXP423 do not provide security for VoIP traffic [78–80]. On the other hand, the IXP425 processor designed for data network security does not provide the speed required for real-time media over IP applications [81].

Although we are not interested in software products, the well-known application Skype provides security on the expense of voice quality and delay. Gao et al. [82] proved by performance analysis that Skype voice quality is worse than MSN Messenger and suffers a longer delay. Moreover, Skype is vulnerable to viruses, malicious code, and DoS attacks [83].

2.3 Chapter Summary

This chapter explored the fundamentals of the VoIP system and security issues related to it. Securing VoIP requires different approaches since it suffers many kinds of threats. These approaches include: encryption/decryption, user authentication data integrity, and key management.

VoIP encryption/decryption as well as message authentication (i.e., data integrity) requires a block cipher working in different modes. In the next chapter, we propose a reconfigurable, high throughput hardware implementation for the different block cipher operational modes. The proposed architecture has the ability to encrypt/decrypt a plaintext/ciphertext efficiently using ECB, CBC, CFB, OFB, CTR operational modes. Moreover, it has the ability to prove data integrity using CBC-MAC, CCM, or RMAC operational modes.

Chapter 3

A Unified, Reconfigurable Architecture Implementing Block Cipher Operational Modes

This chapter presents the first contribution of this dissertation. We propose a reconfigurable, high throughput hardware implementation for the different block cipher operational modes. The proposed architecture is unified; it combines multiple related functions on the same architecture. The proposed architecture has the ability to encrypt/decrypt a plaintext/ciphertext efficiently using ECB, CBC, CFB, OFB, and CTR operational modes. Moreover, it has the ability to prove data integrity using CBC-MAC, CCM, or RMAC operational modes. The proposed architecture is tested using the most widely used block ciphers: DES [31], TDES [31], AES-128 [25], AES-192 [25], AES-256 [25], and IDEA [32]. We synthesized these block ciphers using a Xilinx XC2V4000-6-bf957 FPGA [33]. For the sake of evaluation, we synthesized an architecture that implements only the ECB. In comparison with the architecture that implements only the ECB mode, our proposed architecture requires extra area

ranging from 1.6% for AES-256 to 18.4% for DES and extra delay ranging from 0% for DES to 7% for IDEA. Moreover, the proposed architecture implementation is analyzed and evaluated by comparing it against other implementations.

This chapter is organized as follows: Section 3.1 introduces this chapter. In Section 3.2, the block cipher modes of operation are briefly described. In Section 3.3, the proposed architecture of the reconfigurable, unified architecture is discussed in detail. In Section 3.4, the proposed architecture implementation is analyzed and evaluated by comparing it against other implementations. Section 3.5 summarizes the chapter.

3.1 Introduction

Securing VoIP traffic has become one of the most important issues due to the wide use of the Internet. Persistent network threats and eavesdropping increase the demand for securing VoIP traffic. Therefore, there is an increasing demand for encryption and authentication algorithms to secure VoIP traffic. Encryption techniques are used to secure the transmitted VoIP, whereas authentication techniques are used to provide assurance of VoIP packets integrity along with VoIP packets origin authentication.

A commonly used technique for encryption and authentication is block cipher, which is also known as symmetric-key, secret-key, or private-key cipher [60]. Block cipher could be used in one of eight standard modes, summarized in Table 3.1. These different modes allow matching different application requirements [84–86]. The straightforward mode, also called Electronic Code Book (ECB), is not considered secure when compared with other modes; it generates the same ciphertext for a given plaintext under a given key [60, 84]. As such, the National Institute of Standard and Technology (NIST) has recommended some other block cipher modes for encryption and authentication to overcome security risks in data transmission [87–89].

Combining these modes with state of the art security protocols (e.g. IPSec and IEEE 802.11) helps achieve a high level of security required for crucial applications such as online banking, Voice over IP (VoIP), and electronic commerce. A recent RFC, together with other researchers, have recommended combining Advanced Encryption Standard (AES) with block cipher modes of operation, (e.g. AES in CCM mode) with IPSec [84–86]. The CCM mode, in turn, involves CTR and CBC modes [90].

The expanded high data rate use of block cipher protocols mandates looking for efficient implementations for them at the hardware level. Such architectures should be dynamic and consume less power to meet contemporary specifications [91–93].

Careful inspection of the operation of the applications in Table 3.1 shows that block ciphers might use multiple modes of operation in the same application. Hence, designing a hardware core that efficiently implements all these modes in a reconfigurable way provides a dynamic, small area, low power consumption architecture.

3.2 Block Cipher Modes of Operations

A block cipher breaks the plaintext into n-bit N blocks. The i^{th} plaintext block, P_i, is transformed to the i^{th} ciphertext block, C_i, using one of the modes explained below [87–89]. A block cipher mode combines a block cipher algorithm, some kind of feedback, and some glue logic [32, 60].

3.2.1 Electronic Codebook Mode (ECB)

Figure 3.1 illustrates the ECB mode. ECB is the simplest block cipher mode, in which the block cipher's encryption function (E_k) is applied separately for each plaintext block (P_i) to generate the cipher text (C_i). The plaintext (P_i) is recovered by applying

Table 3.1: Block cipher modes of operation description, usage, characteristics, and some applications.

Mode	Description	Usage*	Characteristics	Some Application(s)
Electronic Codebook (ECB)	Each plaintext block is encrypted independently.	E/D	Each possible plaintext block has a defined corresponding ciphertext block.	Secure transmission of single value (e.g. an encryption key)
Cipher Block Chaining (CBC)	Plaintext is XORed with the previous ciphertext, then encrypted.	E/D	Chaining mechanism: Ciphertext depends on all the preceding ciphertext blocks. XORing hides plaintext patterns.	General purpose secure block oriented transmission (e.g. IPSec, IEEE 802.11)
Cipher Feedback (CFB)	Input is processed s bits at a time. The previous ciphertext block is encrypted and XORed with the current plaintext.	E/D	Different block sizes encryption. Chaining mechanism: Ciphertext depends on all the preceding ciphertext blocks., XORing hides plaintext patterns.	General purpose secure stream oriented transmission
Output Feedback (OFB)	Similar to CFB except that the output of the previous encryption block function is encrypted instead of the previous cipher block.	E/D	Tolerant for error propagation.	Secure stream oriented transmission over noisy channel (e.g. satellite communication)
Counter (CTR)	Each plaintext block is XORed with encrypted counter. The counter is incremented for each subsequent block.	E/D	Ciphertext blocks do not depend on each other. Parallel encryption ability.	High speed secure block oriented transmission (e.g. IEEE 802.11)
Cipher Block Chaining Message Authentication Code (CBC-MAC)	CBC mode is used to encrypt the plaintext blocks. The last ciphertext block is called Message Authentication Code (MAC).	A	Data integrity. Data origin authentication.	General purpose secure block oriented transmission
CTR With CBC-MAC (CCM)	Encrypt using CTR. Authenticate using CBC-MAC.	E/D/A	Uses two other modes: CBC and CTR.	Secure packet transmission (e.g. IEEE 802.11)
Randomized MAC (CCM)	MAC is obtained by using CBC-MAC then encrypting the result using different keys.	A	Uses two secret keys. Hard to break because it uses two secret keys.	General purpose secure block oriented transmission

* E: Encryption , D: Decryption , A: Authentication

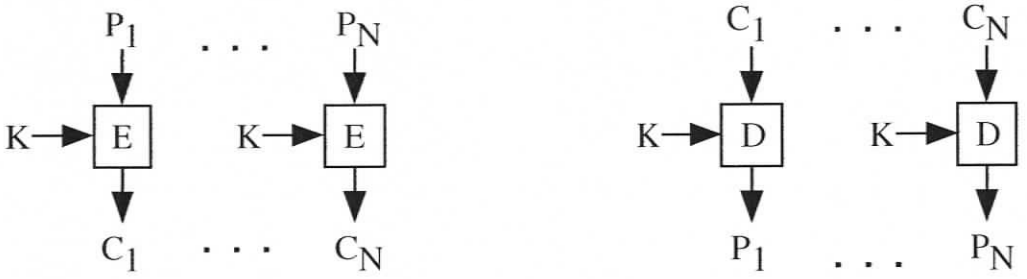


Figure 3.1: ECB mode: (a) encryption and (b) decryption, E: encryption core, D: decryption core, K: key.

the block cipher's inverse encryption function (D_k) to each cipher text block (C_i):

$$C_i = E_k(P_i), \quad i = 1, ..., N \tag{3.1}$$

$$P_i = D_k(C_i), \quad i = 1, ..., N \tag{3.2}$$

3.2.2 Cipher Block Chaining Mode (CBC)

Figure 3.2 illustrates CBC mode. The CBC is the most popular block cipher mode. CBC is a feedback mechanism, where the cipher of block i depends on the cipher of the previous block $i-1$. In CBC encryption, the plaintext P_i is XORed with the previous ciphertext block C_{i-1} then encrypted. To encrypt the first plaintext block P_1, one randomly chooses C_0 as the initialization vector (IV). The plaintext is obtained by decrypting the block resulting from XORing the ciphertext C_i with the previous ciphertext C_{i-1}.

$$C_i = E_k(P_i \oplus C_{i-1}), \quad i = 1, ..., N \tag{3.3}$$

$$P_i = D_k(C_i) \oplus C_{i-1}, \quad i = 1, ..., N \tag{3.4}$$

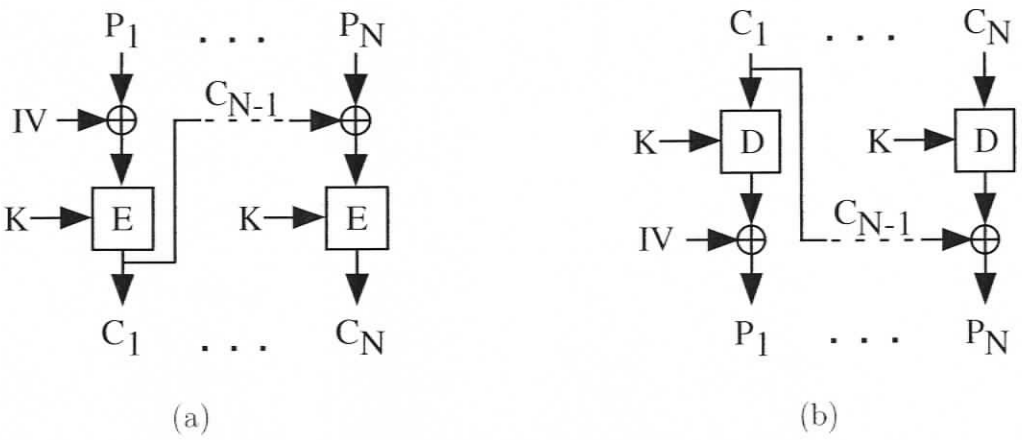


Figure 3.2: CBC mode: (a) encryption and (b) decryption.

3.2.3 Cipher Feedback Mode (CFB)

Figure 3.3 illustrates CFB mode. The CFB is a block cipher mode, in which the block cipher utilizes the forward cipher function for both encryption and decryption. In this mode, the block cipher could be used as a stream cipher [60].

In CFB encryption, the plaintext is divided into s-bits segments ($1 \leq s \leq n$) and the input to the block cipher is the output of an n-bit shift register. The input to the first block cipher is initially set to some initialization vector (IV). The most significant s bits of the encryption function output is XORed with the plaintext segment (P_1) to produce the ciphertext (S_1), where P_i's and C_i's are s-bit segment. Moreover, the shift register contents are shifted left by s bits and S_1 is placed in the least significant s bits of the shift register. The same process continues until the plaintext is all encrypted. The encryption is done according to:

$$I_i = ShL(I_{i-1}, s)|S_{i-1} \tag{3.5}$$

$$S_i = P_i \oplus MS(E_k(I_i), s) \tag{3.6}$$

where I_i is the i^{th} input vector to the encryption core, $ShL(I_{i-1}, s)$ shifts I_{i-1} to the

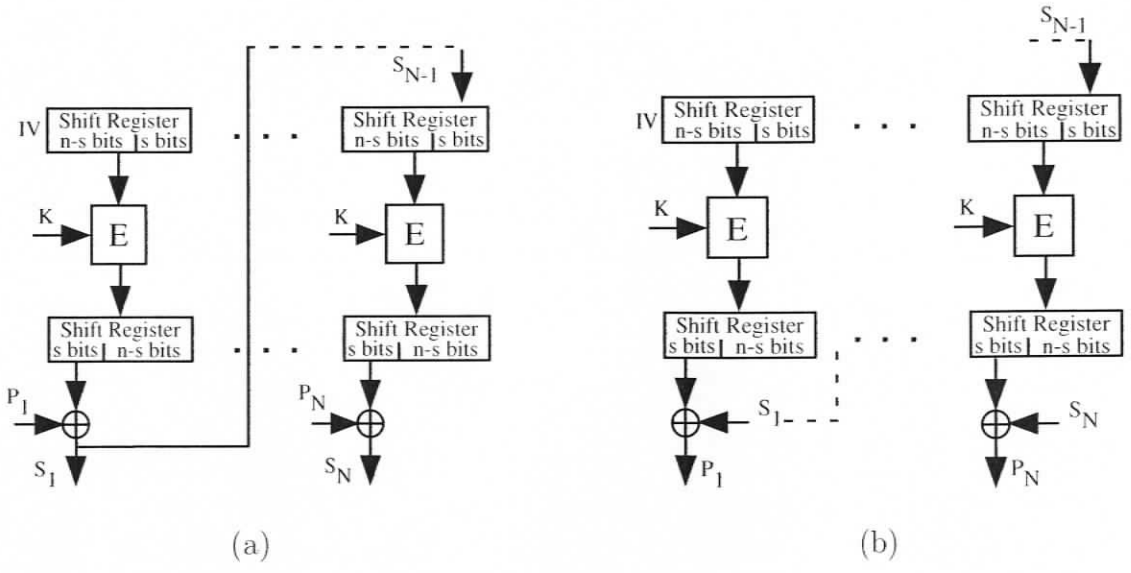


Figure 3.3: CFB mode: (a) encryption and (b) decryption.

left by s bits, $MS(E_k(I_i), s)$ returns the most significant s bits of $E_k(I_i)$, and | is the concatenation operator.

For decryption, the same process is used, except that the plaintext is the result of XORing the cipher text with the output of the encryption function according to:

$$P_i = S_i \oplus MS(E_k(I_i), s) \tag{3.7}$$

3.2.4 Output Feedback Mode (OFB)

Figure 3.4 illustrates the OFB mode. The OFB mode almost has the same structure as the CFB mode, except that the output of the encryption function is fed back to the shift register instead of the cipher as in CFB. The encryption is done according to:

$$I_i = ShL(I_{i-1}, s) | MS(E_k(I_{i-1}, s)) \tag{3.8}$$

$$S_i = P_i \oplus MS(E_k(I_i), s) \tag{3.9}$$

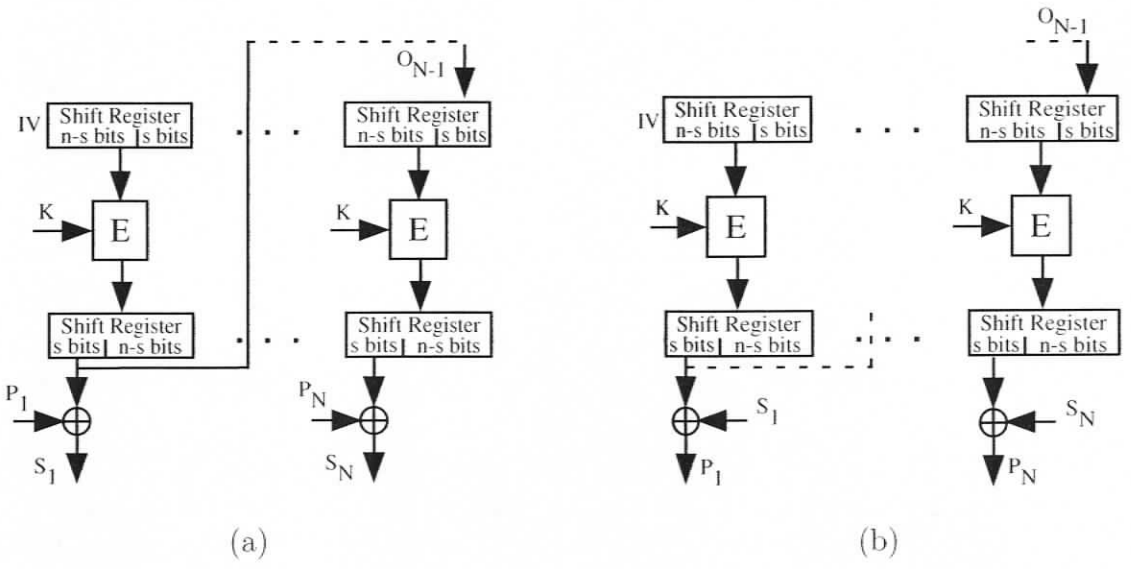


Figure 3.4: OFB mode: (a) encryption and (b) decryption.

The decryption is done according to:

$$P_i = S_i \oplus MS(E_k(I_i), s) \tag{3.10}$$

3.2.5 Counter Mode (CTR)

Figure 3.5 illustrates the CTR mode. The CTR mode is a block cipher mode that uses one secret key along with a block cipher in its forward function to do both encryption and decryption. The CTR mode utilizes a counter ($[CTR]_i$) with the same size as the plaintext block. The counter is a sequence of blocks initialized to some value then incremented by 1 for each subsequent block ($[CTR]_i = [CTR]_{i-1} + 1$). The initial counter block, $[CTR]_1$, must be chosen to ensure that counters are unique across all messages that are encrypted under a given key [60, 87]. For encryption, the counter is encrypted and then XORed with the plaintext to produce the ciphertext.

$$C_i = E_k([CTR]_i) \oplus P_i \tag{3.11}$$

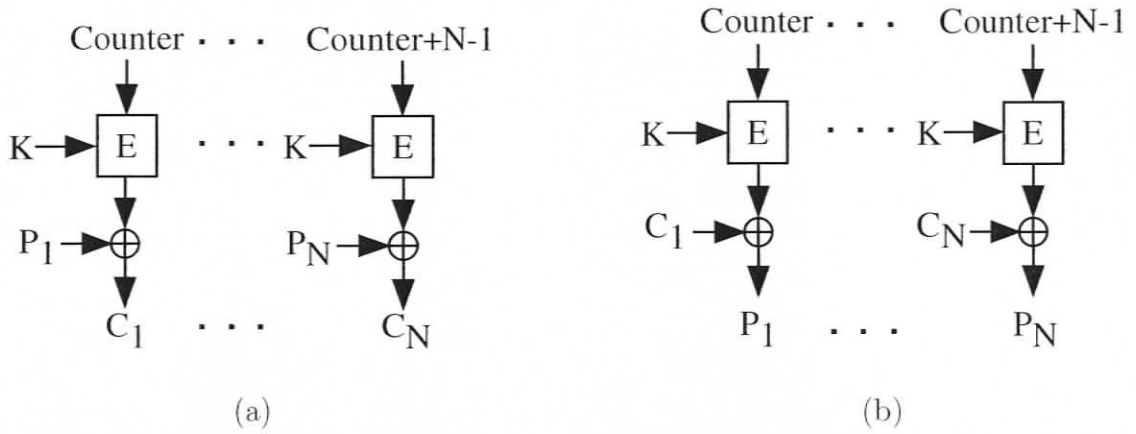


Figure 3.5: CTR mode: (a) encryption and (b) decryption.

The plaintext is obtained by encrypting the counter and then XORing the result with the ciphertext.

$$P_i = E_k([CTR]_i) \oplus C_i \tag{3.12}$$

3.2.6 Cipher Block Chaining Message Authentication Code Mode (CBC-MAC)

Figure 3.6 illustrates the CBC-MAC mode. The CBC-MAC is an authentication mode that uses cipher block chaining mode (CBC) in its forward direction to obtain the message authentication code (MAC). The CBC mode is used to encrypt the plaintext blocks. The last ciphertext block is the Message Authentication Code (MAC), whereas the other blocks are neglected.

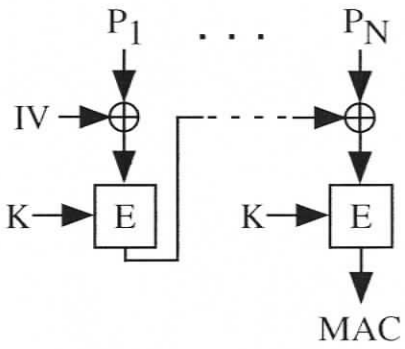


Figure 3.6: CBC-MAC mode.

3.2.7 Counter With CBC-MAC Mode (CCM)

CCM is an encryption and authentication mode that is designed for the packet environment. CCM utilizes the CTR mode for encryption and the CBC mode for message authentication. CCM uses the same secret key for both encryption and authentication. The input to CCM includes:

- Data to be encrypted and authenticated (payload),
- The header that will be authenticated but not encrypted, and
- A unique value called nonce that is assigned to the payload and the header.

CCM works in two directions:

- Generate-encrypt: which generates MAC for the whole packet and encrypts the payload concatenated with the MAC,
- Decrypt-verify: which decrypts the received encrypted payload concatenated with the MAC and verifies the received MAC by calculating the MAC from the decrypted payload and comparing it with the received MAC.

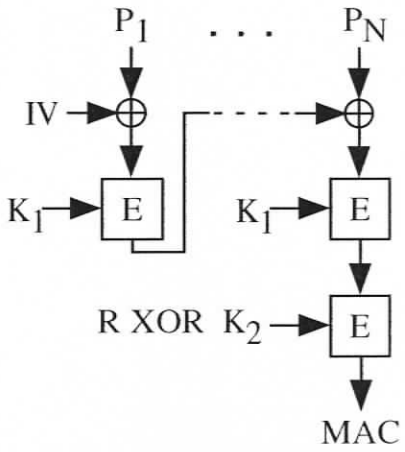


Figure 3.7: RMAC mode.

3.2.8 Randomized MAC Mode (RMAC)

Figure 3.7 illustrates the RMAC mode. The RMAC is an authentication mode that uses a block cipher and two secret keys for message authentication. The RMAC mode works as the CBC-MAC at the first stage using the first secret key. The result from the first stage is encrypted again using a key resulting from XORing the second key with a parameter associated with the message called salt, which is denoted as R.

3.3 Proposed Architecture

We propose a unified architecture for 8 block cipher operational modes. Our proposed architecture is illustrated in Figure 3.8. The inputs to the proposed architecture are the plaintext block P_i, an initialization vector IV, the ciphertext block from the previous stage C_{i-1}/S_{i-1}, the received MAC, the first key K_1, the second key K_2 and the RMAC salt R. The outputs from the proposed architecture are the ciphertext block C_i/S_i, the calculated MAC and the result of the MAC verification unit.

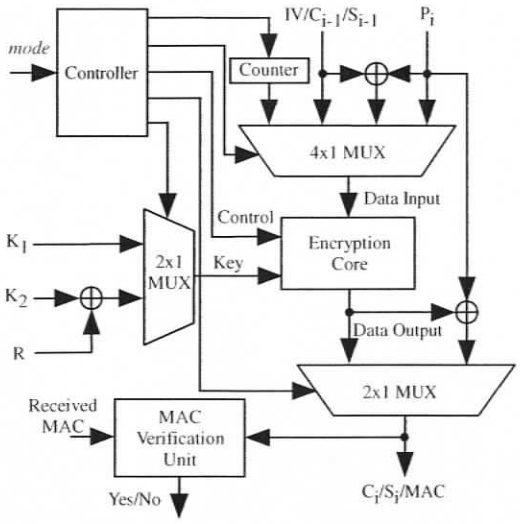


Figure 3.8: The reconfigurable unified architecture for the block cipher operational modes.

The architecture is composed of the encryption core, a counter unit, a MAC verification unit, a controller, multiplexors, and XOR gates. The description of each components is as follows:

- The encryption core represents the implemented block cipher algorithm.
- The counter unit generates and holds an n-bit integer needed for the counter mode operation.
- The MAC verification unit is used to verify data integrity by comparing the received MAC with the produced MAC. The received data is accepted or rejected according to the output of the verification unit. This unit is crucial for the CBC-MAC, CCM, and RMAC authentication mode.
- The controller generates control signals required to control the data bus components. It receives a signal called *mode* as an input and generates the

appropriate control signals for the encryption core, the counter, the MAC verification unit, and the multiplexors according to the selected mode.

- Multiplexors and XOR gates ensure that the appropriate data transformation is done according to the current working mode.

The proposed architecture (Figure 3.8) has several advantages. First, it is reconfigurable; it has the ability to switch between modes of operation according to the application's demand. Second, it is unified; it combines multiple related functions on the same architecture, which in turn reduces the area and power compared to separate implementations of each mode. Third, it is flexible; it has the ability to encrypt/decrypt a plaintext/ciphertext efficiently using the ECB, CBC, CFB, OFB, CTR, CBC-MAC, CCM, or RMAC mode.

3.4 Experimental Results and Comparison

3.4.1 Experimental Setup

To evaluate our proposed architecture, we modeled the most widely used block cipher (DES, TDES, AES-128, AES-192, AES-256, and IDEA) along with our multi-mode architecture using VHDL and then implemented them on an FPGA kit. To conduct a fair evaluation of the examined block cipher algorithms, a common architecture in implementing each of these block ciphers is used. Figure 3.9 illustrates this common architecture, which represents the encryption core found in Figure 3.8. The common architecture for the block cipher is composed of a key expansion unit, key memory, round logic, I/O interface, and a multiplexor. The key expansion unit receives the key and generates the required keys for each round. The memory key stores the generated round keys. The round logic receives the encrypted data from the previous round and

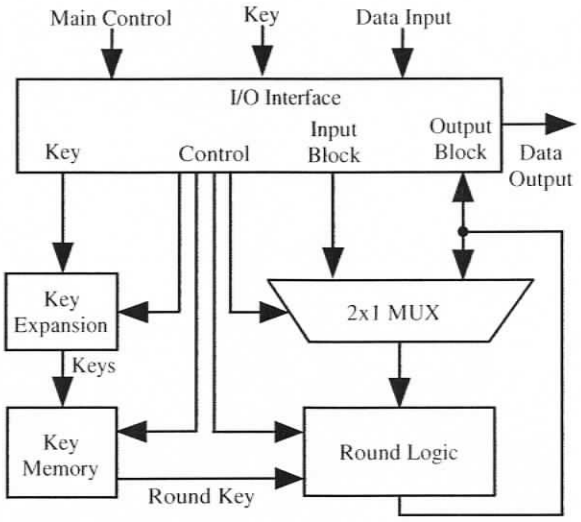


Figure 3.9: A common architecture of the encryption core used in Figure 3.8.

encrypts it using the current round key. The I/O interface receives the plaintext block (*Data Input*), the key (*Key*), and the Control signal (*Main Control*) as an input and passes them to the other unit for processing. The I/O interface generates ciphertext (*Output Data*) as an output. Moreover, the I/O interface generates the required control signals required for data transfer between registers. The multiplexor transfers the data either from the I/O interface or from the previous round.

The Xilinx ISE 6.2 is used to synthesize the design and provide post-placement area and timing results. The architecture is implemented on Xilinx XC2V4000-6 FPGA.

3.4.2 Results

The FPGA synthesis results for the examined block ciphers (DES, TDES, AES-128, AES-192, AES-256, and IDEA) are listed in Table 3.2.

Table 3.2 shows the percentage of the extra resources required for the unified

Table 3.2: Cost paid for proposed unified architecture.

Block Cipher	ECB mode				Unified architecture				Cost (Percentage)			
	Area (Slice)	t^*_{pd} ns	f^*_{Max} MHz	Th^*_{Max} Mpbs	Area (Slice)	t_{pd} ns	f_{Max} MHz	Th_{Max} Mpbs	Area	t_{pd}	f_{Max}	Th_{Max}
DES-64	364	6.9	145.2	580	431	6.9	145.2	580	18.4 %	0 %	0 %	0 %
TDES-64	525	8.9	112	149.4	592	9	111.2	148.3	12.8 %	1 %	-1 %	-1 %
IDEA-64	825	17.8	56.3	450.3	887	19	52.7	421.6	7.5 %	7 %	-7 %	-6 %
AES-128	2,494	6.9	144.4	1,848.1	2,554	7.03	142.2	1,820.2	2.4 %	1.5 %	-1.5	-1.5 %
AES-192	2,936	7.1	140.7	1,500.8	2,994	7.2	138.2	1,474	2 %	1.8 %	-1.8 %	-1.8 %
AES-256	3,392	7.4	135.6	1,239.7	3,449	7.6	132.3	1,209.4	1.6 %	2.4 %	-2.4	-2.4 %

$*$ t_{pd}: Propagation delay time, f_{Max}: Maximum clock frequency, Th_{Max}: Maximum throughput

Table 3.3: Comparison between our proposed architecture and other architectures.

Design	Block cipher used	Implemented operational mode	f_{Max} MHz	Technology	Area Slice/Gates
Xinmiao [34]	AES	ECB	71.8-168.4	XCV1000	9406 Slice
Mangard [35]	AES	ECB,and CBC	295	0.18 μm CMOS	73,200 Gate
Alireza [36]	AES	ECB, CBC, CTR, and CCM	-	0.6 μm CMOS	15,493 Gate
Our Design	AES-128, AES-192, AES-256, DES, TDES, IDEA	ECB, CBC, CFB,OFB, CTR, CBC-MAC, RMAC, CCM	56.3-145.2	XC2V4000	431-3449 Slice

architecture compared with the implemented block ciphers (i.e., ECB mode) with respect to area and maximum frequency. From the table, we see that the extra area ranges from 1.6% for AES-256 to 18.4% for DES. The extra area depends on the examined cipher. The largest extra area happened in the case of DES algorithm. This is because DES has the lowest implementation area compared with the other algorithms. The delay increase ranges from 0% for DES to 7% for IDEA. The frequency reduction ranges from 0% for DES to 7% for IDEA. The throughput reduction ranges from 0% for DES to 7% for IDEA.

It is clear from Table 3.2 that the proposed unified architecture has a very low additional implementation cost for a cipher integration. The higher implementation cost occurred in the case of DES implementation. This is due to the fact that DES allocates the least area resources compared to the other block ciphers as shown in Table 3.2.

Table 3.3 compares the hardware implementation results of our design to other designs. As expected, the ASIC performs better, however, it is not reconfigurable. Our FPGA-based design performs better than the other FPGA implementations regarding the area because of two factors: used technology and improved architecture. Moreover, Table 3.3 compares also our design to others regarding the number of operational modes implemented. None of the other cases support all eight modes of operation, which is to our advantage.

3.5 Chapter Summary

The security strength of a network transaction is based on used powerful block ciphers. In addition to the original mode of operation (ECB), encryption algorithms need to perform a variety of operational modes. In this chapter, a novel multi-mode architecture is proposed, which supports the ciphers' specified operational modes

in only one hardware device. Using a common architecture, the implementation cost is examined for the most widely used block cipher: DES, TDES, AES-128, AES-192, AES-256, and IDEA. The proposed architecture is tested using Xilinx XC2V4000-6-bf957 FPGA. The area, maximum delay, maximum frequency, and throughput of the proposed architecture is compared to the simplest mode (ECB). The shown experimental results show that the additional implementation costs of all the studied block ciphers are very low for the proposed multi-mode architecture. Therefore, the proposed architecture additional costs are negligible compared with its huge advantages.

The unified architecture requires a block cipher that is optimized for VoIP. In the next chapter, we propose a high speed, deep-pipelined architecture for AES algorithm based on the composite field approach targeting VoIP applications.

Chapter 4

A High-Speed, Deep-Pipelined Architecture for Real-Time AES

This chapter presents the second contribution of this dissertation. We propose a high speed deep-pipelined architecture for the AES algorithm based on the composite field approach targeting VoIP applications.

The main contributions in this chapter are summarized as follows. We modified the algorithm in [39] for finding the *isomorphic* mapping matrix to work for any irreducible polynomial, not only the primitive polynomials. Moreover, we used the modified algorithm to find the optimum matrix that gives the minimum delay (2 XOR gates in the critical path for *isomorphic* mapping and 2 XOR gates in the critical path for inverse *isomorphic* mapping). We used this matrix to implement the *SubBytes /InvSubBytes* transformation using composite fields, which in turn allows us to design a very high speed deep-pipelined architecture. As a result of using the optimized matrix, we achieved a processing throughput of 49.401 Gbps, which is two times faster than the fastest design introduced before.

Another contribution in this chapter is the separation of the encryption circuit

from the decryption circuit to allow concurrent encryption and decryption, which facilitates full duplex encryption/decryption for VoIP applications. Although the separated implementation increase the required area, it saves three-multiplexor delay in each round, which in turn, saves a total of 30 multiplexor in the critical path. As a result, the critical path delay is dramatically decreased.

This chapter is organized as follows: Section 4.1 introduces this chapter. In Section 4.2, we briefly describe the advanced encryption standard (AES) algorithm and the composite fields concept. In Section 4.3 the optimum isomorphic mapping parameters are calculated, then the detailed hardware architecture for every non-trivial transformation and the key expansion unit are introduced. Implementation results and the comparison with prior efforts are introduced in Section 4.4. Section 4.5 summarizes the chapter.

4.1 Introduction

Internet rapid expansion resulted in an increase in the transport of real-time IP communication traffic (e.g., Voice over IP (VoIP), Video over IP, video conferencing, interactive distance learning, etc.). Such real-time applications suffer security threats due to the open nature of the Internet. IP packets traveling across the Internet are not immune to interception. Therefore, they can be easily intercepted and captured using any sniffer application, which in turn makes transmitted data available to a third party. As a result, real-time applications such as VoIP are susceptible to a number of attacks, such as eavesdropping, and spoofing (e.g. service theft, toll fraud, and registration hijacking). To overcome these attacks, transmitted packets need to be encrypted. However, encryption may degrade the QoS of the transmitted voice or video traffic. Therefore, encryption time should be kept to a minimum to meet the QoS requirements of these applications. The most attractive and suitable encryption

algorithm to secure these real-time applications is the Advanced Encryption Standard (AES) [25].The National Institute for Standards and Technology (NIST) selected the Rijndael algorithm as the official Advanced Encryption Standard (AES) in 2001 [25]. Later in 2005, NIST recommended securing VoIP using AES [6].

The AES algorithm has a simple structure and can be implemented efficiently on a wide range of general purpose processors. Although a software realization of the AES cipher can achieve high throughput when compared to other block ciphers, a hardware implementation provides physical security and higher speed. Therefore, a hardware implementation is desirable in many special purpose applications, where speed and security are important. VoIP phones, Video conferencing equipment, and network cryptographic coprocessors are examples where the primary concern is the speed of implementation rather than area requirements and power consumption. There are different techniques to speed up the hardware implementation of the AES such as: pipelining, deep-pipelining, and loop unrolling. Among these techniques, deep-pipelining can achieve maximum speedup with reasonable area-speed utilization ratio [34].

The most critical step in AES is the *SubBytes/InvSubBytes* transformation (also called S-Box) [25,94]. This transformation demands high computational power, which in turn increases the implementation area as well as the critical path delay (the critical path delay as well as the area are measured in gates). It is clear from previous implementations that the *SubBytes/InvSubBytes* transformation requires around 60% to 70% of the total delay of each round delay [34, 40, 95].

SubBytes/InvSubBytes transformation is based on finding the multiplicative inverse in $GF(2^8)$. There exist several methods to calculate multiplicative inverse in $GF(2^8)$. These methods include: Look Up Table (LUT)-based implementations and non-LUT-based implementations such as: Fermat's theorem and composite field [96].

Most AES implementations utilize a LUT for implementing *SubBytes* transfor-

mation [35, 97–100]. The LUT-based technique is not suitable for deep-pipelining implementation because it is not dividable. This in turn prohibits dividing each round to more than two sub-stages, which prevents achieving any further speed up. Although non-LUT-based implementations are suitable for deep-pipelining, they require calculating inversion in $GF(2^8)$, which in turn increases the hardware complexity [96, 101, 102].

Composite fields could be utilized such that the complex operations in $GF(2^8)$ are mapped to simpler operations in an *isomorphic* composite field such as $GF((2^4)^2)$. Composite field implementations were first proposed by one of the AES inventors in [103]. Many composite field-based AES implementations have been proposed subsequently [34, 40, 95, 104–107]. These implementations differ in how they represent the field (i.e., how they choose composite field parameters such as field polynomials). Some of them use a one level composite field ($GF((2^4)^2)$, i.e., generating $GF(2^8)$ using the *subfield* $GF(2^4)$), and the rest using two level composite fields ($GF(((2^2)^2)^2)$, i.e., generating $GF(2^8)$ using the *subfield* $GF(2^2)$). These different representations affect the transformation matrix that, in turn, affects the performance of the implementation. The advantages of composite field implementation are that the inversion in $GF(2^8)$ can be performed with almost the same gate count as the multiplication in $GF(2^8)$ and the delay is much lower than that in other techniques [39].

Satoh et al. in [95] used $GF(((2^2)^2)^2)$, which resulted in a more complex architecture and longer critical path delay than the composite field $GF((2^4)^2)$. Zhang et al. in [34] used a non-efficient *isomorphic* mapping matrix that needs 8 XOR gates in the critical path for mapping and inverse mapping. Järvinen et al. in [40] used the composite field approach for all transformations, which resulted in a complex multiplication process for the *MixColumns*/*InvMixColumns* transformation. Su et al. in [104] used complex polynomials, which resulted in a complex and slow

implementation, even though they implemented their design in ASIC.

4.2 AES and Composite Fields

In this section, we present a brief background on the Advanced Encryption Standard (AES) followed by a brief background on the composite field concept.

4.2.1 Advanced Encryption Standard (AES)

The AES algorithm is an iterative, symmetric-key block cipher that encrypts/decrypts a 128-bit block using *key* lengths of 128, 192, or 256 bit (denoted as AES-128, AES-192, AES-256, respectively) [25]. The number of iterations (also called rounds) is $N_r = 10, 12, 14$ for AES-128, AES-192, AES-256, respectively. The *key* is expanded to $N_r + 1$ keys (denoted as $RoundKey(N_r)$), one *key* for the initialization and N_r keys for N_r rounds. AES is also known as the Rijndael algorithm, named after its inventors Joan Daemen and Vincent Rijmen [94]. In our implementation, we assume a *key* length of 128 bits only as it is the most commonly used length and strong enough against well-known attacks [60, 108, 109]. Moreover, it is suitable for real-time applications, since it provides higher throughput and lower computational complexity [110]. In the AES-128 algorithm, the input block is divided into 16 bytes, then mapped onto a byte-wise 4×4 matrix, called the state array. Each byte in the state array is denoted as $S_{r,c}$ $(0 \leq r, c \leq 3)$ and considered as an element of $GF(2^8)$. All operations done in AES over $GF(2^8)$ are done modulo the irreducible polynomial

$$R(z) = z^8 + z^4 + z^3 + z + 1 \tag{4.1}$$

where, z is a root of the field polynomial $R(z)$.

Figure 4.1 shows the mapping between the input block and the state array [25,94]. In AES-128, the input block is encrypted by the repeated application of the *round*

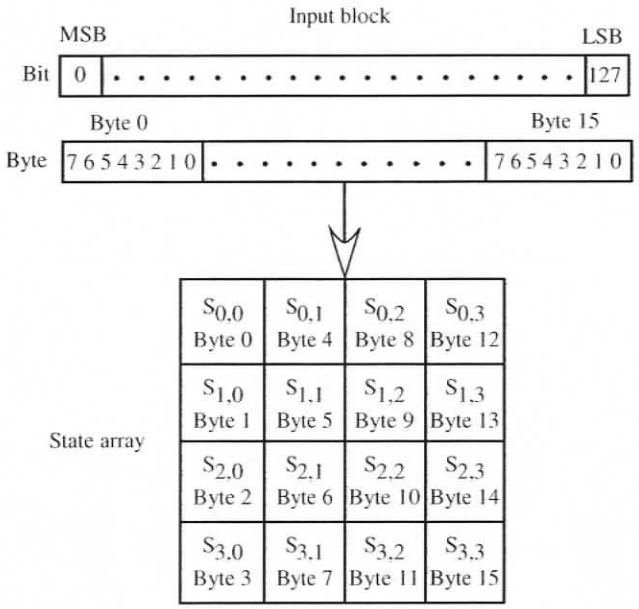


Figure 4.1: Mapping the input block onto the state array.

function to the state array. The *round* function is composed of four different byte-wise transformations:

1. *SubBytes*: Performs nonlinear byte transformation,
2. *ShiftRows*: Shifts each row of the state array,
3. *MixColumns*: Mixes data within each column of the state array, and
4. *AddRoundKey*: XORing the *RoundKey* with the state array.

Figure 4.2 illustrates the structure of the AES-128 algorithm in both encryption and decryption modes. Algorithm 1 lists the pseudo-code of the AES-128 algorithm. The encryption process is initialized by adding the first *RoundKey*, followed by applying the *round* function $N_r - 1$ times, then applying a last *round*. The last

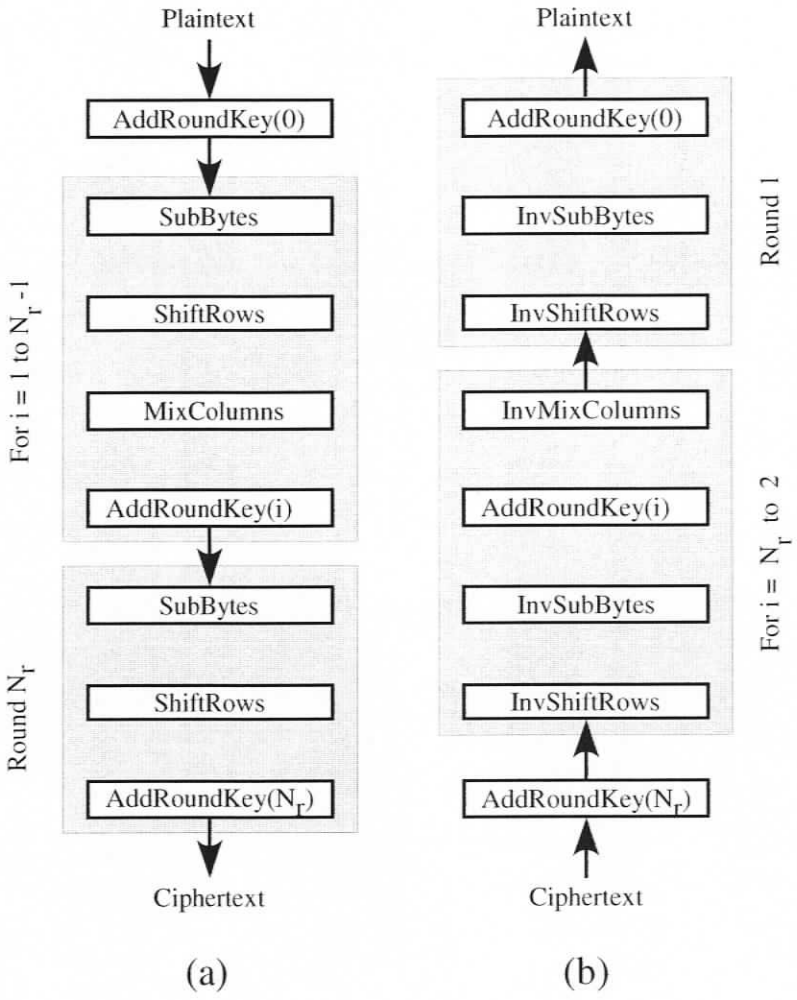


Figure 4.2: The AES algorithm (a) encryption, (b) decryption.

round is different from the first $N_r - 1$ rounds, since no *MixColumns* transformation is performed.

The *AddRoundKey* transformation is just XORing the state array with the *RoundKey* of the current *round*. The *SubBytes* transformation is a nonlinear byte substitution that operates independently on each byte. This transformation is done by finding the multiplicative inverse in $GF(2^8)$, followed by an affine transformation according to the equation

$$S^E_{r,c} = \text{SubBytes}(S_{r,c}) = S^{-1}_{r,c}B + C \tag{4.2}$$

where B is an 8×8 matrix and C is an 8×1 vector defined in [25, 94].

In the *MixColumns* transformation, each column is considered a polynomial over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{4.3}$$

where { } denotes a hexadecimal number. The *MixColumns* transformation can be written in a matrix form

$$\begin{bmatrix} s^E_{0,c} \\ s^E_{1,c} \\ s^E_{2,c} \\ s^E_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}, 0 \leq c \leq 3. \tag{4.4}$$

ShiftRows transformation is just a simple shifting function for each row, the first row is not shifted, while the second, third, and fourth rows are shifted one byte, two bytes, and three bytes to the left, respectively.

In the decryption process, the inverse of each transformation is applied to the encrypted state array in the same sequence, but in a reverse order. The *InvSubBytes* transformation is done by applying the inverse of affine transformation followed by

Algorithm 1 The AES algorithm.

Procedure $AES(InputBlock, Key, Encryption)$

```
1: State = InputBlock
2: RoundKeys = KeyExpansion(Key)
3: if Encryption = 1 then
4:    State = AddRoundKey(State,RoundKeys[0])
5:    for i = 1 to N_r − 1 do
6:       SubBytes(State)
7:       ShiftRows(State)
8:       MixColumns(State)
9:       AddRoundKey(State, RoundKeys[i])
10:   end for
11:   SubBytes(State)
12:   ShiftRows(State)
13:   AddRoundKey(State, RoundKeys[N_r])
14: else
15:   State = AddRoundKey(State,RoundKeys[N_r])
16:   for N_r − 1 downto i = 1 do
17:      InvShiftRows(State)
18:      InvSubBytes(State)
19:      AddRoundKey(State, RoundKeys[i])
20:      InvMixColumns(State)
21:   end for
22:   InvShiftRows(State)
23:   InvSubBytes(State)
24:   AddRoundKey(State, RoundKeys[0])
25: end if
26: return State
```

taking the multiplicative inverse in $GF(2^8)$ according to the equation

$$S^D_{r,c} = \text{InvSubBytes}(S^E_{r,c}) = (B^{-1}S^E_{r,c} + C^{-1})^{-1} \tag{4.5}$$

In the *InvMixColumns*, each column is multiplied by $a^{-1}(x)$ modulo $x^4 + 1$, where

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \tag{4.6}$$

This can be written in a matrix form

$$\begin{bmatrix} s^D_{0,c} \\ s^D_{1,c} \\ s^D_{2,c} \\ s^D_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}, 0 \leq c \leq 3. \tag{4.7}$$

In the *InvShiftRows* transformation, the same number of bytes is shifted to the right instead of the left.

In AES-128, the *KeyExpansion* function takes four 4-byte words (16 bytes) as input and generates a linear array of 44 4-byte words (156 bytes) as output. These 44 words form 11 keys, one for the initial round and 10 for the 10 subsequent rounds [25, 94]. Each *RoundKey* is formed by concatenating four words, i.e., $RoundKey(i)$=$(w[4i],w[4i+1],w[4i+2],w[4i+3])$. Algorithm 2 lists the pseudo-code of the AES-128 *KeyExpansion* function. In Algorithm 2 *SubWord* applies *SubBytes* transformation to each of the 4 bytes in a word, whereas *RotWord* performs one byte circular left shift on a word. The *Rcon* is a round constant word array with nonzero leftmost bytes in each word [25, 60, 94].

4.2.2 Composite Fields Arithmetic

Although *Galois fields* of the same order are *isomorphic*, the complexity of the field operations heavily depend on the representations of the field elements [39]. Composite

Algorithm 2 The key expansion algorithm.

Procedure $KeyExpansion(wordKey[4], wordw[44])$

1: **for** $i = 0$ to 3 **do**
2: w[i] = Key[i]
3: **end for**
4: **for** $i = 4$ to 43 **do**
5: **if** $i \bmod 4 = 0$ **then**
6: Temp = SubWord(RotWord(w[$i-1$])) $\bigoplus$ Rcon[$i/4$]
7: **else**
8: Temp = w[$i-1$]
9: **end if**
10: w[i] = w[$i-4$] $\bigoplus$ Temp
11: **end for**

field arithmetic can be employed to reduce the hardware complexity [39]. The two pairs

$$\begin{aligned} GF(2^n), \quad & Q(y) = y^n + \Sigma_{i=0}^{n-1} q_i y^i, q_i \in GF(2) \\ GF((2^n)^m), \quad & P(x) = x^m + \Sigma_{i=0}^{m-1} p_i x^i, p_i \in GF(2^n) \end{aligned} \tag{4.8}$$

are called a composite field if

- $GF(2^n)$ is constructed from $GF(2)$ by $Q(y)$;
- $GF((2^n)^m)$ is constructed from $GF(2^n)$ by $P(x)$.

The composite field $GF((2^n)^m)$ is *isomorphic* to the field $GF(2^k)$, where $k = nm$. The field $GF(2^n)$ is called *subfield* while the field $GF(2^k)$ is called *extension field*. The irreducible polynomial that defines multiplication and addition over the *extension field* is called *extension field irreducible* and is denoted by $R(z)$. The *subfield irreducible* defines multiplication and addition over the *subfield* and is denoted by $Q(y)$. Finally, the irreducible polynomial that defines multiplication and addition over the composite field is called composite field irreducible and is denoted by $P(x)$. To work with the

composite field representation, an element in the *extension field* should be mapped to its equivalent, one in the composite field before doing any calculation and then the resultant composite field element should be mapped again to its equivalent in the *extension field*. This *isomorphic* mapping and its inverse are done by a $k \times k$ matrix T and its inverse T^{-1} (elements of T and T^{-1} are $\in GF(2)$).

The AES irreducible polynomial is given in Equation (4.1). We have chosen the composite field representation $GF((2^4)^2)$ because it is simpler than $GF(((2^2)^2)^2)$ i.e., it is faster and requires less area [39]. To minimize the cost of multiplication in the *subfield*, the irreducible polynomial for the *subfield* $GF(2^4)$ has been chosen as

$$Q(y) = y^4 + y + 1 \tag{4.9}$$

Therefore, the composite field irreducible $P(x)$ must be a polynomial of degree 2 whose coefficients are elements of $GF(2^4)$

$$P(x) = x^2 + Ax + B, \qquad A, B \in GF(2^4) \tag{4.10}$$

The parameters A and B should be chosen such that $P(x)$ is irreducible. The optimum parameters (A, B, T, T^{-1}) that give minimum delay will be calculated in Subsection 4.3.1.

4.3 Proposed Hardware Architecture

In our proposed AES architecture we adopted a pipelining approach to increase the AES throughput. Moreover, we adopted a composite field implementation for *SubBytes /InvSubBytes* transformation to decrease the critical path delay. The AES proposed architecture is improved by the deep-pipelined architecture shown in Figure 4.3. In this architecture, each round has a separate hardware (called round unit).

In our architecture we adopted a different technique in deep-pipelining implementation in which we calculated the total critical path delay in gate count and then divided the critical path into equal delay deep-pipelines stages. The number of deep-pipelined stages was chosen such that we achieve the maximum throughput with acceptable pipeline latency. Moreover, we separated the encryption circuit from the decryption circuit to allow concurrent encryption and decryption, which facilitates full duplex voice encryption/decryption. Additionally, the separated implementation saves three-multiplexor delays in each round, which in turn saves 30 multiplexors in the critical path. Moreover, we implemented the *SubBytes* transformation in a composite fields to make it suitable for deep-pipelined architecture. To speed up the *SubBytes* transformation in composite field, we generated all possible transformation matrices and chose the optimum one for speed. Moreover, we sped up the *MixColumns* transformation by choosing a different arrangement of the elements (as will discussed in Section 4.3.3).

In this section, we calculate the optimum composite fields parameters (A,B,T,T^{-1}) and then we present a detailed hardware architecture for every non-trivial transformation in the AES. The architecture is optimized for speed, as we are targeting VoIP applications such as VoIP and video conferencing. The *SubBytes* /*InvSubBytes* and *MixColumns*/*InvMixColumns* transformations are introduced in detail. The key expansion unit is also introduced in detail. The optimized deep-pipeline architecture is introduced based on the analysis of the gate counts in the critical path.

4.3.1 Optimum Composite Fields Parameters

The irreducible polynomials that generate the composite field $GF((2^4)^2)$ are given in Equations 4.9 and 4.10. In Equation (4.10), the parameters A and B should be chosen such that $P(x)$ is irreducible. For each value of the pair A, B, which makes

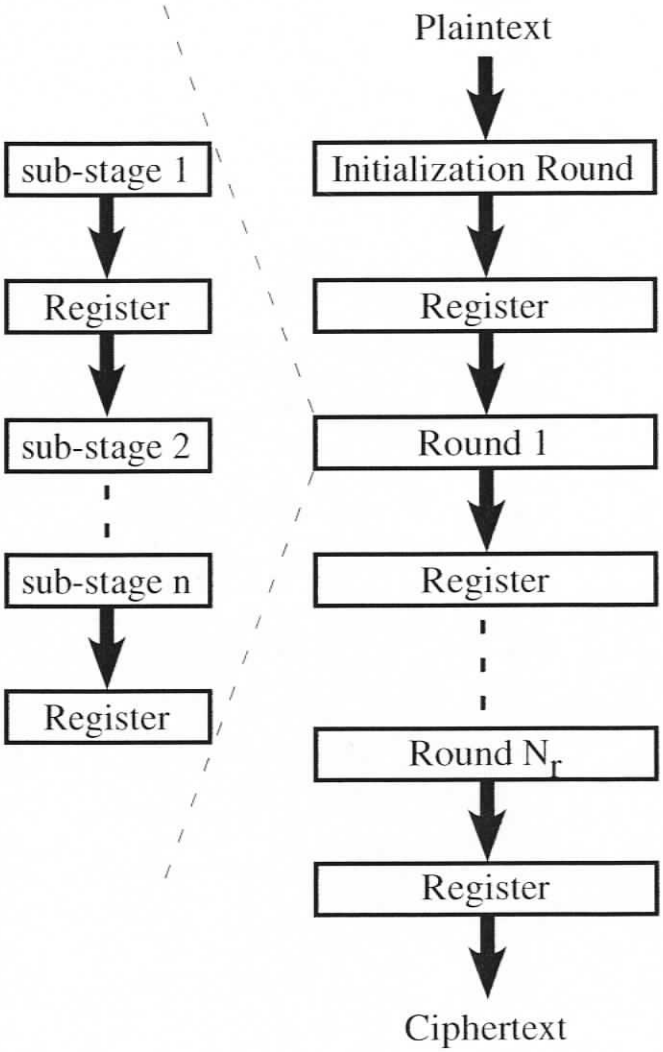


Figure 4.3: The AES deep-pipelined architecture.

$P(x)$ irreducible, there exists 8 possible roots for $R(z)$. Each root is associated with a mapping and inverse mapping pairs of matrices T and T^{-1} [39]. There are 120 irreducible polynomials of degree 2 whose coefficients are elements of $GF(2^4)$ [111]. Out of these 120 polynomials, there are 64 primitive polynomials. Thus, the total possible solutions (i.e., T and T^{-1}) are $8 \times 120 = 960$. For calculating the *isomorphic* mapping matrices (T and T^{-1}), we expanded the algorithm in [39] to work for irreducible polynomials as well as primitive polynomials. Our modified algorithm is listed in Algorithm 3.

Based on our expanded algorithm, we wrote a MATLAB code to produce the entire possible *isomorphic* mapping matrices and their inverses. We found that one solution of the 8 possible solutions for $A = \omega^2 = 4$ and $B = \omega^0 = 1$ has a minimum delay (where ω is the generator element of the *subfield* $GF(2^4)$). This minimum delay corresponds to the mapping matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{4.11}$$

The inverse mapping matrix T^{-1} is the inversion of T with coefficients taken modulo

Algorithm 3 Composite field optimum parameters.

Procedure $CFOP(R(z), P(x)[N], Q(y)$

R[1:8] = Roots(R(z)) {Find the 8 roots of R(z) as in [39]}
K=1
for $i = 1$ to N **do**
    IP = P(x)[i] {Select an irreducible polynomial from the list of all possible polynomials }
    A,B = Coefficients(IP) {Extract parameters A and B}
    for $j = 1$ to 8 **do**
        r = R[j] {Select a root from the root list}
        P[1:8] = Powers(r) {Find r^0,...r^7 using IP, as in [39]}
        $T = [r^0\ r^1\ r^2\ r^3\ r^4\ r^5\ r^6\ r^7]$ {As in [39]}
        T^{-1} = Inverse(T) {Mathematical inverse of T with every element calculated modulo 2}
        δ_A = Delay(A) {Critical path delay to multiply by A}
        δ_B = Delay(B) {Critical path delay to multiply by B}
        δ_T = Delay(T) {*Isomorphic* mapping critical path delay}
        $\delta_{T^{-1}}$ = Delay(T^{-1}) {Inverse *isomorphic* mapping critical path delay}
        δ_{total} = MAX(δ_A, δ_B) + δ_T + $\delta_{T^{-1}}$
        Result[k] = A,B,δ_{total} {Store the results}
        K++
    end for
end for
$[A, B]_{optimum}$ = Optimum(Result) {Find optimum parameters}
return A, B, T, T^{-1}

2 as follows:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{4.12}$$

4.3.2 Implementation of SubBytes/InvSubBytes Transformation

SubBytes /InvSubBytes is based on finding the multiplicative inverse in $GF(2^8)$. There exist several methods to calculate the multiplicative inverse in $GF(2^8)$. These methods include: the Look Up Table (LUT)-based implementations, Fermat's theorem-based implementations, and composite field-based implementations [96].

Most AES implementations utilize a LUT for implementing *SubBytes* transformation [35, 97–100]. These LUTs could be implemented using an internal memory located on most of the recent FPGA chips (also called Block RAM). Although this technique has obvious benefits, it does not exploit the mathematical structure of the multiplicative inverse transformation. Moreover, it is not suitable for deep-pipelined implementations for two reasons. First, the amount of available internal memory (Block RAM) may become a bottleneck, as pipelined architecture is based on duplicating the *round* hardware. Secondly, the LUT technique is not suitable for deep-pipelining implementations since a LUT is not dividable.

Few implementations based on Fermat's theorem inversion are introduced in [101, 102]. This technique is based on raising the element to the 254^{th} power

in $GF(2^8)$. In [101] the overall area is 386 XOR and 234 AND gates, and the latency is a 38 gate delay. On the other hand, the total latency in [102] is a 25 gate delay. The disadvantages of this technique are that the area is at least three times that of the $GF(2^8)$ multiplier and the latency is higher than the composite field implementation [96]. Moreover, this implementation calculates the inversion only without affine transformation, which in turn needs more area and delay.

Our approach utilizes the composite field technique to calculate the multiplicative inverse required for *SubBytes* in $GF((2^4)^2)$ instead of $GF(2^8)$. In Subsection 4.3.1 we calculated the optimum *isomorphic* mapping matrix T and its inverse. In composite field $GF((2^4)^2)$, an element $E \in GF(2^8)$ can be expressed as $E(x) = e_h x + e_l$, where e_h, $e_l \in GF(2^4)$, e_h, e_l represents the higher and lower four bits respectively, and x is the root of $P(x)$. Using the Extended Euclidean algorithm, the multiplicative inverse of $E(x)$ modulo $P(x)$ can be calculated using

$$(e_h x + e_l)^{-1} = \frac{e_h x + (e_h A + e_l)}{e_l^2 + e_h e_l A + e_h^2} \tag{4.13}$$

The proof of this equation is given in Appendix A. The performance of the hardware system that implements Equation (4.13) depends on the method used to evaluate the terms on the right hand side. In other words, the implementation of Equation (4.13) depends on how the denominator is arranged. We have three possible arrangements listed below in Equations 4.14, 4.15, and 4.16

$$(e_h x + e_l)^{-1} = \frac{e_h x + (e_h A + e_l)}{e_l^2 + e_h e_l A + e_h^2} \tag{4.14}$$

$$(e_h x + e_l)^{-1} = \frac{e_h x + (e_h A + e_l)}{e_l (e_h A + e_l) + e_h^2} \tag{4.15}$$

$$(e_h x + e_l)^{-1} = \frac{e_h x + (e_h A + e_l)}{e_l^2 + e_h (e_h + e_l A)} \tag{4.16}$$

The different implementations of these different arrangements are shown in Figure 4.4. The implementations of these equations require some operations in

Table 4.1: Area and critical path delay for the different techniques used to implement inverse in $GF(2^4)$.

Technique	Area	Delay
Composite field	12 AND + 16 XOR	2 AND + 7 XOR
Fermat's theorem	48 AND + 57 XOR	2 AND + 6 XOR
Direct	10 AND + 18 XOR	2 AND + 3 XOR
LUT	24 XOR	3 XOR

$GF(2^4)$. These operations are: multiplication, multiplicative inversion, addition, squaring, and constant multiplication. Moreover, we need to apply *isomorphic* mapping before working with $GF(2^4)$, then apply inverse *isomorphic* mapping after that. A multiplier in $GF(2^4)$ could be implemented either by composite field (i.e., implement the multiplication in $GF((2^2)^2)$) or by direct implementation. The composite field implementation requires 21 XOR and 9 AND gates with the critical path consisting of an AND and 4 XOR gates, while the direct implementation requires 16 XOR and 16 AND gates with a critical path of an AND and 3 XOR gates. Thus, we select the direct implementation because it is faster, although it needs slightly more area. The multiplicative inversion in $GF(2^4)$ could be implemented by either composite field, Fermat's theorem, direct implementation, or by using LUTs. The area and delay for each of these techniques are listed in Table 4.1. The area and speed of the LUT implementation is calculated by implementing the LUT on a Xilinx XC2V6000FF1152-6 FPGA chip, which is the target chip of our final implementation [33]. The LUT implementation is the optimum circuit for finding inverse in $GF(2^4)$. Therefore, we use a LUT for inversion circuit. The addition, squaring, and constant multiplication are trivial and straight forward. The area and delay of all required operations are listed in Table 4.2.

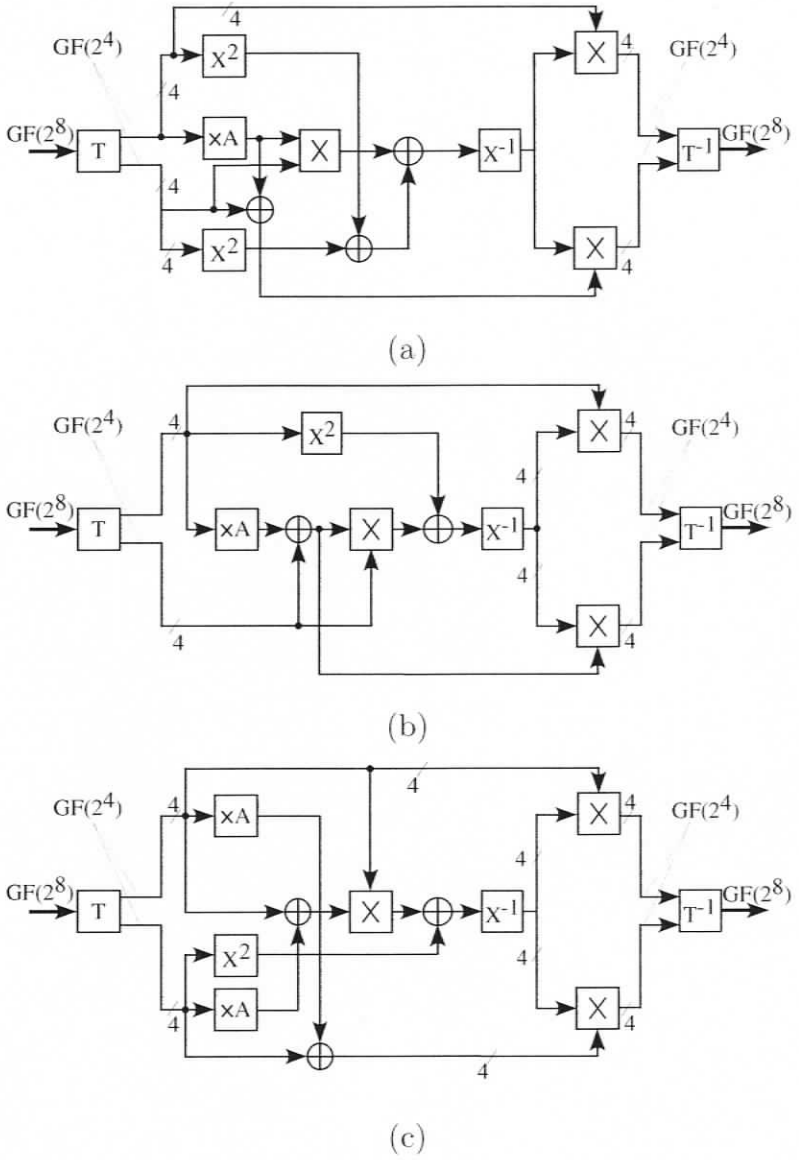


Figure 4.4: Three different multiplicative inverse implementations: (a) Equation (4.14) (b) Equation (4.15) (c) Equation (4.16). $\bigoplus$: adder in $GF(2^4)$, $\times A$: constant multiplier in $GF(2^4)$, $\times$: multiplier in $GF(2^4)$, X^{-1}: multiplicative inverter in $GF(2^4)$, X^2: squarer in $GF(2^4)$.

Table 4.2: Area and critical path delay measured in gate count of functional blocks in the *SubBytes* transformation (Figure 4.4).

Block	Meaning	Area	Delay
$\oplus$	Adding in $GF(2^4)$	4 XOR	1 XOR
$\times A$	Multiplying by constant A	2 XOR	1 XOR
X	Multiplier in $GF(2^4)$	16 AND + 16 XOR	AND + 3 XOR
X^2	Squaring in $GF(2^4)$	2 XOR	XOR
X^{-1}	Multiplicative inverse in $GF(2^4)$	24 XOR	3 XOR
T	*Isomorphic* mapping	12 XOR	2 XOR
T^{-1}	Inverse *isomorphic* mapping	10 XOR	2 XOR
$T^{-1}\times$ Affine	Combined inverse mapping and affine	19 XOR	3 XOR
Inverse Affine$\times T$	Combined inverse affine and mapping	21 XOR	3 XOR

Table 4.3: Area and critical path delay measured in gate count of three possible implementation of the multiplicative inverse.

Equ.	Fig.	Area	Delay
4.14	4.4(a)	48 AND + 112 XOR	15 XOR + 2 AND
4.15	4.4(b)	48 AND + 106 XOR	16 XOR + 2 AND
4.16	4.4(c)	48 AND + 112 XOR	16 XOR + 2 AND

Based on the area and delay of the required operations listed in Table 4.2, we calculated the area and delay for the three different implementations of the multiplicative inverse in $GF(2^8)$. Table 4.3 shows the area and delay performance of the three different implementations. From Table 4.3 we observed that Implementation (a) shows the smallest delay. Since we are targeting VoIP applications, we will choose the fastest implementation, which is Implementation (a). Although, it shows an improvement by one gate delay, it saves a 10 gate delay in the overall critical path since the multiplicative inverse will be used 10 times in 10 consecutive rounds.

The *SubBytes /InvSubBytes* transformation is implemented according to the block diagram illustrated in Figure 4.5. Combining inverse *isomorphic* mapping with affine transformation and inverse affine transformation with *isomorphic* mapping will save area and increase speed. The combined inverse *isomorphic* mapping and affine transformation requires 19 XOR gates with 3 XOR gates in the critical path, whereas the combined inverse affine transformation and *isomorphic* mapping requires 21 XOR gates with 3 XOR gates in the critical path.

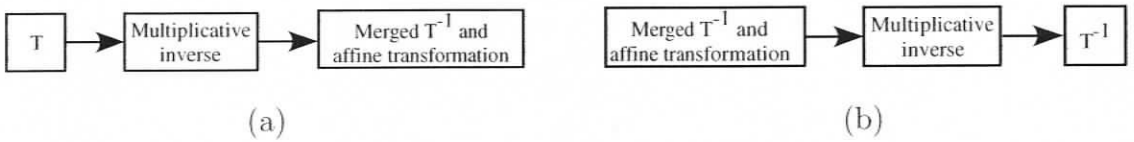


Figure 4.5: Block diagram of *SubBytes* transformation: (a) encryption (b) decryption.

4.3.3 MixColumns/InvMixColumns Transformation

Different architectures of *MixColumns/InvMixColumns* have been introduced [34, 36, 99, 100]. Our optimization for this transformation is based on rearranging Equation (4.4) as follows:

$$\begin{aligned} s^E_{0,c} &= \{02\}[s_{0,c} + s_{1,c}] + [s_{2,c} + s_{3,c} + s_{1,c}] \\ s^E_{1,c} &= \{02\}[s_{1,c} + s_{2,c}] + [s_{3,c} + s_{0,c} + s_{2,c}] \\ s^E_{2,c} &= \{02\}[s_{2,c} + s_{3,c}] + [s_{0,c} + s_{1,c} + s_{3,c}] \\ s^E_{3,c} &= \{02\}[s_{3,c} + s_{0,c}] + [s_{1,c} + s_{2,c} + s_{0,c}] \end{aligned} \tag{4.17}$$

The multiplication $\{02\}$ is done in $GF(2^8)$ modulo $R(z)$ (Equation (4.1)) and is called *XTime()* [25]. If an element D in $GF(2^8)$ can be expressed as $D = d_7x^7 + d_6x^6 + d_5x^5 + d_4x^4 + d_3x^3 + d_2x^2 + d_1x^1 + d_0$, the *XTime()* function can be written as

$$\begin{aligned} E &= \text{XTime(D)} = x \bullet D(\text{modulo } R(z)) \\ &= x \bullet [d_7x^7 + \ldots + d_1x^1 + d_0](\text{modulo } R(z)) \\ &= d_7x^8 + \ldots + d_1x^2 + d_0x^1(\text{modulo } R(z)) \\ &= d_6x^7 + d_5x^6 + d_4x^5 + (d_3 + d_7)x^4 \\ &+ (d_2 + d_7)x^3 + d_1x^2 + (d_0 + d_7)x^1 + d_7 \end{aligned} \tag{4.18}$$

The *XTime* function uses 3 XOR gates with one XOR gate in the critical path. Figure 4.6-a shows the implementation of the *MixColumns* transformation for one column. The *MixColumns* transformation uses 108 XOR gates with 3 XOR gates in

the critical path, which is similar to that provided by Zhang et al. in [34]. Similarly, Equation (4.7) can be rewritten as

$$\begin{aligned} s^D_{0,c} &= \{08\}[F] + \{04\}[s_{0,c} + s_{2,c}] + s^E_{0,c} \\ s^D_{1,c} &= \{08\}[F] + \{04\}[s_{1,c} + s_{3,c}] + s^E_{1,c} \\ s^D_{2,c} &= \{08\}[F] + \{04\}[s_{2,c} + s_{0,c}] + s^E_{2,c} \\ s^D_{3,c} &= \{08\}[F] + \{04\}[s_{3,c} + s_{1,c}] + s^E_{3,c} \end{aligned} \tag{4.19}$$

where $F = s_{0,c} + s_{1,c} + s_{2,c} + s_{3,c}$, and $s^E_{0,c}$, $s^E_{1,c}$, $s^E_{2,c}$, and $s^E_{3,c}$ are the *MixColumns* results found from Equation (4.17).

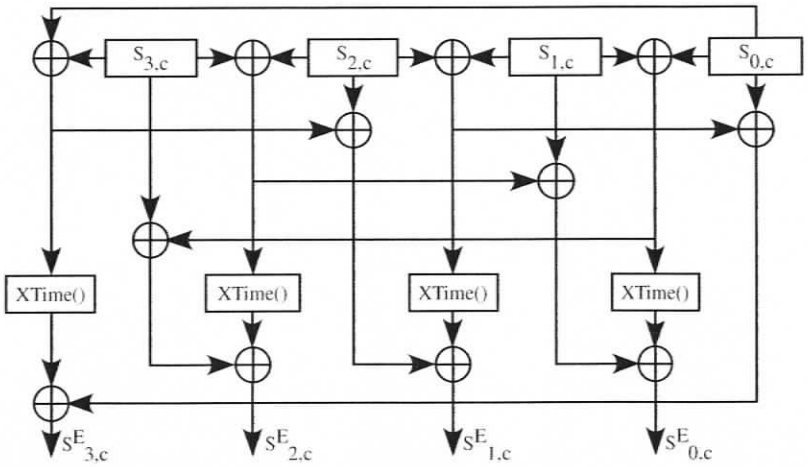
Multiplying a field element by $\{04\}$ is identical to multiplying by x^2 and it is done in $GF(2^8)$ modulo $R(z)$ (Equation (4.1)). The function *XTime2()* will do this multiplication as follows:

$$\begin{aligned} E &= \text{XTime2(D)} = x^2 \bullet D(\text{modulo } R(z)) \\ &= x^2 \bullet [d_7x^7 + \ldots + d_1x^1 + d_0](\text{modulo } R(z)) \\ &= d_7x^9 + \ldots + d_1x^3 + d_0x^2(\text{modulo } R(z)) \\ &= d_5x^7 + d_4x^6 + (d_3 + d_7)x^5 + (d_2 + d_6 + d_7)x^4 \\ &+ (d_1 + d_6)x^3 + (d_0 + d_7)x^2 + (d_6 + d_7)x^1 + d_6 \end{aligned} \tag{4.20}$$

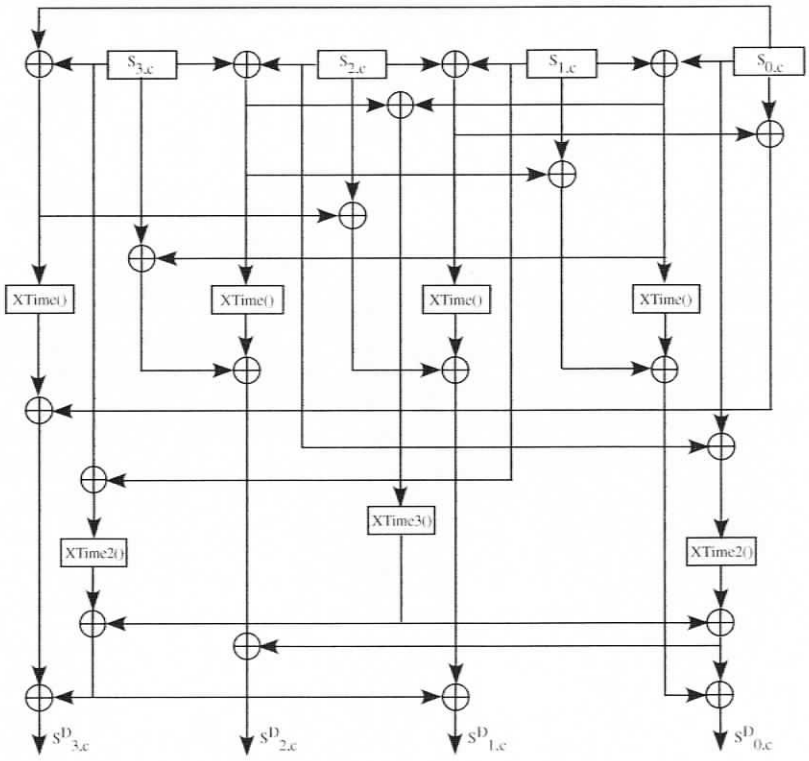
The *XTime2()* function uses 5 XOR gates with 2 XOR gates in the critical path.

Similarly, multiplying a field element by $\{08\}$ is identical to multiplying by x^3 and it is done in $GF(2^8)$ modulo $R(z)$ (Equation (4.1)). The function *XTime3()* will do this multiplication as follows:

$$\begin{aligned} E &= \text{XTime3(D)} = x^3 \bullet D(\text{modulo } p(x)) \\ &= x^3 \bullet [d_7x^7 + \ldots + d_1x^1 + d_0](\text{modulo } R(z)) \\ &= d_7x^{10} + \ldots + d_1x^4 + d_0x^3(\text{modulo } R(z)) \\ &= d_4x^7 + (d_3 + d_7)x^6 + (d_2 + d_6 + d_7)x^5 + (d_1 + d_6 + d_5)x^4 \\ &+ (d_0 + d_5 + d_7)x^3 + (d_6 + d_7)x^2 + (d_5 + d_6)x + d_5 \end{aligned} \tag{4.21}$$



(a)



(b)

Figure 4.6: Implementation of: (a) *MixColumns* transformation (b) *InvMixColumns* transformation. all buses are 8 bits in width.

The *XTime3()* function uses 7 XOR gates with 2 XOR gates in the critical path. Figure 4.6-b shows the implementation of *InvMixColumns* transformation for one column. The *InvMixColumns* transformation is implemented by using 197 XOR gates with 5 XOR gates in the critical path, as compared to 7 in [34].

4.3.4 Key Expansion Unit

RoundKeys can be either calculated on the fly or in advance and stored in a register file. Generating *RoundKey* on the fly is suitable for the application where the key is changed frequently. However, since the *RoundKey* are used in a reverse order, the decryption process can start only after the last *RoundKey* is generated, which in turn increases the decryption delay.

Since we are targeting VoIP applications, the key does not change frequently. Therefore, we generate *RoundKeys* and store them in a register file. In the case of changing the key frequently, the *RoundKeys* could be calculated while the device is not in operation. This technique saves the delay required for key calculations since the *RoundKeys* are ready for processing. As a result, the delay for encryption and decryption will almost be the same. Moreover, this technique saves a lot of area since there is no need to duplicate the key expansion circuit for each round. As a result, there is no need to use the deep-pipelining technique with the key expansion unit, which in turn saves the registers required for deep-pipelining. Figure 4.7 illustrates the key expansion unit used for *RoundKeys* generation. The key expansion unit consists of: *next key logic*, 16×8 ROM, modulo 11 counter, 4×16 decoder, *RoundKey* register file and *KeyIn* register. The role of each components is as follow:

- *next key logic*: Figure 4.8 illustrates this unit. This unit receives round constant *Rcon(i)* and the previous *RoundKey* (*RK(i-1)*) and produces next *RoundKey* (*RK(i)*) as output according to Algorithm 2.

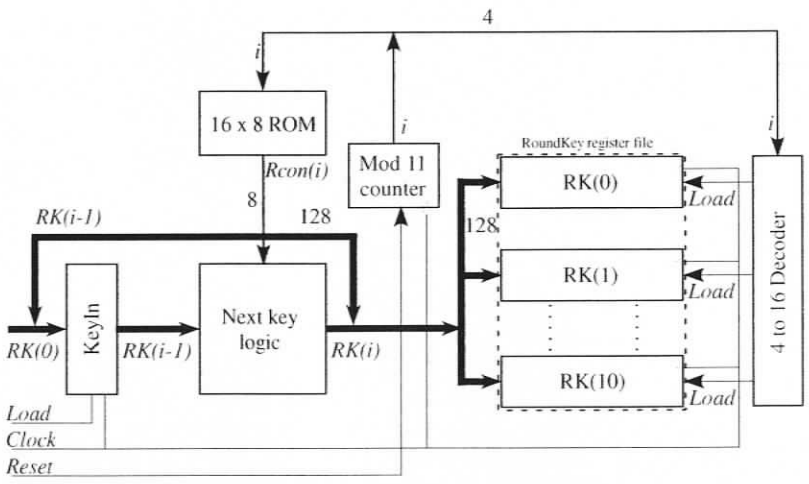


Figure 4.7: The key expansion unit.

- 16×8 ROM: This ROM stores 10 round constants (*Rcon(i)*) required to calculate *roundkeys*. The round constants are located from address 1 to address 11.

- Modulo 11 counter: This counter produces the current round number i, which is used as the address of the current round constant *Rcon*. The counter output is also used to feed the decoder with the round number i to activate the proper *load* signal. It is clear that the *RK(0)* is the original key.

- 4 × 16 decoder: The decoder receives the round number i as an input and outputs the proper *Load* signal to store *RK(i)* in its assigned register.

- *RoundKey* register file: It consists of 11 registers to store 11 keys, one key for each round (*RK(0)-RK(10)*), where the initial *RoundKey* is the original key. Therefore, there is no need to generate it because it is directly stored at the first register on the first clock cycle.

- *KeyIn* register: This register stores the previous *RoundKey RK(i-1)*.

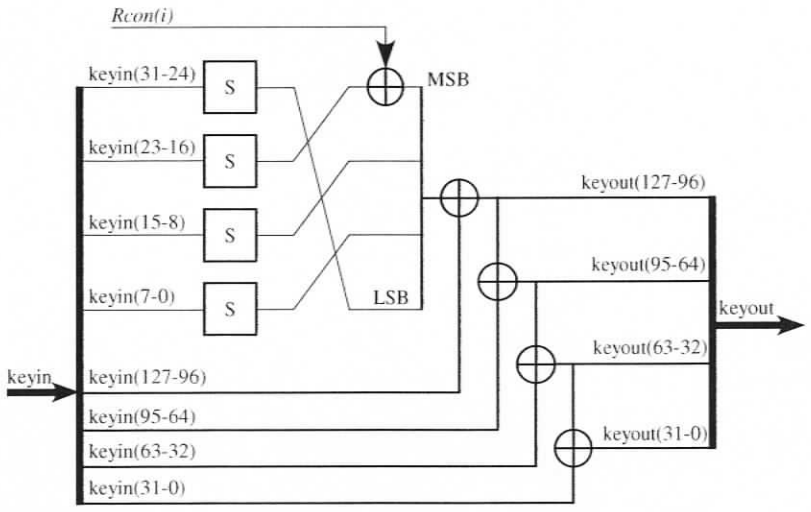


Figure 4.8: The next key logic. S: *SubBytes* transformation. *Rcon*: round constant.

The *Key expansion unit* generates a *RoundKey* each clock cycle. Initially, *Reset* is asserted and the original key is fed to the input. As a result, the counter outputs 0, which activates the *Load* signal for the *RK(0)* register and selects address 0 of the ROM. At the first positive edge, the original key is stored directly to the *RK(0)* register and the *KeyIn* register simultaneously. At the first negative edge, the counter is incremented to 1, which activates the *Load* signal for the *RK(1)* register and selects address 1 of the ROM. As a result, the ROM outputs the first round constant *Rcon(1)*. Before the second positive edge, the *next key logic* calculates the *RK(1)*. This process is repeated 10 times to generate the 10 required *RoundKeys*. As a result we will have 11 *RoundKeys* (10 generated plus the original), one key for each round.

4.3.5 Deep-pipelining Architecture

Maximum speedup could be achieved by dividing each round to equal-delay deep-pipelining. Based on the gate count analysis given in Sections 4.3.2 and 4.3.3, we

Table 4.4: Critical path delay measured in gate count for AES round in both encryption and decryption.

Transformation	Encryption	Decryption
AddRoundKey	XOR	XOR
MixColumns/InvMixColumns	3 XOR	5 XOR
SubBytes/InvSubBytes	16 XOR + 2 AND	16 XOR + 2 AND
ShiftRows/InvShiftRows	0	0
Total round delay	20 XOR + 2 AND gates	22 XOR + 2 AND gates

found that the critical path delay includes 20 XOR and 2 AND gates for encryption and 22 XOR + 2 AND gates for decryption. These critical path delays are illustrated in Table 4.4. We tried three deep-pipelined architectures of equal critical path delay (measured in gates). These deep-pipelined architectures have an 8, 6, and 4 gate length for each deep-pipeline stage, which is in turn equivalent to 3, 4, and 6 sub-stages per round, respectively (i.e., $S = 3, 4$, and 6, respectively).

We used equal-delay deep-pipeline stages for both encryption and decryption to get the same performance for both of them, since they are separated from each other. This will also allow the encryption and decryption to take place at the same time.

4.4 Experimental Results and Comparison

To evaluate our proposed architecture, we first select the modeling language, tools, and implementation platform. Then we compare the resulting performance parameters (maximum frequency, throughput, latency, and area) with other previously implemented architectures.

4.4.1 Experimental Setup

To evaluate our proposed architecture, we implemented the AES-128 architecture in five different ways. These ways are: (1) Implementation without pipelining ($S = 0$), (2) Implementation with 1 stage/round pipelining ($S = 1$), (3) Implementation with 3 stages/round deep-pipelining ($S = 3$), (4) Implementation with 4 stages/round deep-pipelining ($S = 4$), (5) Implementation with 6 stages/round deep-pipelining ($S = 6$).

We modeled these five different implementations using VHDL and then implemented them on an FPGA kit. Xilinx ISE 8.1 is used to synthesize the design and provide post-placement area and timing results. The architecture is implemented on a Xilinx XC2V6000FF1152-6 FPGA device.

4.4.2 Results and Comparison

The FPGA synthesis results for the different implementations are listed in Table 4.5. In a pipelined architecture a plaintext/ciphertext block is received each clock cycle. After a delay of $S \times Nr + 1$ clock cycles, the corresponding ciphertext/plaintext block appears at the output. The deep-pipelined architecture generates data block at each clock cycle, which in turn increases the amount of the throughput. The throughput is calculated according to [112]

$$Throughput = \frac{Frequency \times 128\ bit}{Cycles\ per\ 128\ Block} \tag{4.22}$$

Table 4.5 shows the results for different implementations. It is clear from the table that increasing the number of deep-pipelined stages increases the throughput, but will in turn increase the latency as shown in Figure 4.9-c. The latency is calculated

Table 4.5: Performance results of our proposed architecture implementations.

Implementation	Stages/round (S)	Frequency (MHz)	Throughput (Mbps)	Latency (ns)	Area (slices)	Device utilization (Mbps/slices)
No pipelining	$S = 0$	149.424	1912	73.6	2341	0.74
1 stage per round	$S = 1$	143.649	18387	76.6	11586	1.59
3 stages deep-pipelining	$S = 3$	228.310	29223	135	14078	2.08
4 stages deep-pipelining	$S = 4$	286.903	36723	143	15204	2.42
6 stages deep-pipelining	$S = 6$	385.951	49401	158	17479	2.83

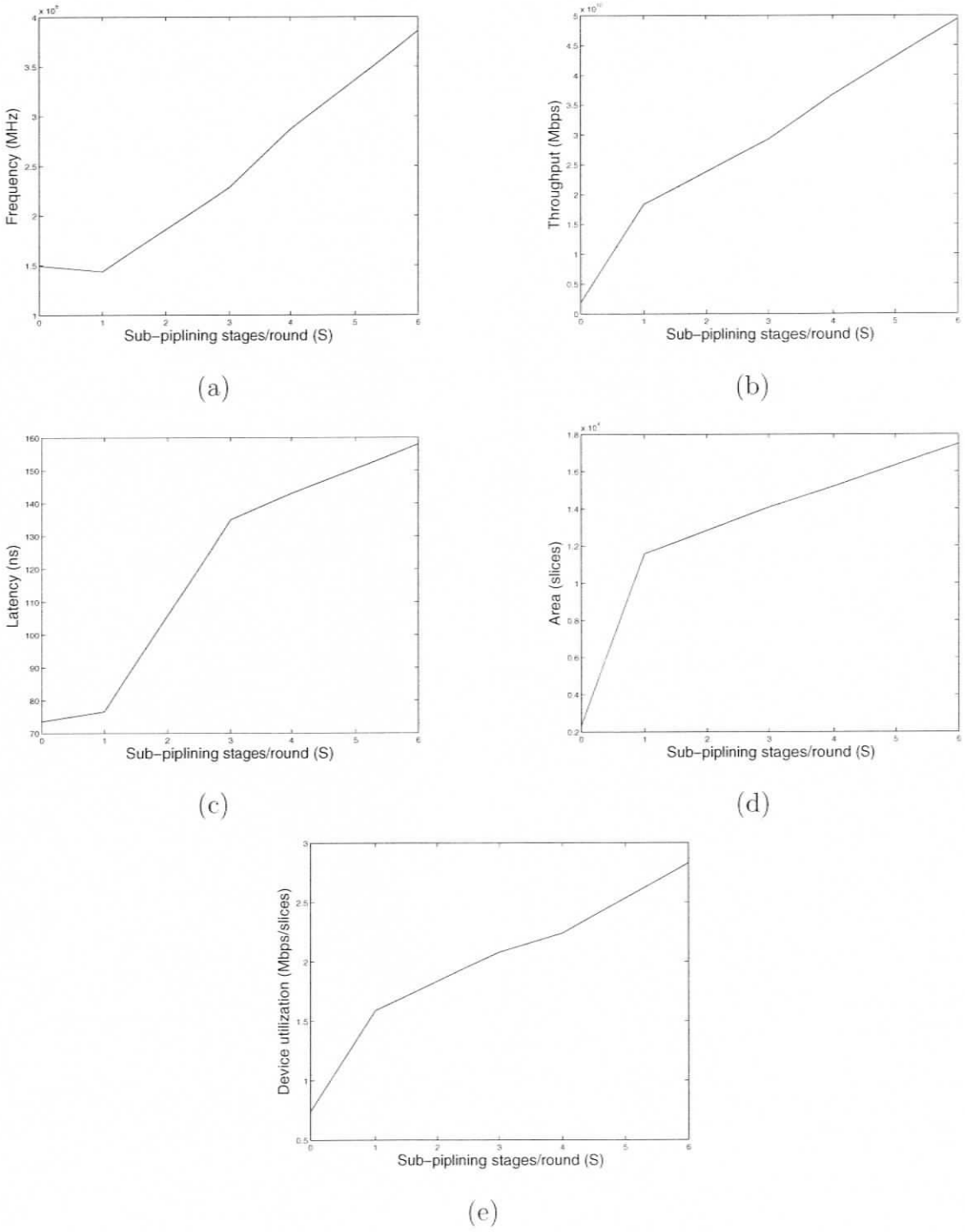


Figure 4.9: Effect of increasing deep-pipelining stages on: (a) frequency, (b) throughput, (c) latency, (d) area, (e) device utilization.

according to [112]

$$Latency = \frac{Cycles\ per\ 128\ bit\ Block}{Frequency} \tag{4.23}$$

Table 4.6 compares the FPGA implementation results of our design to other published designs. Because we optimized our design for speed, the 6-stage deep-pipelined architecture can achieve 49.401 *Gbps* on a Xilinx XC2V6000FF1152-6 device. Although our implementation needs more area, it has better area-speed utilization in which throughput per slice is maximized. It is clear from Table 4.6 that our architecture is faster than all prior FPGA implementations. Our architecture is not only faster but also it is twice the speed of the fastest prior FPGA implementations, which in turn makes our throughput twice the throughput of the fastest prior FPGA implementations.

The frequency and throughput could be increased by increasing the number of the deep-pipelined stages with equal delay as shown in Figure 4.9-a and Figure 4.9-b respectively. But this in turn increases the latency for each data block as shown in Figure 4.9-c. Also, increasing the number of the deep-pipelined stages will increase the area required for implementation as shown in Figure 4.9-d. Moreover, increasing the number of the deep-pipelined stages will increase the device utilization as shown in Figure 4.9-d.

4.5 Chapter Summary

In this chapter, an efficient deep-pipelined architecture of the AES algorithm are presented. In order to explore the advantages of deep-pipelining further, the *SubBytes /InvSubBytes* transformation is implemented by combinational logic to avoid the unbreakable delay of LUTs in the traditional designs. The combinational logic implementation of *SubBytes /InvSubBytes* utilizes the composite field approach.

Table 4.6: Comparison between our implementations and previous implementations.

Design	Device Used	Pipelined Stages	Frequency (MHz)	Throughput (Mbps)	Latency (ns)	Area (slices)	Device Utilization (Mbps/slices)
M. McLoone et al. [44]	XCV812E-8	10	93.9	12020	106	2000	6.01
K. Järvinen et al. [40]	XC2V2000-5	11	139.1	17800	318	10750	1.656
G. Saggese et al. [42]	XCE2000-8	10	70	8900	157	2778	3.2
G. Saggese et al. [42]	XCE2000-8	50	158	20300	323	5810	3.5
F. Standaert et al. [43]	XCV3200e-8	70	145	18560	496	15112	1.228
A. Hodjat et al. [41]	XC2VP20-7	40	168.3	21540	244	12450	1.73
A. Hodjat et al. [41]	XC2VP20-7	70	169.1	21640	420	9446	2.29
X. Zhang et al. [34]	XCV812e-8	30	93.5	11965	332	9406	1.272
X. Zhang et al. [34]	XCV1000e-8	70	168.4	21556	422	11022	1.956
Our design	XC2V6000-6	0	149.424	1738	73.6	2341	0.74
	XC2V6000-6	10	143.649	18387	76.6	11586	1.59
	XC2V6000-6	30	228.310	29223	135	14078	2.08
	XC2V6000-6	40	286.903	36723	143	15204	2.42
	XC2V6000-6	60	385.951	49401	158	17479	2.83

In order to speed up our architecture, we calculated the optimum *isomorphic* mapping parameters by first generating all possible irreducible polynomials of degree 2 whose coefficients are in $GF(2^4)$. Then, we proposed an algorithm to calculate the optimum *isomorphic* mapping parameters given all possible irreducible polynomials. Finally, we used our modified algorithm to find the optimum *isomorphic* mapping parameters. As a result of our novel approach in generating the optimum *isomorphic* mapping parameters, the complexity of the design is reduced and the delay is dramatically decreased. We also explored different approaches for the implementation of inversion in *subfield* $GF(2^4)$ to reduce the complexity and delay.

As an evaluation of our proposed architecture, different fully deep-pipelined implementations using 128-bit key are modeled using VHDL and implemented on an FPGA device. As a result of this approach in utilizing the composite fields for AES implementation, we achieved a throughput of 49.401 Gbps. The achieved throughput is twice the throughput of the fastest Advanced Encryption Standard FPGA implementation known to date.

VoIP processor requires field ALU that performs basic fiels arithmetics. In the next chapter, we propose a high speed, low area ALU to perform field operations required for cryptographic applications.

Chapter 5

A Unified, Memoryless Processor Array Architecture for Basic Field Operations over $GF(2^m)$

This chapter presents the third contribution of this dissertation. We propose a high speed, small area ALU to perform field operations required for cryptographic applications. Although the proposed architecture works for any cryptographic application, we target an ECC implementation for VoIP applications.

The main contributions in this chapter are summarized as follows:

- A processor array design space exploration for $GF(2^m)$ multiplier. This exploration results in different processor array configurations. Among these configurations, we choose the fastest one since we are targeting VoIP applications.

- A modification to the proposed multiplier architecture to work as a squarer. As a result, the squaring operation requires very few number of clock cycles in comparison to multiplication. As an example, squaring operation requires 6%

of the multiplication time for $GF(2^{163})$.

- A unified architecture to calculate addition, multiplication, squaring, and inversion based on the proposed multiplier architecture.
- An optimization to the overall area by using three types of processing elements (PEs) instead of using regular PEs everywhere. There is a PE for the right-most bit, a PE corresponding to 1 in the reduction polynomial $R(x)$, and a PE corresponding to 0 in the reduction polynomial $R(x)$. As a result, this optimization saves some area in comparison to using the same PE everywhere else. As an example, this optimization saves 160 XOR gates and 160 AND gates for $GF(2^{163})$ using NIST reduction polynomials.
- NIST-recommended irreducible polynomials are used, which makes the architecture more secure and more suitable for cryptographic applications.

This chapter is organized first by the introduction, found in Section 5.1. In Section 5.2, we explore previous efforts in implementing field mathematics. In Section 5.3, we briefly describe the background of $GF(2^m)$ arithmetic operations. In Section 5.4, we introduce a processor array design space exploration for $GF(2^m)$ multipliers. In Section 5.5, we introduce a processor array for $GF(2^m)$ squarer based on the multiplier architecture proposed earlier. In Section 5.6, we introduce a processor array for $GF(2^m)$ inverter based on the multiplier and squarer architectures proposed earlier. In Section 5.7, we introduce a processor array unified architecture for $GF(2^m)$ addition, squaring, multiplication, inversion based on the multiplier and squarer architectures proposed earlier. In Section 5.8, comparison with prior processor array architectures are introduced. In Section 5.9, implementation results and comparisons with prior implementations are introduced. Section 5.10 summarizes the chapter.

5.1 Introduction

Finite or Galois fields have become increasingly important in cryptography. A number of cryptographic algorithms (e.g. the Advanced Encryption Standard (AES), Elliptic Curve Cryptography (ECC), RSA, Deffie-Hellman, etc.) rely heavily on properties of finite fields [30, 32, 113]. All these algorithms require fast, inexpensive, and secure implementation to perform the required arithmetic operations, such as addition, multiplication, division, and squaring over $GF(2^m)$. Therefore, the design of efficient high-speed algorithms and hardware architectures for computing arithmetic operations are highly required and considered.

Among the cryptographic algorithms, ECC has been an active area of research because it provides a higher level of security with smaller key lengths than RSA [28–30]. This in turn reduces the cost of hardware and software implementations. However, software implementations of ECC are comparatively slower. Therefore, it is desired to design an efficient architecture in hardware for cryptographic algorithms [29]. Moreover, there is a high demand to implement the required operations (addition, multiplication, division, and squaring) on a single architecture in order to save area and power.

ECC requires arithmetic operations such as: addition, multiplication, squaring, and inversion. All operations in ECC are done over either the prime field $GF(p)$ or the binary field $GF(2^m)$, where m is an integer and $m \in \{163, 233, 283, 409, 571\}$, as recommended by NIST (National Institute of Standard and Technology). NIST recommends ten finite fields, five of which are binary fields, for use in the Elliptic Curve Digital Signature Algorithm (ECDSA) [114]. Table 5.1 lists the irreducible polynomials used to generate binary fields recommended by NIST. The binary field is the most suitable for hardware implementation because elements in $GF(2^m)$ are represented as binary numbers and all the operations are performed using standard

logic gates in the binary domain [28, 113].

Table 5.1: NIST binary field recommendation.

Field	Irreducible
$GF(2^{163})$	$R(x) = x^{163} + x^7 + x^6 + x^3 + 1$
$GF(2^{233})$	$R(x) = x^{233} + x^{74} + 1$
$GF(2^{283})$	$R(x) = x^{283} + x^{12} + x^7 + x^5 + 1$
$GF(2^{409})$	$R(x) = x^{409} + x^{87} + 1$
$GF(2^{571})$	$R(x) = x^{571} + x^{10} + x^5 + x^2 + 1$

Arithmetic operations (e.g. addition, multiplication, squaring, and inversion) over $GF(2^m)$ are very important in many cryptographic areas. The basic operations in field arithmetic are multiplication and addition, since squaring could be done by utilizing a multiplier and an adder [28], whereas inversion could be implemented by repeated squaring and multiplication [28, 115]. Addition in $GF(2^m)$ is simpler than it is in normal binary addition, as the sum can be obtained by a bit-wise XOR operation of the two operands. Furthermore, the latency is independent of operand sizes since there is no carry chain [28]. However, multiplication is more complex because it involves calculating the modulus with respect to an irreducible polynomial $R(x)$ [28, 30, 113]. Since operations such as squaring and division could be implemented using multiplication, an efficient design of a finite field multiplier is needed.

5.2 Related Work

Hardware implementation techniques for $GF(2^m)$ multiplier include traditional techniques [116–120] and processor array techniques [115, 121–132]. Traditional multipliers are not attractive, since their hardware structures are irregular and may

be quite different for different m. Moreover, as m gets larger, the propagation delay increases, which causes unavoidable performance deterioration. On the contrary, processor multipliers do not suffer from above problems. They have regular structures consisting of a number of replicated basic cells. Furthermore, since each basic cell is only connected to its neighboring cells, signals propagate at a high clock speed [133]. In 1984, Yeh et al. [134] proposed a parallel-in parallel-out processor architecture to calculate $AB + C$ in a general field $GF(2^m)$. Since then, many processor multipliers have been proposed [115, 121–132].

There exist several methods to calculate the $GF(2^m)$ inverse. These methods include: composite field-based implementations, Euclid's algorithm-based implementations, and Fermat's theorem-based implementations [96]. Composite field-based implementations are presented in [135–137]. Examples of Euclid's algorithm-based implementations are presented in [138–145]. Some of these implementations are processor array-based implementations [138–142], whereas the others are traditional implementations [143–145]. Inversion architectures based on Euclid's algorithm require a polynomial division and a multiplication for each iteration, which demands a high computational power. This in turn increases the implementation area as well as the critical path delay [28]. Moreover, processor array architectures based on the Euclid's algorithm have complex Processing Elements (PEs), which increases the area and critical path delay [138–142]. Inversion architectures based on Fermat's theorem require recursive squaring and multiplication over finite fields [101, 102]. Inverters based on this technique require $m-1$ multiplications and $m-2$ squaring to calculate inversion [28].

There are few efforts in the literature to implement more than one $GF(2^m)$ arithmetic operations on a single architecture. Leong et al. [146] proposed a finite field ALU based on a non-processor parallel finite field multiplier. The parallel multiplier requires large area and longer critical path delay for large m.

Cheung et al. [147] proposed a non-processor architecture to calculate finite field operations (multiplication, inversion, squaring, and adding) required for ECC scalar multiplication. Their proposed architecture has different circuits for each field operation, which requires a large area to implement. Yeh et al. [134] proposed an architecture to calculate $AB + C$. Wei [148] proposed an architecture to calculate $AB^2 + C$ for $GF(2^m)$. Jain et al. [149] proposed an architecture to calculate $AB^n + C$ for $GF(2^m)$. Wang et al. [150] proposed an architecture to calculate $C + AB^2$ for $GF(2^m)$. Kim et al. [151] proposed a processor array architecture for multiplication and squaring. All the above mentioned architectures ([134, 148–151]) have complex PEs, which results in large area and long critical path delay. As a result, they are not suitable for real-time cryptographic applications such as VoIP.

Lee et al. [152] proposed an architecture to calculate $AB^2 + C$ for $GF(2^m)$. Although their PE is simpler than previous ones, it still needs a large area for large m since it requires $(m + 1)^2$ PEs. Moreover, the design uses All-One-Polynomial (AOP), which is not secure for cryptographic applications [114].

Lee et al. [153] proposed an architecture to calculate AB^2 for $GF(2^m)$. The proposed PE is complex, which requires large area and long critical path delay. Moreover, the design uses a polynomial not recommended by NIST, which is not secure for cryptographic applications.

All previous processor array architectures require large number of cells (PEs) (e.g. the architecture in [150] requires $m^4/2$ PEs), which makes them not suitable for large-m implementations.

5.3 Background on Arithmetic over Binary Fields $GF(2^m)$

The elements of the binary field $GF(2^m)$ are interrelated through addition, multiplication, squaring, and inversion operations [154]. This section introduces the basic methods for computing addition, subtraction, and multiplication of two elements. Moreover, the basic methods for computing squaring and inverse of an element are introduced. The squaring is a special case of multiplication. The inverse of an element, along with a multiplication, is used to implement division [154, 155].

Let $A(x)$ and $B(x)$ be two elements in $GF(2^m)$, $R(x)$ being the primitive polynomial used to generate the field elements, then

$$A(x) = \sum_{i=0}^{m-1} a_i x^i \tag{5.1}$$

$$B(x) = \sum_{i=0}^{m-1} b_i x^i \tag{5.2}$$

$$R(x) = x^m + \sum_{i=0}^{m-1} r_i x^i = x^m + f(x) \tag{5.3}$$

5.3.1 Addition and Subtraction

The summation of field elements A and B is written as

$$S(x) = A(x) + B(x) = \sum_{i=0}^{m-1} (a_i + b_i) x^i \tag{5.4}$$

Addition of field elements $(a_i + b_i)$ is performed modulo 2, which is accomplished by bitwise XOR operations. As a result, addition does not require a carry chain. Since element 1 is its own additive inverse (i.e. $1 + 1 = 1 - 1 = 0$ *or* $1 = -1$), subtraction of field elements is exactly the same as the addition of field elements [28, 154, 155].

5.3.2 Field Multiplication

The product of field elements A and B is written as

$$\begin{aligned} C(x) &= A(x).B(x) \bmod R(x) \\ &= \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j x^{i+j} \bmod R(x) \end{aligned} \tag{5.5}$$

Algorithm 4 illustrates the right-to-left shift-and-add method for field multiplication. This method is based on the observation that

$$C(x) = b_{m-1}x^{m-1}A(x) + \ldots + b_1 xA(x) + b_0 A(x) \bmod R(x) \tag{5.6}$$

Iteration i in Algorithm 4 computes $x^i A(x) \bmod R(x)$ and adds the result to accumulator $C(x)$ if $b_i = 1$ [28].

Algorithm 4 Right-to-Left shift-and-add field multiplication.

Input: Binary polynomials $A(x)$ and $B(x)$ of degree at most $m-1$ and $R(x)$ of degree m

Output: $C(x) = A(x).B(x) \bmod R(x)$

1: $C(x) = b_0.A(x)$
2: **for** $i = 1$ to $m-1$ **do**
3: $\quad A(x) = x.A(x) \bmod R(x)$
4: $\quad C(x) = C(x) + b_i.A(x)$
5: **end for**
6: **return** $C(x)$

Algorithm 5 illustrates the left-to-right shift-and-add method for field multiplication [154]. This method is based on the observation that expanding $B(x)$ and distributing $A(x)$ through its terms give

$$\begin{aligned} C(x) &= (b_{m-1}x^{m-1}A(x) + \ldots + b_1 xA(x) + b_0 A(x)) \bmod R(x) \\ &= ((b_{m-1}A(x))x + \ldots + b_1 A(x))x + b_0 A(x) \bmod R(x) \end{aligned} \tag{5.7}$$

The expressions in Step 3 in Algorithms 4 and 5 could be calculated as follows: If

Algorithm 5 Left-to-Right shift-and-add field multiplication.

Input: Binary polynomials $A(x)$ and $B(x)$ of degree at most $m-1$ and $R(x)$ of degree m

Output: $C(x) = A(x).B(x) \bmod R(x)$

1: $C(x) = 0$
2: **for** $i = m-1$ downto 0 **do**
3: $\quad C(x) = x.C(x) \bmod R(x)$
4: $\quad C(x) = C(x) + b_i.A(x)$
5: **end for**
6: **return** $C(x)$

$C(x) = \sum_{i=0}^{m-1} c_i x^i$, then

$$\begin{aligned} x.C(x) \bmod R(x) &= c_{m-1}x^m + c_{m-2}x^{m-1} + \ldots + c_0 x \bmod R(x) \\ &= c_{m-1}f(x) + (c_{m-2}x^{m-1} + \ldots + c_0 x) \end{aligned} \tag{5.8}$$

Thus, $x.C(x) \bmod R(x)$ can be evaluated by a left-shift of $C(x)$ and adding $f(x)$ to $C(x)$ if $c_{m-1} = 1$.

5.3.3 Field Squaring

Binary field squaring is a linear operation. Therefore, it is much faster than field multiplication [28]. The squaring of field element A is written as

$$A^2(x) = \sum_{i=0}^{m-1} a_i x^{2i} \bmod R(x) \tag{5.9}$$

The binary representation of $A^2(x)$ is obtained by inserting an 0 bit between consecutive bits of the binary representation of $A(x)$, followed by a reduction process [28].

5.3.4 Field Inversion

For any element $A \in GF(2^m)$, the equality $A^{2^m-1} = 1$ holds [156]. When $A \neq 0$ and A^{-1} exists, dividing both sides by A results in $A^{2^m-2} = A^{-1}$. Using this equality the

inverse, A^{-1}, can be computed through successive field squarings and multiplications. Algorithm 6 shows how the inverse of an element is computed using this method. The primary advantage of this inversion method is that it does not require hardware dedicated specifically for inversion. The field multiplier can be used to perform all required field operations.

Algorithm 6 Field inversion using square and multiply.

Input: Field element $A(x) = (a_{m-1}x^{m-1} + \ldots + a_1x + a_0)$

Output: $B = A^{-1}$

1: $B = A$
2: **for** $i = 1$ to $m - 2$ **do**
3: $\quad B = B^2.A \bmod R(x)$
4: **end for**
5: $B = B^2 \bmod R(x)$
6: **return** B

5.4 Proposed Processor Array Architecture for a $GF(2^m)$ Multiplier

Multiplication over $GF(2^m)$ is very important, since all other operations (e.g. squaring, division, etc.) could be implemented using it. In this section, we explore processor array design options of the $GF(2^m)$ multiplier.

5.4.1 Expressing Multiplication as an Iterative Algorithm

To develop a processor array, first we must describe the multiplication algorithm using recursions that convert an algorithm into a regular iterative algorithm (RIA). We can rewrite the left-to-right multiplication algorithm (Algorithm 5) as in Algorithm 7. This algorithm can also be expressed in the form of an iteration using two indices

i and j, where i represents iteration index and j represents bit index. The values of i and j are $0 < i \leq m$, and $0 \leq j < m$, respectively. Initially, $c(0, j) = 0$ for $0 \leq j \leq m-1$, then we can write

$$c(i,j) = \begin{cases} (a(0) \wedge b(m-i)) \oplus c(i-1, m-1) & j = 0 \\ (a(j) \wedge b(m-i)) \oplus c(i-1, j-1) & \\ \oplus (c(i-1, m-1) \wedge r(j)) & j > 0 \end{cases} \tag{5.10}$$

where $\oplus$ represents logical XOR operation, and $\wedge$ represents logical AND operation. The final result, obtained at iteration $(i = m)$, is given by

$$C = (c(m, m-1), \ldots, c(m,1), c(m,0)) \tag{5.11}$$

Algorithm 7 RIA for Left-to-Right field multiplication.

Input: Binary polynomials $A(x)$ and $B(x)$ of degree at most $m-1$ and $R(x)$ of degree m

Output: $C(x) = A(x).B(x) \bmod R(x)$

1: **for** $j = 0$ to $m-1$ **do**
2: $\quad c(0, j) = 0$
3: **end for**
4: **for** $i = 1$ to m **do**
5: $\quad c(i, 0) = [a(0) \wedge b(m-i)] \oplus c(i-1, m-1)$
6: $\quad$ **for** $j = 1$ to $m-1$ **do**
7: $\quad\quad c(i, j) = [a(j) \wedge b(m-i)] \oplus c(i-1, j-1) \oplus [c(i-1, m-1) \wedge r(j)]$
8: $\quad$ **end for**
9: **end for**
10: **return** $C(x)$

5.4.2 Obtaining the Algorithm Dependency Graph (DG)

The field multiplication algorithm of Equation (5.10) is defined on a two dimensional (2D) domain since there are two indices (i, j). The data dependency graph is shown

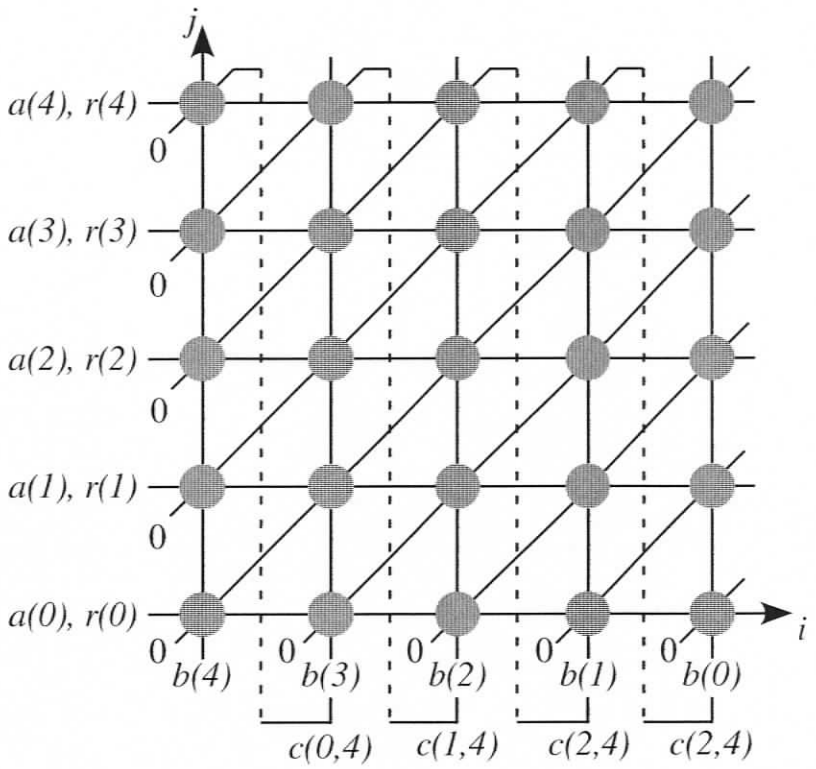


Figure 5.1: Field multiplication dependency graph for $m = 5$.

in Figure 5.1. The *computation domain* is the convex hull in the 2D space, where the algorithm operations are defined as indicated by the grayed circles in the 2D plane [157, 158].

From Equation (5.10), we notice that input variables $a(j)$ and $r(j)$ are represented by horizontal lines, input variables $b(m-1-i)$ and $c(i-1, m-1)$ are represented by the vertical lines, and the output variables $c(i-1, j-1)$ are represented by the diagonal lines.

5.4.3 Data Scheduling

Pipelining or broadcasting the variables of an algorithm is determined by the choice of a timing function that assigns a time value to each node in the DG. A simple but useful timing function is an affine scheduling function of the form [157]

$$t(\mathbf{p}) = \mathbf{s}\mathbf{p} - s \tag{5.12}$$

where the function $t(\mathbf{p})$ associates a time value t to a point $\mathbf{p}$ in the DG. Since all points in the DG must have non negative time index values, we must have $s = 0$. The row vector $\mathbf{s} = [s_1 \ s_2]$ is the scheduling vector and s is an integer. Such affine scheduling function must satisfy several conditions. Input data timing restricts the space of valid scheduling functions. We note from Equation (5.10) that, if $b(m-1-i)$ arrives at iteration i corresponding to time t then, $b(m-1-1)$ arrives at at iteration $i+1$ corresponding to time $t+1$. The bits, $b(m-1-i)$ and $b(m-i-1)$ arrive at points $\mathbf{p}_1 = (i, 0)$ and $\mathbf{p}_2 = (i+1, 0)$ at the times $t(\mathbf{p}_1)$ and $t(\mathbf{p}_2)$, respectively. By applying our scheduling function in Equation (5.12) to these two points, we get

$$\begin{aligned} t(\mathbf{p}_1) &= \begin{bmatrix} s_1 & s_2 \end{bmatrix} \begin{bmatrix} i \\ 0 \end{bmatrix} = is_1 - s \\ t(\mathbf{p}_2) &= \begin{bmatrix} s_1 & s_2 \end{bmatrix} \begin{bmatrix} i+1 \\ 0 \end{bmatrix} = (i+1)s_1 - s \end{aligned} \tag{5.13}$$

Since the time difference $t(\mathbf{p}_2) - t(\mathbf{p}_1) = 1$, we must have $s_1 = 1$. Therefore, a scheduling vector that satisfies input data timing must be specified as

$$\mathbf{s} = \begin{bmatrix} 1 & s_2 \end{bmatrix} \tag{5.14}$$

This leaves two unknowns in the possible timing functions, mainly the component s_1 and the integer s.

From Equation (5.10), we observe that the output $c(i,j)$ depends on the previous output value $c(i-1,j-1)$, thus we can write

$$\begin{array}{lcl} t(\mathbf{p}(i-1,j-1)) & < & t(\mathbf{p}(i,j)) \\ \begin{bmatrix} 1 & s_2 \end{bmatrix} \begin{bmatrix} i-1 \\ j-1 \end{bmatrix} & < & \begin{bmatrix} 1 & s_2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \\ i-1+(j-1)s_2 & < & i+js_2 \end{array} \tag{5.15}$$

which could be simplified into

$$s_2 > -1 \tag{5.16}$$

Similarly from Equation (5.10), we observe that the output $c(i,j)$ depends on the previous output value $c(i-1,m-1)$, thus we can write

$$\begin{array}{lcl} t(\mathbf{p}(i-1,m-1)) & < & t(\mathbf{p}(i,j)) \\ \begin{bmatrix} 1 & s_2 \end{bmatrix} \begin{bmatrix} i-1 \\ m-1 \end{bmatrix} & < & \begin{bmatrix} 1 & s_2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \\ i-1+(m-1)s_2 & < & i+js_2 \end{array} \tag{5.17}$$

which could be simplified into

$$(m-1-j)s_2 < 1 \tag{5.18}$$

Hence, Equations (5.16) and (5.18) could be merged as

$$-1 < s_2 < \frac{1}{(m-1-j)} \qquad 0 \le j < m \tag{5.19}$$

The worst case in the above inequality is when $j = 0$, which can be written as

$$-1 < s_2 < \frac{1}{(m-1)} \tag{5.20}$$

The above inequality can be rewritten as

$$-\frac{\alpha}{\beta} \le s_2 \le \frac{1}{m} \tag{5.21}$$

where α and β are positive integers, and $\alpha < \beta$.

From Equation (5.21), there are three possible reasonable solutions to $\mathbf{s}$

$$\mathbf{s}_1 = [1\ 0] \tag{5.22}$$

$$\mathbf{s}_2 = [1\ \frac{1}{m}] \tag{5.23}$$

$$\mathbf{s}_3 = [1\ -\frac{\alpha}{\beta}] \tag{5.24}$$

The timing functions $\mathbf{s}_2$ and $\mathbf{s}_3$ imply that only one node is active at a given time step. To prove this, suppose that we have two points (i, j) and $(\acute{i}, \acute{j})$ mapped to the same time value. Thus, by using Equation (5.23) we can write

$$i + \frac{j}{m} = \acute{i} + \frac{\acute{j}}{m} \qquad 0 \leq j < m \tag{5.25}$$

The only solution to the above equation is when $i = \acute{i}$ and $j = \acute{j}$, which proves that only one node is active at the same time step. A similar argument is valid also for $\mathbf{s}_3$.

Since timing functions $\mathbf{s}_2$ and $\mathbf{s}_3$ imply that only a single node is active at a given time step, the algorithm could be implemented using a single processor. Therefore, it is not recommended to explore the processor array design for these two timing functions since there is no processor array. Hence, we consider the timing function $\mathbf{s}_1$ only for exploration.

If we decide to pipeline a variable whose null-vector is $\mathbf{e}$, we must have [157]

$$\mathbf{se} \neq 0 \tag{5.26}$$

On the other hand, to broadcast a variable whose null-vector is $\mathbf{e}$, we must have [157]

$$\mathbf{se} = 0 \tag{5.27}$$

To study the timing of the input variables $a(j)$, $r(j)$, $b(m-1-i)$ $c(i-i, m-1)$, and

$c(i-1, j-1)$, we first find their null-vectors

$$\mathbf{e}_{a(j)} = [1\ 0] \tag{5.28}$$

$$\mathbf{e}_{r(j)} = [1\ 0] \tag{5.29}$$

$$\mathbf{e}_{b(m-i)} = [0\ 1] \tag{5.30}$$

$$\mathbf{e}_{c(i-1,j-1)} = [1\ 1] \tag{5.31}$$

$$\mathbf{e}_{c(i-1,m-1)} = [0\ 1] \tag{5.32}$$

The product of $\mathbf{s}_1$ and these null-vectors gives

$$\mathbf{s}_1\mathbf{e}_{a(j)} = 1 \tag{5.33}$$

$$\mathbf{s}_1\mathbf{e}_{r(j)} = 1 \tag{5.34}$$

$$\mathbf{s}_1\mathbf{e}_{b(m-i)} = 0 \tag{5.35}$$

$$\mathbf{s}_1\mathbf{e}_{c(i-1,j-1)} = 1 \tag{5.36}$$

$$\mathbf{s}_1\mathbf{e}_{a(i-1,m-1)} = 0 \tag{5.37}$$

Therefore, inputs $a(j)$ and $r(j)$ are pipelined, whereas input $b(m-1-i)$ and $c(i-i, m-1)$ are broadcast. The intermediate output of each node $c(i-1, j-1)$ is pipelined to the next node in the next iteration $c(i, j)$.

5.4.4 DG Node Projection

The projection operation is a many-to-one function that maps several nodes of the DG onto a single node, which constitutes the resulting processor array. Thus, several operations in the DG are mapped to a single PE. The projection operation allows hardware economy by multiplexing several operations in the DG on a single PE. ElGuibaly et al. [157] explained how to perform the projection operation using a *projection matrix* $\mathbf{P}$. To obtain the projection matrix we need to define a desired

projection direction $\mathbf{d}$. The vector $\mathbf{d}$ belongs to the null space of $\mathbf{P}$. Since we are dealing with a two-dimensional DG, matrix $\mathbf{P}$ is a row vector and $\mathbf{d}$ is a column vector [157]. A valid projection direction $\mathbf{d}$ must satisfy the inequality [157]

$$\mathbf{sd} \neq 0 \tag{5.38}$$

In the following, we discuss design space explorations based on the timing function $\mathbf{s}_1$ obtained in Equation (5.22). There are many projection vectors that satisfy Equation (5.38) for the scheduling function in Equation (5.22). For simplicity we choose three of them as follows:

$$\mathbf{d}_1 = [1\ 0]^t \tag{5.39}$$

$$\mathbf{d}_2 = [1\ 1]^t \tag{5.40}$$

$$\mathbf{d}_3 = [1\ -1]^t \tag{5.41}$$

The corresponding projection matrices could be given by

$$\mathbf{P}_1 = [0\ 1] \tag{5.42}$$

$$\mathbf{P}_2 = [1\ -1] \tag{5.43}$$

$$\mathbf{P}_3 = [1\ 1] \tag{5.44}$$

Our processor design space now allows for three processor array configurations for each projection vector for the timing function $\mathbf{s}_1$.

Design 1: Using $\mathbf{d}_1 = [1\ 0]^t$:

A point in the DG $\mathbf{p} = [i\ j]^t$ will be mapped by the projection matrix $\mathbf{P}_1 = [0\ 1]$ onto the point

$$\acute{\mathbf{p}} = \mathbf{P}_1\mathbf{p} = j \tag{5.45}$$

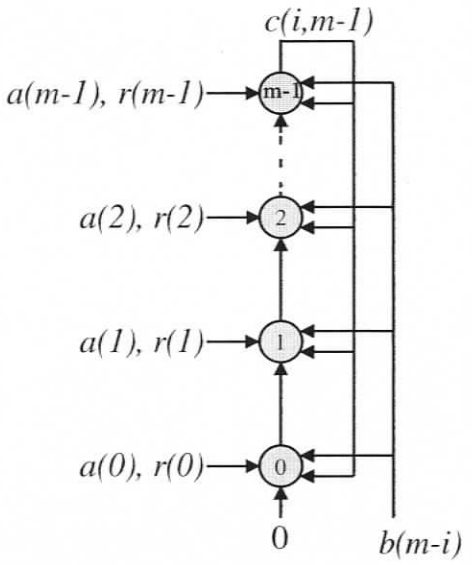


Figure 5.2: Processor array when $\mathbf{d}_1 = [1\ 0]^t$.

The processor array corresponding to the projection matrix $\mathbf{P}_1$ is shown in Figure 5.2. The processor array consists of m PEs, each is active for m time steps. The calculation latency is m clock cycles.

Design 2: Using $\mathbf{d}_2 = [1\ 1]^t$:

A point in the DG $\mathbf{p} = [i\ j]^t$ will be mapped by the projection matrix $\mathbf{P}_2 = [1\ -1]$ onto the point

$$\acute{\mathbf{p}} = \mathbf{P}_2\mathbf{p} = i - j \tag{5.46}$$

The resulting processor array consists of $2m-1$ PEs. after adding a fixed increment to all PE indices to ensure non-negative PE index values [157]. Although the processor array consists of $2m-1$ PEs. only m PEs are active at each time step according to the scheduling vector $\mathbf{s} = [1\ \ 0]$. Figure 5.3 shows the processor array activity for the case $m = 5$. where grayed nodes represent active PEs and white nodes represent

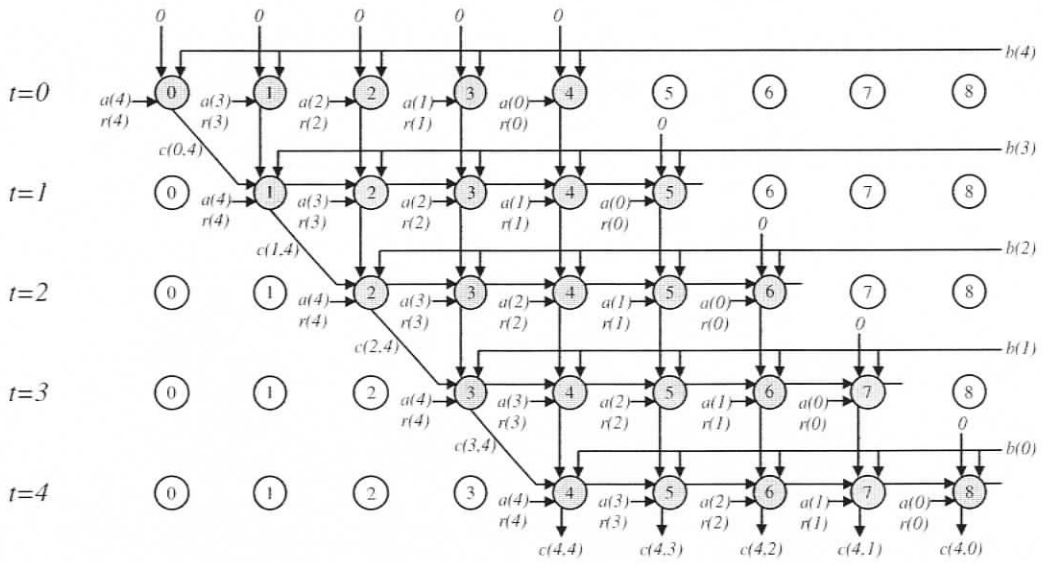


Figure 5.3: Active PEs when $\mathbf{d}_2 = [1\ 1]^t$ and $m = 5$.

idle PEs. Since only m PEs are active at a given time step, the PEs are not well utilized. We note from Figure 5.3 that PE_i and PE_{m+i} are active at non-overlapping time steps. Thus, each pair of PEs (PE_i, PE_{m+i}) could be mapped to a single PE without causing any timing conflicts. In fact, all PEs whose index is expressed as

$$\acute{i} = i \bmod m \tag{5.47}$$

can be mapped to the same processor without any timing conflicts. The resulting processor array after applying the above *modulo* operations on the array in Figure 5.3 is shown in Figure 5.4. The processor array now consists of m PEs, each is active for m time steps. The calculation latency is m clock cycles.

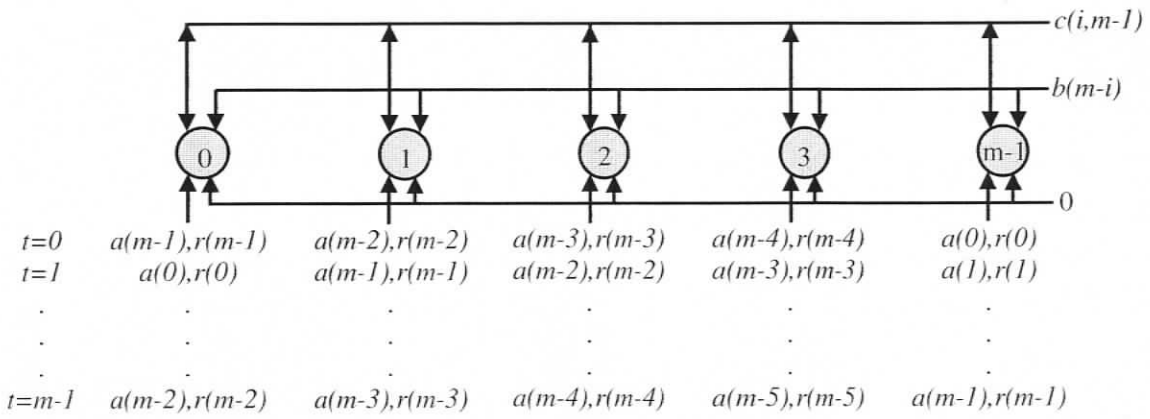


Figure 5.4: Processor array when $\mathbf{d}_2 = [1\ 1]^t$.

Design 3: Using $\mathbf{d}_3 = [1\ -1]^t$:

A point in the DG $\mathbf{p} = [i\ j]^t$ will be mapped by the projection matrix $\mathbf{P}_3 = [1\ 1]$ onto the point

$$\acute{\mathbf{p}} = \mathbf{P}_3\mathbf{p} = i + j \tag{5.48}$$

The resulting processor array consists of $2m - 1$ PEs. Although the processor array consists of $2m - 1$ PEs, only m PEs are active at each time step according to the scheduling vector $\mathbf{s} = [1\ 0]$. Figure 5.5 shows the processor array activity for the case $m = 5$, where grayed nodes represent active PEs and white nodes represent idle PEs. Since only m PEs are active at a given time step, the PEs are not well utilized. We note from Figure 5.5 that PE_i and PE_{m+i} are active at non-overlapping time steps. Thus, each pair of PEs (PE_i, PE_{m+i}) could be mapped to a single PE without causing any timing conflicts. In fact, all PEs whose index is expressed as

$$\acute{i} = i \bmod m \tag{5.49}$$

can be mapped to the same processor without any timing conflicts. The resulting processor array after applying the above modulo operations on the array in Figure 5.5

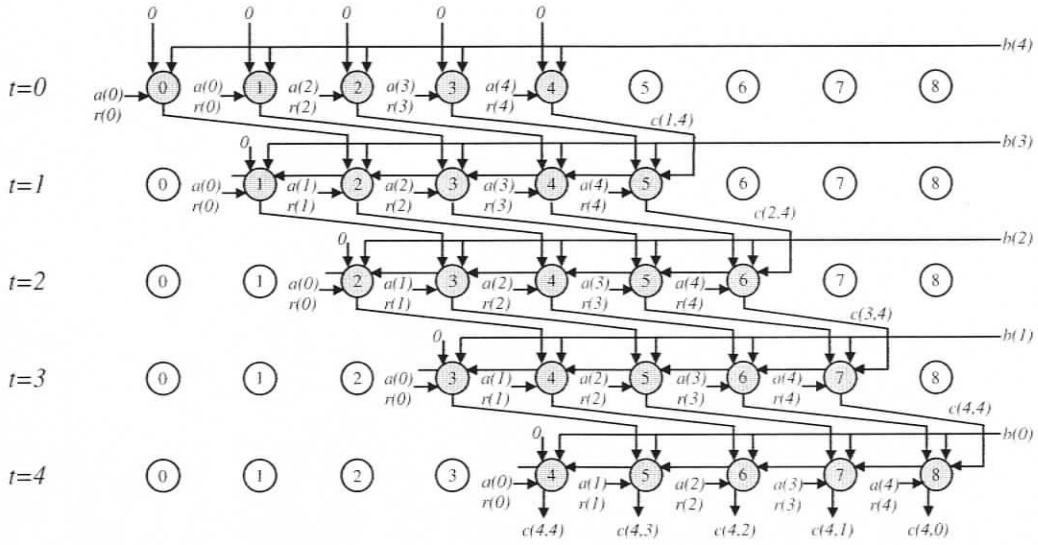


Figure 5.5: Active PEs when $\mathbf{d}_3 = [1 \ -1]^t$ and $m = 5$.

is shown in Figure 5.6. The processor array now consists of m PEs, each is active for m time steps. The calculation latency is m clock cycles.

5.4.5 Processor Array Multiplier Architecture

In Section 5.4.4, we explored the different designs for the multiplier processor array. All the resulting processor array configuration have m processors and requires m clock cycles to calculate the result. However, Design 1 has the simplest signal control. Therefore, we use Design 1 in our implementation.

Figure 5.7 (a) illustrates the design of the PE based on Design 1 discussed earlier. Since NIST reduction polynomials (Table 5.1) have few 1's (e.g. the $GF(2^{163})$ reduction polynomial has only 4 1's out of 163-bit vector), there is no need to use the PE of Figure 5.7 (a) everywhere. Therefore, we designed two optimized types of PEs: PE for $R_i = 0$, and PE for $R_i = 1$. Moreover, we designed a PE for the first bit since

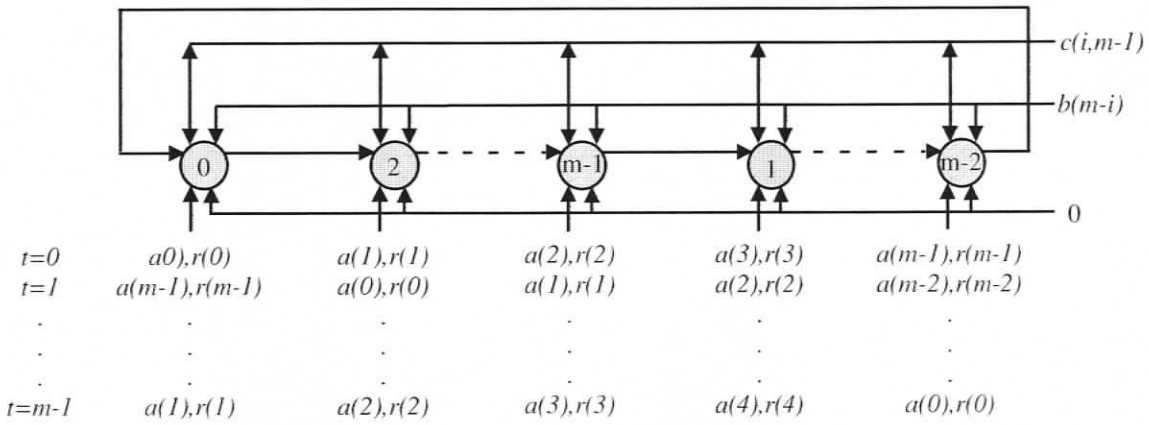


Figure 5.6: Processor array when $\mathbf{d}_3 = [1 \ -1]^t$, m is odd.

the initial value $c(0, j) = 0$ for $0 \leq j \leq m - 1$. As a result of designing three types of PEs, we reduced the processor array area by 160 AND gates and 160 XOR gates for $GF(2^{163})$. Moreover, we saved an m-bit register required to save the reduction polynomial R, and the time required to load this register. Figure 5.7 (b-d) illustrates these three types of PE.

Figure 5.8 illustrates the proposed architecture for the multiplier based on Design 1 discussed earlier. The proposed architecture consists of: register RA, register RB, multiplier processor array, controller, and W-bit data buses. The role of each component is as follows:

- **Register** RA**:** an m-bit parallel-in-parallel-out (PIPO), which holds the multiplicand A.
- **Register** RB**:** an m-bit parallel-in-serial-out, which holds the multiplier B.
- **Multiplier processor array:** consists of m processing elements as shown in Figure 5.2. Each PE type is selected according to its inputs as detailed in Figure 5.7. The multiplier processor array performs the multiplication operation

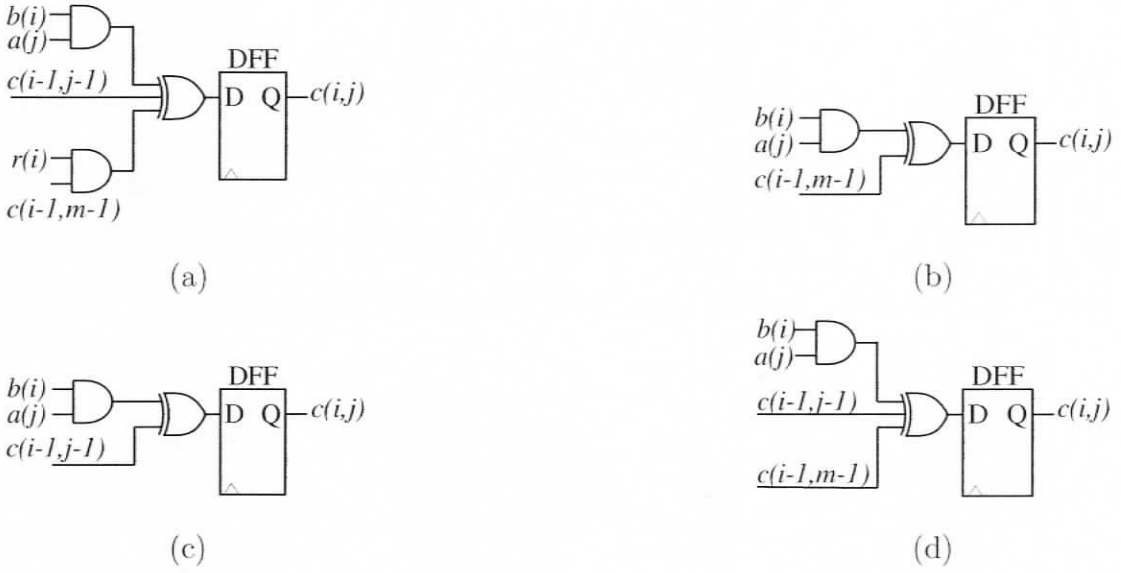


Figure 5.7: Processing Element for: (a) Regular PE (b) First bit PE (c) $R_i = 0$ PE (d) $R_i = 1$ PE.

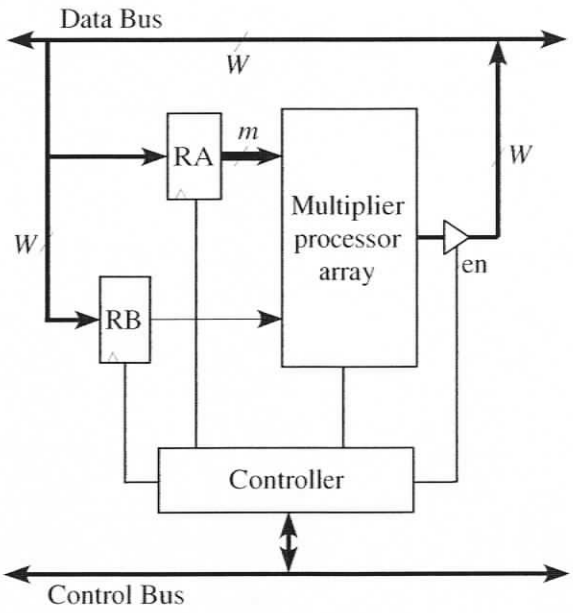


Figure 5.8: Processor array architecture for a $GF(2^m)$ multiplier.

and stores the partial results and the final result C. The reduction polynomial R is embedded (hard coded) inside the processor array in order to save area.

- **Controller:** coordinates data movement between registers and operation of multiplier processor array.

Because the data buses are W bits in width, registers RA and RB require $\lceil m/W \rceil$ clock cycles each to load the input data, while the result is unloaded in $\lceil m/W \rceil$ clock cycles. Therefore, the proposed architecture requires $2\lceil m/W \rceil$ clock cycles to load the input registers, m clock cycles to calculate the results, and $\lceil m/W \rceil$ clock cycles to unload the result. Hence, it requires $3\lceil m/W \rceil + m$ clock cycles in total to calculate $A(x).B(x) \bmod R(x)$ in $GF(2^m)$. If T_m denotes the number of clock cycles required

to perform the multiplication operation, we can write

$$T_m = 3\lceil\frac{m}{W}\rceil + m \tag{5.50}$$

5.5 Proposed Processor Array Architecture for a $GF(2^m)$ Squaring

Squaring is a special case of multiplication and can be performed by the multiplier structure in Section 5.4 after some modifications. Squaring performance can be improved significantly by optimizing the multiplier architecture specifically for the case of squaring. The square of a field element $A(x)$ involves two steps.

The first is the polynomial multiplication of $A(x)$ by itself, resulting in

$$\begin{aligned} T(x) &= A(x).A(x) \\ &= a_{m-1}x^{2m-2} + \ldots + a_1x^2 + a_0 \end{aligned} \tag{5.51}$$

The second is the reduction of this polynomial modulo $R(x)$. If the terms with a degree greater than $m-1$ are separated and x^{m+1} is factored out, the result will be

$$\begin{aligned} A^2(x) &= T(x) && \bmod R(x) \\ &= T_h(x)x^{m+1} + T_l(x) && \bmod R(x) \end{aligned} \tag{5.52}$$

where

$$T_l(x) = \sum_{j=0}^{m-1} t_j x^j \tag{5.53}$$

$$T_h(x) = \sum_{j=m}^{2m-2} t_j x^j \tag{5.54}$$

The polynomial $T_l(x)$ has a degree less than m and does not need to be reduced. The product $T_h(x)x^{m+1}$ may have a degree as large as $2m-2$. The reduction polynomial $R(x)$ gives us the equality

$$x^m = x^d + \ldots + 1 \tag{5.55}$$

where d is the right hand side maximum power of x in the reduction polynomial after rewriting it as in Equation (5.55) (e.g. $d = 7$ for $GF(2^{163})$). See Table 5.1 for the values of d. Multiplying both sides by x, we get

$$x^{m+1} = x^{d+1} + \ldots + x \tag{5.56}$$

Using Equations (5.52) and (5.56) we can write

$$T_h(x)x^{m+1} = T_h(x)(x^{d+1} + \ldots + x) \mod R(x) \tag{5.57}$$

Therefore, Equation (5.51) could be rewritten as

$$A^2(x) = T_l(x) + [T_h(x)(x^{d+1} + \ldots + x) \mod R(x)] \tag{5.58}$$

The operation $T_h(x)(x^{d+1} + \ldots + x) \mod R(x)$ can be performed using the multiplier architecture described in Section 5.4. Algorithm 8 illustrates the above mentioned squaring method.

To develop a processor array for field squaring, first we must describe the squaring operation using recursions that convert an algorithm into RIA. We can rewrite the squaring algorithm as in Algorithm 9.

Since squaring requires multiplication, we can utilize the proposed multiplier in Section 5.4. However, we need to modify this architecture to add $T_l(x)$ to the multiplication result. To modify the multiplier to work as squarer, we add an extra register RX and an adder. The modified architecture that performs multiplication, addition, and squaring is illustrated in Figure 5.9. Our architecture has the adder separated from the multiplier processor array as opposed to previous architectures [148, 150]. Merging the adder to the multiplier processor array to get $AB+C$ directly will increase the delay of each iteration of the multiplication operation by one mux. Therefore, the total multiplier delay will increase by mT_{Mux}. The role of each component is as follows:

Algorithm 8 Field squaring.

Input: Binary polynomials $A(x)$ of degree at most $m-1$, $R(x)$ of degree m, and $B(x) = x^{d+1} + \ldots + x$.

Output: $C(x) = A^2(x) \mod R(x)$

1: **for** $i = 0$ to $m-1$ **do**
2: $\quad T(2*i) = A(i)$
3: $\quad T(2*i+1) = 0$
4: **end for**
5: $T_l = T_{m-1}, \ldots, T_0$
6: $T_h = T_{2m-2}, \ldots, T_m$
7: $C(x) = 0$
8: **for** $i = 1$ to $d+2$ **do**
9: $\quad C(x) = x.C(x) \mod R(x)$
10: $\quad C(x) = C(x) + b_i.T_h(x)$
11: **end for**
12: $C(x) = C(x) + T_l(x)$
13: **return** $C(x)$

- **Register** RA: an m-bit parallel-in-parallel-out, which holds the multiplicand A if the architecture is used as a multiplier or the most significant m bits (T_h) of T if the architecture is used as a squarer. If the architecture is used as a multiplier, register RA loads m bits in $\lceil m/W \rceil$ clock cycles with W bits each clock cycle. If the architecture is used as a squarer, register RA uses W bits to fill $2W$ bits each clock cycle by inserting a 0 bit between each two consecutive bits.

- **Register** RB: an m-bit parallel-in-serial-out with dual usage. If the architecture is used as a multiplier, it holds the multiplier B. Otherwise, it holds the constant $x^{d+1} + \ldots + x$ when the architecture is used as a squarer.

- **Register** RX: an m-bit parallel-in-parallel-out, which holds the least

significant m bits (T_l) of T if the architecture is used as a squarer. Register RX uses W bits to fill $2W$ bits each clock cycle by inserting a 0 bit between each two consecutive bits.

- **Multiplier processor array:** consists of m processing elements. Each PE type is selected according to its inputs as detailed in Figure 5.7.
- **Adder:** consists of m XOR gates, which are isolated from the data bus by using a tri-state buffer.
- **Controller:** receives control signals and controls the data bus as required for each operation.

The modified architecture performs squaring in two steps. First, it calculates $[(T_h(x)(x^{d+1} + \ldots + x)) \bmod R(x)]$ by loading $T_h(x)$ into register RA, loading the constant $x^{d+1} + \ldots + x$ to register RB, and adjusts the number of iterations to be $d+2$. Secondly, it adds the multiplication results to $T_l(x)$. The multiplication process requires $d+2$ clock cycles, since the length of the multiplier is $d+2$ bits only, while addition requires 1 clock cycle. Figure 5.9 illustrates the proposed architecture for the squarer.

Because data buses are W bits in width, registers RA and RX require $\lceil m/(2W) \rceil$ clock cycles each to load the input data, while the result is unloaded in $\lceil m/W \rceil$ clock cycles. Moreover, register RX requires 1 clock cycle to load the constant $x^{d+1}+\ldots+x$ since $d \ll w$. Therefore, the proposed architecture requires $\lceil m/W \rceil + 1$ clock cycles to load the input registers, $d+2$ clock cycles to perform the multiplication, 1 clock cycle to perform the addition, and $\lceil m/W \rceil$ clock cycles to unload the result. Hence, it requires $2\lceil m/W \rceil + d + 4$ clock cycles in total to calculate $A^2(x)$ in $GF(2^m)$. If T_s denotes the number of clock cycles required to perform squaring operation, we can

Algorithm 9 RIA for field squaring.

Input: Binary polynomials $A(x)$ of degree at most $m-1$, $R(x)$ of degree m, and $B(x) = x^{d+1} + \ldots + x$.

Output: $C(x) = A^2(x) \mod R(x)$

1: **for** $i = 0$ to $m-1$ **do**
2: $\quad T(2*i) = A(i)$
3: $\quad T(2*i+1) = 0$
4: **end for**
5: $T_l = T_{m-1}, \ldots, T_0$ {Load T_l into register RX}
6: $T_h = T_{2m-2}, \ldots, T_m$ {Load T_h into register RA}
7: **for** $j = 0$ to $m-1$ **do**
8: $\quad c(0,j) = 0$
9: **end for**
10: **for** $i = 1$ to $d+2$ **do**
11: $\quad c(i,0) = [a(0) \wedge b(d+1-i)] \oplus c(i-1, m-1)$
12: $\quad$ **for** $j = 1$ to $m-3$ **do**
13: $\quad\quad c(i,j) = [a(j) \wedge b(d+1-i)] \oplus c(i-1, j-1) \oplus [c(i-1, m-1) \wedge r(j)]$
14: $\quad$ **end for**
15: **end for**
16: **for** $j = 0$ to $m-3$ **do**
17: $\quad c(m-3, j) = c(m-3, j) \oplus a(j)$
18: **end for**
19: **return** $C(x)$

write

$$T_s = 2\lceil \frac{m}{W} \rceil + d + 4 \tag{5.59}$$

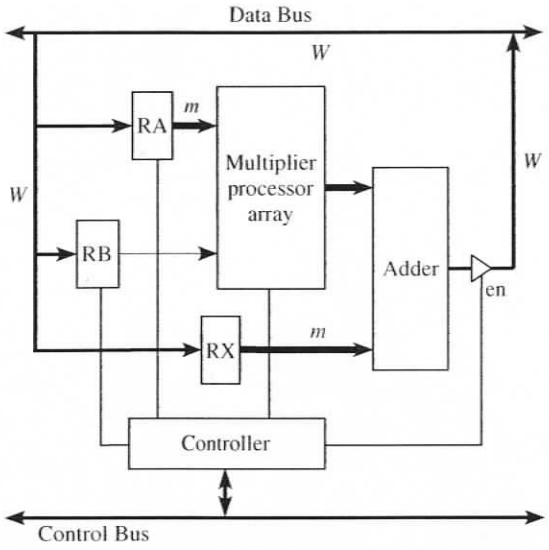


Figure 5.9: Processor array architecture for a $GF(2^m)$ squarer.

5.6 Proposed Processor Array Architecture for a $GF(2^m)$ Inversion

The inversion method described in Algorithm 6 requires $m-1$ squaring operations and $m-2$ multiplication operations. If T_{inv} denotes the number of clock cycles required to perform inversion operation, we can write

$$T_{inv} = (m-1)T_s + (m-2)T_m \tag{5.60}$$

Performance can be improved by using Algorithm 10 as proposed by Itoh et al. [156]. This algorithm is derived from the equation $A^{-1} = A^{2^m-2} = (A^{2^{m-1}-1})^2$, which is true for any element $A \in GF(2^m)$.

We can write

$$A^{2^t-1} = \begin{cases} (A^{2^{t/2}-1})^{2^{t/2}}(A^{2^{t/2}-1}) & t \text{ is even} \\ A(A^{2^{t-1}-1})^2 & t \text{ is odd} \end{cases} \tag{5.61}$$

The computations required for the exponentiation $2^{2^{m-1}-1}$ can be iteratively broken down. Algorithm 10 requires $\lfloor log_2(m-1) \rfloor + H(m-1) - 1$ multiplications and $m-1$ squarings, where $H(m-1)$ is the Hamming weight of $m-1$. This translates to

$$T_{inv} = (m-1)T_s + [\lfloor log_2(m-1) \rfloor + H(m-1) - 1]T_m \tag{5.62}$$

Since field inversion based on Algorithm 10 requires multiplication and squaring operations, the squaring architecture of Figure 5.9 can be utilized to perform inversion. However, the controller requires some modifications to accommodate the new operation.

Algorithm 10 Optimized field inversion using square and multiply [156].

Input: Field element $A(x) = (a_{m-1}x^{m-1} + \ldots + a_1x + a_0)$ and Binary representation of $m-1 = (m_{l-1}, \ldots, m_1, m_0)_2$

Output: $B = A^{-1}$

1: $B = A^{m_{l-1}} \bmod R(x)$
2: $e = 1$
3: **for** $i = l-2$ downto 0 **do**
4: $\quad B = B^{2^e} \bmod R(x)B$
5: $\quad e = 2e$
6: $\quad$ **if** $m_i = 1$ **then**
7: $\quad\quad B = B^2.A \bmod R(x)$
8: $\quad\quad e = e + 1$
9: $\quad$ **end if**
10: **end for**
11: $B = B^2 \bmod R(x)$
12: **return** B

5.7 Unified Architecture for $GF(2^m)$ Arithmetic Unit

The squaring architecture of Figure 5.9 performs many operations under proper instruction from the controller. These operations include: addition, multiplication, squaring, and inversion. The number of clock cycles required to perform multiplication, squaring, and inversion are given in Equations (5.50), (5.59), and 5.62, respectively. Addition operation requires $2\lceil m/W \rceil$ clock cycles to load input registers, one clock cycle to calculate the result, and $\lceil m/W \rceil$ clock cycles to unload the result. Hence, it requires $3\lceil m/W \rceil + 1$ clock cycles in total to calculate $A + B$ in $GF(2^m)$. If T_a denotes the number of clock cycles required to perform addition operation, we can write

$$T_a = 3\lceil \frac{m}{W} \rceil + 1 \tag{5.63}$$

The squaring architecture of Figure 5.9 can perform many operations in addition to the basic operations ($A + B$, AB, A^2, and A^{-1}). Out of these operations, we choose AB^2, $AB+C$, AB^2+C, and AB^{-1} to facilitate comparison with previous work. If T_1, T_2, T_3, and T_4 denote the number of clock cycles required to perform AB^2, $AB + C$, $AB^2 + C$, and AB^{-1} operations, respectively, then we can write

$$T_1 = T_s + T_m - \lceil \frac{m}{W} \rceil \tag{5.64}$$

$$T_2 = T_m + T_a - \lceil \frac{m}{W} \rceil \tag{5.65}$$

$$T_3 = T_s + T_m + T_a - 2\lceil \frac{m}{W} \rceil \tag{5.66}$$

$$T_4 = T_{inv} + T_m - \lceil \frac{m}{W} \rceil \tag{5.67}$$

Table 5.2 shows the clock cycles required to complete each $GF(2^m)$ operation for different m ($m \in \{163, 283, 571\}$) and different W ($W \in \{32, 64, 128\}$). The least expensive operation is addition. We noticed from Table 5.2 that addition is dominant

by the data bus width W. Squaring is more expensive than addition. We noticed from Table 5.2 that squaring is dominant by both the data bus width W and the parameter d since they are comparable. Multiplication is more expensive than addition and squaring. We noticed from Table 5.2 that multiplication is dominant by the field size m. Operations involving inversion are the most expensive but they are not sensitive to bus width W. They depend on the field width m as opposite to W.

5.8 Comparison to Other Implementations

In this section, we compare our proposed architecture with previous work [148, 150, 153, 159]. The factors considered for comparison include: functionality, cell count, cell complexity, cell latency, and operation latency. Table 5.3 compares the area of the proposed architecture to previous architectures area. We follow the same approach used in previous references, where the area is presented as a transistors count. The number of equivalent transistors is estimated using [160], where 2-input XOR, 2-input AND, 2×1 Mux all require 6 transistors each, 1-bit latch requires 8 transistors, and tri-state buffer requires 4 transistors. Table 5.4 compares other performance parameters such as functionality, cell count, cell complexity, cell latency, and operation latency.

Although the cell (PE) critical path delay of our architecture and the previous architectures are comparable, our architecture has a much lower area and fewer clock cycles required to complete a specific arithmetic operation. Moreover, we also considered the extra time required to load and unload registers.

Tables 5.3 and 5.4 show that the presented architecture has a much lower area and a shorter propagation delay (in terms of clock cycles). It is clear from Tables 5.3 that our design requires a very low area since it needs only $63m$ times the area of a transistor. Therefore, The proposed architecture required less area in comparison to the best previous design in [153], which requires $40m^2 + 71m$ times the area of a

Table 5.2: Cost of $GF(2^m)$ arithmetic operations measured in clock cycles for fields $GF(2^{163})$, $GF(2^{283})$, and $GF(2^{571})$ for different bus widths w.

Operation	Latency (clock cycles)	$m = 163$			$m = 283$			$m = 571$		
		$W =$			$W =$			$W =$		
		32	64	128	32	64	128	32	64	128
$A + B$	T_a	19	10	7	28	16	10	55	28	16
AB	T_m	181	172	169	310	298	292	625	598	586
A^2	T_s	23	17	15	34	26	22	50	32	24
A^{-1}	T_{inv}	5,193	4,140	3,789	12,716	10,328	9,134	36,055	25,444	20,728
AB^2	T_1	198	186	182	335	319	311	657	621	605
$AB + C$	T_2	194	179	174	329	309	299	662	617	597
$AB^2 + C$	T_3	211	193	187	354	330	318	694	640	616
AB^{-1}	T_4	5,368	4,309	3,956	13,017	10,621	9,423	36,662	26,033	21,309

Table 5.3: Area comparison between the proposed implementation and previous implementations.

Design	AND gates	XOR gates	1-bit Latches	2×1 Mux	3×1 Mux	Tri-state Tri-state	Total # of Transistors
Wei [148]	$3m^2$	$3m^2$	$13m^2$	0	0	0	$140m^2$
Wang [150]	$3m^2$	$3m^2$	$8.5m^2$	0	0	0	$128m^2$
Kim [159]	$3m^2 - 2m$	$3m^2 - 2m$	$9m^2 - 8m$	0	0	0	$108m^2 - 88m$
Lee [153]	m^2	$m^2 + 3m$	$3.5m^2 + 3m$	$3m$	m	0	$40m^2 + 71m$
Our Design	m	$2m$	$3m$	$2m - 2$	0	$2m$	$62m$

Table 5.4: Detailed comparison between proposed implementations and previous implementations.

Design	Function	Number of cells	Basic cell						Cell latency	Latency (clock cycles)
			X_2	X_3	X_4	A_2	L	Mux		
Wei [148]	AB^2+C	m^2	1	1	0	3	10	0	$T_{A_2}+T_{X_3}+T_L$	$3m$
Wang [150]	$C+AB^2$	$m^2/2$	0	0	2	6	17	0	$T_{A_2}+T_{X_4}$	$2m+m/2$
Wang [150]	AB^{-1}	$\frac{m^3-m^2}{2}$	0	0	6	6	17	0	$T_{A_2}+T_{X_4}$	$2m^2-3m/2$
Kim [159]	AB^2	m^2	1	0	0	1	1	0	$T_{A_2}+T_{X_3}$	$3m-2$
Lee [153]	AB^2	$m\lceil m/2\rceil$	2	0	0	2	11	2	$T_{A_2}+T_{X_2}+T_L+T_S$	$4m$
Our Design	Table 5.2	m	1	0	0	1	1	0	$T_{A_2}+T_{X_3}+T_L$	Table 5.2

T_{X_n}: delay time for n-input XOR gate

T_{A_n}: delay time for n-input AND gate

T_L: delay time for i-bit latch

T_S: delay time for switch or Mux

transistor.

Wei [148] proposed an architecture to calculate $AB^2 + C$, which requires an area of m^2 PEs and a latency of $3m$ clock cycles to complete this operation. Our proposed architecture has an area of m PEs and calculates $AB^2 + C$ in $\ll 2m$ clock cycles (Table 5.2).

Wang et al. [150] proposed two architectures to calculate $C + AB^2$ and AB^{-1}. The first architecture requires an area of $m^2/2$ PEs and a latency of $2m + m/2$ clock cycles to complete this operation. Our proposed architecture has an area of m PEs and calculates $AB^2 + C$ in $\ll 2m$ (Table 5.2). The second Architecture requires an area of $m^2(m-1)/2$ PEs and a latency of $2m^2 - 3m/2$ clock cycles to complete this operation. Our proposed architecture has an area of m PEs and calculates AB^{-1} in $\ll m^2$ clock cycles (Table 5.2).

Kim et al. [159] proposed an architecture to calculate AB^2, which requires an area of m^2 PEs and a latency of $3m - 2$ clock cycles to complete this operation. Our proposed architecture has an area of m PEs and calculates AB^2 in $\ll 2m$ clock cycles (Table 5.2).

Lee et al. [153] proposed an architecture to calculate AB^2, which requires $m\lceil m/2 \rceil$ PEs and a latency of $4m$ clock cycles to complete this operation. Our proposed architecture has an area of m PEs and calculates AB^2 in $\ll 2m$ clock cycles (Table 5.2).

As a conclusion, the previously proposed architectures [148, 150, 153, 159] require a huge area, especially for large m (as in the case of ECC). Therefore, implementing these architectures for large m is not realistic. Moreover, they consume a lot of power since they have a huge area.

We use a novel approach to utilize the area by making sure that all implemented PEs are fully utilized and always active. As a result, our architecture is very realistic for large m. Our approach does not only reduce the area dramatically, but also

decreases the number of clock cycles to complete a specific operation as well as the power consumption. Moreover, previous architectures use general polynomials as a generator for the field element, which in turn makes them not suitable for any cryptographic algorithm, since general polynomials are not secure [114].

5.9 Implementation Results

This section introduces our synthesis approach and the numerical results of our implementation.

5.9.1 Experimental Setup

To evaluate our proposed architecture, we implemented the proposed $GF(2^m)$ ALU unified architecture for three different m values as recommended by NIST. These sizes are for $m \in \{163, 283, 571\}$.

We modeled these three different implementations using VHDL and implemented them on an FPGA kit. Xilinx ISE 9.1 [33] is used to synthesize the design and provide post-placement area and timing results. The architecture is implemented on a Xilinx XC2V4000-6 FPGA device.

5.9.2 Synthesis Results

The FPGA synthesis results for the different m are listed in Table 5.5. Table 5.5 shows the delay time measured in nanoseconds ns for commonly used field operations. Although critical path delay does not depend on m, there is a slight variation in maximum frequency f_{max}. This variation is due to FPGA routing. As m increases the number of slices needed increases, which in turn increases the routing pathes, which increase the critical path delay slightly.

Table 5.5: Performance results of our proposed architecture implementations.

m (bits)	W (bits)	**Area** (slice)	f_{Max} (MHz)	**Operation latency** (ns)							
				$A+B$	AB	A^2	A^{-1}	AB^2	$AB+C$	AB^2+C	AB^{-1}
163	32	1,068	264	72	686	87	19,676	750	735	799	20,339
	64			38	652	64	15,687	705	678	731	16,327
	128			27	640	57	14,357	690	659	709	14,989
283	32	1,802	236	119	1,314	144	53,878	1,419	1,394	1,500	55,153
	64			68	1,263	110	43,760	1,352	1,309	1,398	45,001
	128			42	1,237	93	38,701	1,318	1,267	1,347	39,925
571	32	3,606	226	243	2,760	221	159,219	2,901	2,923	3,065	161,899
	64			124	2,641	141	112,361	2,742	2,725	2,826	114,962
	128			71	2,588	106	91,535	2,672	2,636	2,720	94,101

Table 5.6 compares our worst case results to other selected designs. Since the previous implementations have different m lengths, it is hard to conduct a precise comparison with our architecture. However, we proved that we achieved better results. Although the fastest multiplier (in [161]) calculates AB in 1.48 μs for $m = 191$, we calculated AB in 1.3 μs for $m = 283$. On the other hand, we found a single result for field inversion [161]. Although the inverter in [161] calculates A^{-1} in 145 μsec for $m = 191$, we calculated A^{-1} in 53.9 μs for $m = 283$. We could not compare our architecture area with the fastest previous architecture due to lack of information. It is clear from Table 5.6 that our architecture is faster than all prior FPGA implementations.

5.10 Chapter Summary

In this chapter, a novel, area-speed efficient, high-speed architecture for the finite fields ALU has been presented. The proposed architecture is a processor array based on most significant bit multiplication algorithm and polynomial basis. In order to find the best design that fits VoIP applications, we did a design space exploration to optimize the area and speed of the proposed architecture. We used a novel technique to optimize the number of required processing elements. Therefore, our architecture requires only m processing elements as compared to $m^2/2$ for the best previous design. Moreover, the required clock cycles for multiplication is $\ll 2m$ as compared to $2m + m/2$ for the best previous design. We used NIST-recommended irreducible polynomials, which makes our design secure and more suitable for cryptographic applications. In order to evaluate the proposed architecture and measure its performance, the proposed architecture is implemented for $GF(2^m)$ ($m \in \{163, 283, 571\}$) on Xilinx XC2V4000-6 device. We achieve a maximum frequency of 264 MHz, which allows the architecture to calculate $GF(2^{163})$

Table 5.6: Comparison between our implementations and previous implementations.

Design	Platform	**Area** (slice)	**Frequency** (MHz)	m (bits)	**Operation latency** (μs)	
					AB	A^{-1}
Örs et al. [162]	Xilinx V812E-8	806	98	128	3.974	-
		1,548	100	256	7.686	-
		2,972	95	512	16.171	-
Crowe et al. [163]	Xilinx XC2V2000-6	5,267	44.9	256	5.75	-
Alberto et al. [164]	Xilinx XCV300	3,072	200	239	3.1	-
Miguel et al. [161]	Xilinx XC2V1000-4	-	166	191	1.48	135
Kitsos et al. [165]	Xilinx XC2V1000E-8	-	22.1	163	7.4	-
Our design	Xilinx XC2V4000-6	1,086	264	163	0.686	19.7
		1,802	236	283	1.3	53.9
		3,606	226	571	2.8	159

multiplication in 640 ns, and inversion in 14.357 μs.

The purpose of proposing the field ALU is to be utilized by the ECC protocol for key exchange and user authentication protocols. In the next chapter, we propose a high speed ECC architecture based on a high-radix *scalar* multiplication targeting VoIP applications.

Chapter 6

A High-Speed, High-Radix, Processor Array Architecture for ECC Over $GF(2^m)$

This chapter presents the fourth contribution of this dissertation. We propose a high speed ECC architecture based on a high-radix *scalar* multiplication targeting VoIP applications.

The main contributions in this chapter are summarized as follows:

- Propose a new high-radix *scalar* multiplication algorithm.
- Design a merged double-and-add elliptic curve ALU based on the proposed algorithm. The merged double-and-add ALU combines point doubling and adding operations in one architecture, which in turn reduces the critical path delay. The merged double-and-add ALU calculates $2^wQ + iP$ in a single step.
- Utilize an optimized processor array based field ALU in performing *scalar* multiplication calculations, which implements addition, squaring, multiplication,

and division over $GF(2^m)$.

- Speed up calculations of the $2^wQ + iP$ operations by using 4 parallel field ALUs

This chapter is organized first by Section 6.1, which introduces this chapter. Section 6.2 describes related work in ECC cryptosystems. Section 6.3 describes the the mathematical background of the Elliptic Curve. In Section 6.4, we introduce our new algorithm and the related proposed architecture. Implementation results and the comparison with prior efforts are shown in Section 6.5. Section 6.6 concludes the chapter.

6.1 Introduction

Elliptic curves are used in several kinds of cryptosystems, including key exchange protocols and digital signature algorithms (DSA) [26, 27]. Elliptic Curve-based cryptosystems (ECC) were proposed in 1985 by N. Koblitz [166] and V. Miller [167]. ECC algorithms for encryption, digital signature, or key-exchange belong to the class of group-based protocols, such as ElGamal and Diffe-Hellman protocols [28,32], which base their security on the difficulty in solving the Discrete Logarithm Problem (DLP) over a group of elliptic curve points.

ECC is very attractive because of its smaller key length and increased theoretical robustness. There is no known sub-exponential algorithm to solve the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is the foundation of ECC [28]. The relatively small key size (a 160-bit key provides the same security as a 1024-bit RSA modulus) is a major advantage for devices with limited hardware resources such as smartcards, cell phones, and PDAs.

In ECC based algorithms, the underlying cryptographic primitive is elliptic curve *scalar* multiplication [28]. This operation is the most computationally intensive

step in each algorithm. In applications where many clients authenticate to a single server (such as a server supporting SSL or WTLS), the computation of the *scalar* multiplication becomes the bottleneck, which limits throughput. Therefore, it is required to accelerate the elliptic curve *scalar* multiplication with a specialized hardware to increase throughput and reduce the server computational burden.

Elliptic Curve Digital Signature Algorithm (ECDSA) is an example of ECCs, which are based on the DLP. ECDSA parameters have to be chosen carefully to optimize the efficiency of the implementation. Incorrect selection of ECDSA parameters may lead to an insecure system. The National Institute for Standards and Technology (NIST) recommends ten finite fields, five of which are binary fields, for use in the ECDSA [114]. The binary fields include $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$, which are generated using the irreducible polynomials listed in Table 5.1. NIST curves have carefully been selected for both high security and efficient implementation.

6.2 Related Work

The difficulty of the elliptic curve discrete logarithm problem (ECDLP) makes ECC cryptosystems suitable for applications that need long-term security and low bandwidth measurements [147]. In 2002, the USA Government adopted ECC for protecting mission-critical information. For instance, the NIST has recommended specific curves for implementation [114] and the IEEE has provided detailed specification for the choices of their private key length and underlaying field parameters [26, 27].

ECC research has been extensively conducted and it can first be divided into two groups depending on the underlying field representation: prime field, $GF(p)$ and binary field, $GF(2^m)$ [28]. Two bases, Optimal Normal Basis (ONB) and

Polynomial Basis (PB) are commonly used for manipulating binary fields. With PB representation, field elements are represented as polynomials, while with ONB the irreducible polynomial used in PB is not required. The binary field is the most suitable for hardware implementation because elements in Galois field $GF(2^m)$ are represented as binary numbers and all the operations are performed using standard logic gates in the binary domain [28, 113, 154]. ONB is preferred to implement fast squaring [28].

Fast *scalar* multiplication requires efficient field multiplication, inversion [156], squaring and an efficient coordinate system [168]. In recent years, there has been a lot of research in software [169, 170] and hardware [171, 172] to improve the performance of *scalar* multiplication. The first ECC hardware implementation [173] was presented in 1989. Previous hardware work includes: the first ASIC implementation with the Motorola M68008 microcontroller [173], ASIC designs for field operations over specific fields [165, 174, 175], an ECC implementation on the 8051 microprocessor in smart cards [176], an ECC processor on a smart card device [177], and recent FPGA implementations for ECC designs including [146, 147, 171, 172, 178–185]. To improve the performance of *scalar* multiplication, we propose a polynomial basis, a high-radix architecture to calculate the *scalar* multiplication based on a modified version of the sliding window *scalar* multiplication algorithm [28] and the processor array field ALU.

6.3 Background of Elliptic Curve Cryptography

In this section, we briefly introduce the main concepts of the elliptic curve, including the mathematical background behind ECC, the *scalar* multiplication operation, and point doubling and adding operations.

6.3.1 ECC Mathematical Derivation

An elliptic curve E over a field K, denoted $E(K)$, is defined by the equation [28]

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{6.1}$$

where a_1, a_2, a_3, a_4, $a_6 \in K$. The *discriminant* of E is $\Delta \neq 0$ and is defined as follows [28]

$$\begin{aligned} \Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\ d_2 &= a_1^2 + 4a_2 \\ d_4 &= 2a_4 + a_1a_3 \\ d_6 &= a_3^2 + 4a_6 \\ d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \end{aligned} \tag{6.2}$$

Equation (6.1) is called a Weierstrass form of an elliptic curve. In ECC, curves without any singular points are used. In practice, the Weierstrass equation of an elliptic curve can be transformed into simpler equations depending on the characteristic of the underlying field. Binary and prime fields are two popular classes of fields used for implementations. For an elliptic curve E defined over binary fields $GF(2^m)$, Equation (6.1) can be written as

$$y^2 + xy = x^3 + ax^2 + b \tag{6.3}$$

where $a, b \in GF(2^m)$ and $a \neq 0$. In this case, if $\Delta = b \neq 0$, the curve is said to be *non-supersingular*.

On the other hand, the Weierstrass equation over prime fields $GF(p)$ takes a simpler form

$$y^2 = x^3 + ax + b \tag{6.4}$$

where $a, b \in GF(p)$ and the curve is free of any singularities if $\Delta = 4a^3 + 27b^2 \neq 0$. The set $E(K)$ of rational points on an elliptic curve E defined over a field K is an abelian

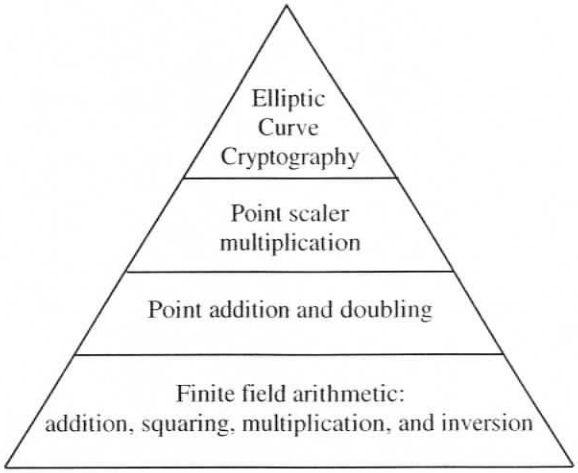


Figure 6.1: Hierarchy of operations in ECC.

group, where the operation is defined by the well-known law of chord and tangent, and the identity element, which is the special point $\mathcal{O}$, called *point at infinity* [28].

6.3.2 *Scalar* Multiplication

The most important operation in ECC is the *Scalar* multiplication operation. Let $P \in E(K)$ and $k \in N$, the operation of computing the new point

$$Q = k \times P = \underbrace{P + P + \ldots P}_{k \text{ times}} \tag{6.5}$$

is called *scalar* multiplication. *Scalar* multiplication is the main building block used in ECC. In an ECC implementation, *scalar* multiplication is the most computationally expensive operation. *Scalar* multiplication is computed by repeated point doubling and addition, which require field adding, squaring, multiplication, and division. Figure 6.1 illustrates the operations hierarchy in ECC.

Several methods have been proposed in the literature to speed up *scalar* multiplication, which use various representations of the base point P (affine

coordinates, standard projective coordinates, Jacobian projective coordinates, López-Dahab projective coordinates), various representations of the *scalar* k (binary, ternary, NAF, w-NAF, etc.), and various curve operations (addition, doubling, halving, and tripling). The computational cost (time) of these curve operations depends on the cost of the arithmetic operations that have to be performed in the underlying field. *Scalar* multiplication requires addition, multiplication, squaring, and inversion in the underlying field. From now on, we denote these operations as A, M, S and I respectively, while the number of clock cycles required to perform these operations is denoted as T_a, T_m, T_s and T_i, respectively. The area required to implement these operations is denoted as A_a, A_m, A_s and A_i, respectively. In an actual implementation, addition and squaring are much faster than multiplication, while inversion is more computationally expensive than the multiplication (inversion based on Fermats Theorem requires at least seven multiplications in $GF(2^m)$ if $m \geqslant 128$ [186]).

The l-bit binary expansion of an integer k is written as $k = \sum_{i=0}^{l-1} k_i 2^i$ where $k_i \in \{0, 1\}$. Algorithms 11 and 12 introduce the binary method, which is considered the simplest (and oldest) algorithm for evaluating *scalar* multiplication. Algorithm 11 processes the bits of k from left to right, while Algorithm 12 processes the bits of k from right to left [28].

Assuming an average Hamming weight, the expected number of elliptic curve additions in Algorithms 11 and 12 is approximately $0.5l$ and the number of doublings is approximately l. Therefore, the expected delay for the *scalar* multiplication (T_{SM}) is approximately [28]

$$T_{SM} = \frac{l \times T_{P_A}}{2} + l \times T_{P_D} \tag{6.6}$$

where T_{P_A} represents the required time for point addition and T_{P_D} represents the required time for point doubling.

Algorithm 11 The left-to-right binary method for *scalar* multiplication algorithm.

Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ and point P on the elliptic curve E over $GF(2^m)$

Output: $Q = kP$

1: $Q = \mathcal{O}$
2: **for** $i = l - 1$ downto 0 **do**
3: $\quad Q = 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q = Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** Q

Algorithm 12 The right-to-left binary method for *scalar* multiplication algorithm.

Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ and point P on the elliptic curve E over $GF(2^m)$

Output: $Q = kP$

1: $Q = \mathcal{O}$
2: **for** $i = 0$ to $l - 1$ **do**
3: $\quad$ **if** $k_i = 1$ **then**
4: $\quad\quad Q = Q + P$
5: $\quad$ **end if**
6: $\quad Q = 2Q$
7: **end for**
8: **return** Q

The number of elliptic curve point additions can be reduced by replacing the binary representation of k with a representation that has fewer non-zero coefficients. The nonadjacent form (NAF) and the window NAF (w-NAF) are examples of representations that have fewer non-zero coefficients.

The non-adjacent form (NAF) of a positive integer k is an expression $k = \sum_{i=0}^{l-1} k_i 2^i$ where $k_i \in \{0, \pm 1\}$, $k_{l-1} \neq 0$, and no two consecutive digits k_i are non-zero. Non-adjacent form (NAF) of an integer k can be used to reduce the number of non-zero coefficients to nearly $l/3$. Therefore, the expected delay of the *scalar* multiplication is approximately [28]

$$T_{SM} = \frac{lT_{P_A}}{3} + lT_{P_D} \tag{6.7}$$

Algorithm 13 modifies the left-to-right binary method for *scalar* multiplication (Algorithm 11) by using NAF representation instead of binary representation of k.

Algorithm 13 Binary NAF method for *scalar* multiplication.

Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ and point P on the elliptic curve E over $GF(2^m)$

Output: $Q = kP$

1: Calculate the NAF representation of k
2: $Q = \mathcal{O}$
3: **for** $i = l - 1$ downto 0 **do**
4: $\quad Q = 2Q$
5: $\quad$ **if** $k_i = 1$ **then**
6: $\quad\quad Q = Q + P$
7: $\quad$ **end if**
8: $\quad$ **if** $k_i = -1$ **then**
9: $\quad\quad Q = Q - P$
10: $\quad$ **end if**
11: **end for**
12: **return** Q

The expected delay of Algorithm 13 can be reduced using a window method

(w-NAF), which process w digits of k at a time, as described in Algorithm 14.

A width-w NAF of an integer k is an expression in the form $k = \sum_{i=0}^{l-1} k_i 2^i$, where each non-zero coefficient k_i is an odd integer, where $|k_i| < 2^{w-1}$, and at most one of w consecutive digits is zero. The length of width-w NAF l is at most one bit longer than the binary representation of k and $k_{l-1} \neq 0$. The width-w NAF is denoted by $NAF_w(k)$, while the number of non-zero coefficients in $NAF_w(k)$ is approximately $l/(w+1)$ [28].

The expected delay for the *scalar* multiplication using $NAF_w(k)$ (Algorithm 14) is approximately

$$T_{SM} = T_{P_D} + (2^{w-2} - 1)T_{P_A} + \frac{lT_{P_A}}{w+1} + lT_{P_D} \tag{6.8}$$

Algorithm 14 Window NAF method for *scalar* multiplication.

Input: Window width w, $k = (k_{l-1}, ..., k_1, k_0)_2$ and point P on the elliptic curve E over $GF(2^m)$

Output: $Q = kP$

1: Calculate the w-NAF representation of k
2: Pre-computation, Compute $P_i = iP$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$
3: $Q = \mathcal{O}$
4: **for** $i = l - 1$ downto 0 **do**
5: $\quad Q = 2Q$
6: $\quad$ **if** $k_i \geq 1$ **then**
7: $\quad\quad Q = Q + P_{k_i}$
8: $\quad$ **end if**
9: $\quad$ **if** $k_i \leq -1$ **then**
10: $\quad\quad Q = Q - P_{k_i}$
11: $\quad$ **end if**
12: **end for**
13: **return** Q

6.3.3 Point Addition and Doubling

Let E be an elliptic curve defined over a field K and let $P_1, P_2 \in E(K)$. There exists a rule for adding P_1 and P_2, which generates a third point in $E(K)$. A geometrical interpretation is given by the fact that the three points P_1, P_2 and $-(P_1 + P_2)$ form a straight line. Together with this addition law and a special point $\mathcal{O}$ called the *point at infinity* playing the role of the identity, the set of points forms an abelian group.

In our implementation, we consider ECC over binary fields since it is the most suitable for hardware implementation [28, 113]. Assume $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, and $P_1 \neq -P_2$. The sum $P_3 = (x_3, y_3) = P_1 + P_2$ is computed as follows [28]: If $P_1 \neq P_2$ then

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \end{aligned} \tag{6.9}$$

If $P_1 = P_2$ (i.e., point doubling) then

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \\ y_3 &= x_1^2 + \lambda x_3 + x_3 \\ \lambda &= x_1 + \frac{y_1}{x_1} \end{aligned} \tag{6.10}$$

In either case, the computation requires two field multiplications, a squaring, and a field inversion, denoted by $2A + S + I$ [28]. Actual implementation of the elliptic curves indicates that field squaring and field addition are much faster than field multiplication, while field inversion is more expensive than the multiplication (inversion based on Fermats Theorem requires at least seven multiplications in $GF(2^m)$ if $n \geq 128$ [186]). Therefore, the projective coordinates have been suggested to replace the inversion operation by multiplications, where a projective point (X, Y, Z), $Z \neq 0$, maps to the affine point $(X/Z^c, Y/Z^d)$. The Jacobian projective coordinates have $c = 2$, $d = 3$. The sum $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$ in the Jacobian

projective coordinates is computed as follows [28, 187]: If $P_1 \neq P_2$ then

$$\begin{aligned} T_1 &= X_1 Z_2^2 + X_2 Z_1^2 \\ T_2 &= Y_1 Z_2^3 + Y_2 Z_1^3 \\ T_3 &= T_1 Z_1 \\ T_4 &= T_2 X_2 + T_3 Y_2 \\ Z_3 &= Z_2 T_3 \\ X_3 &= a Z_3^2 + T_2(Z_3 + T_2) + T_1^3 \\ Y_3 &= X_3(Z_3 + T_2) + T_4 T_3^2 \end{aligned} \tag{6.11}$$

If $P_1 = P_2$ (i.e., point doubling) then

$$\begin{aligned} Z_3 &= X_1 Z_1^2 \\ X_3 &= X_1^4 + b Z_1^8 \\ Y_3 &= X_1^4 Z_3 + X_3(Z_3 + X_1^2 + Y_1 Z_1) \end{aligned} \tag{6.12}$$

In Jacobian projective coordinates, a point addition, Equation (6.11), requires 5 field squarings and 15 general field multiplications, whereas point doubling, Equation (6.12), requires 5 field squarings and 5 general field multiplications. In the case of $Z_1 = 1$ and $a \in \{0, 1\}$, a point addition requires 4 field squarings and 10 general field multiplications, whereas a point doubling requires 3 field squarings and 2 general field multiplications. Note that there is no known security threat if a is restricted to 0 or 1 [28].

López and Dahab introduced new formulae in projective coordinates, where a projective point, (X, Y, Z), $Z \neq 0$, maps to the affine point $(X/Z, Y/Z^2)$.

The sum $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$ is computed as follows [28, 188]: If $P_1 \neq P_2$,

$Z_1 = 1$, and $a \in \{0, 1\}$ then

$$\begin{aligned} T_1 &= Y_1 Z_2^2 + Y_2 \\ T_2 &= X_1 Z_2 + X_2 \\ T_3 &= Z_2 T_2 \\ Z_3 &= T_3^2 \\ T_4 &= T_2^2 (T_3 + Z_2^2) \\ T_5 &= T_1 T_3 \\ X_3 &= T_1^2 + T_4 + T_5 \\ T_6 &= X_3 + X_2 Z_3 \\ T_7 &= X_3 + Y_2 Z_3 \\ Y_3 &= T_5 T_6 + Z_3 T_7 \end{aligned} \tag{6.13}$$

If $P_1 = P_2$ (i.e., point doubling), and $a \in \{0, 1\}$ then

$$\begin{aligned} Z_3 &= X_1^2 Z_1^2 \\ X_3 &= X_1^4 + b Z_1^4 \\ T_1 &= a Z_3 + Y_1^2 + b Z_1^4 \\ Y_3 &= b Z_1^4 Z_3 + X_3 T_1 \end{aligned} \tag{6.14}$$

In the López-Dahab projective coordinates, a point addition (Equation (6.13)) requires 4 field squarings and 9 general field multiplications, while a point doubling (Equation (6.14)) requires 5 field squarings, 4 general field multiplications, and 4 field additions.

6.4 Proposed Architecture

We propose combined point doubling and addition to calculate $2Q + P$ in a single unit, which reduces the critical path delay without no increase in the required area. Moreover, combining point quadrupling and addition to calculate $4Q + iP$ in a single

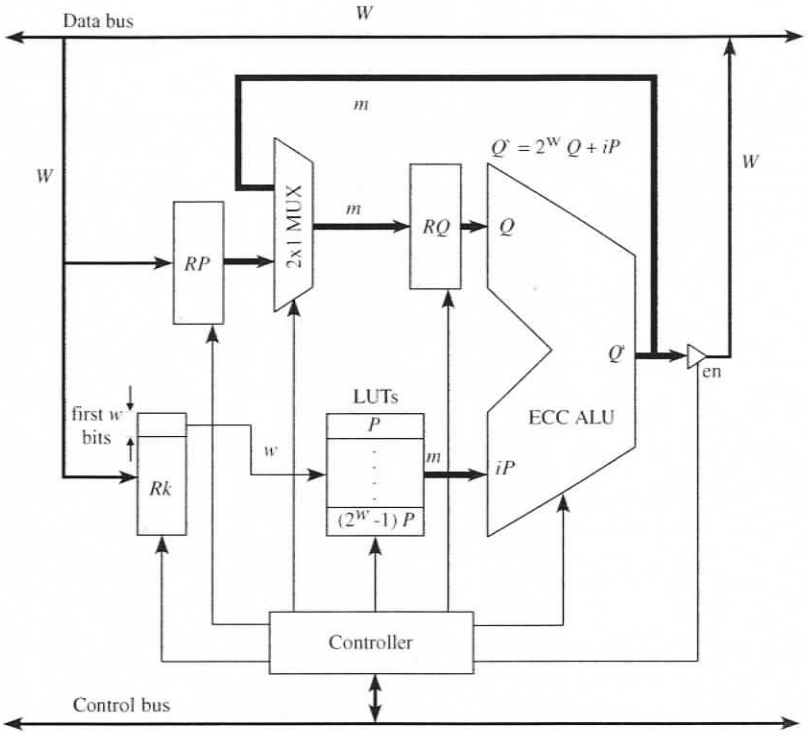


Figure 6.2: Proposed architecture for *scalar* multiplication calculation.

unit reduces the critical path delay at the expense of more area. As a result, we adopt a high-radix implementation of the *scalar* multiplication, which requires a modified version of the sliding window *scalar* multiplication algorithm [28]. Algorithm 15 lists the modified version of the sliding window *scalar* multiplication algorithm. We denoted the modified sliding window algorithm by MSWA. Figure 6.2 illustrates the proposed architecture based on the high-radix design. The proposed architecture deals with radix-2^w, where w represents the number of binary bits processed at a time. The proposed architecture consists of 2^w-location Look-Up Table (LUT), register RP, register Rk, register RQ, 2-1 multiplexor, elliptic curve ALU, controller, and W-bit data buses. The role of each component is as follow:

Algorithm 15 MSWA algorithm for *scalar* multiplication.

Input: Window width w, $k = (k_{l-1}, ..., k_1, k_0)_2$ and point P on the elliptic curve E over $GF(2^m)$

Output: $Q = kP$

1: Calculate the w-NAF representation of k
2: Pre-computation. Compute $P_i = iP$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$
3: $Q = \mathcal{O}$
4: **for** $i = 0$ to $l - 1$ step w **do**
5: $\quad i = k_{w+i}, .., k_i$
6: $\quad Q = 2^w Q + iP$
7: **end for**
8: **return** Q

- **2^w-location Look-Up table (LUT):** Each location stores the pre-calculated components iP_x, iP_y, and iP_z of the point iP where $0 \leq i < 2^w$. Each component is an m-bit word. The LUT receives a w-bit address and outputs the corresponding iP components (iP_x, iP_y, and iP_z).

- **RP registers:** Three m-bit registers that hold the components P_x, P_y, and P_z of the point P.

- **Register** Rk: An m-bit register, which holds the *scalar* value k. The register outputs the least significant w bits then shifts its contents to the right by w bits each clock cycle.
- RQ **registers:** Three m-bit registers that hold the components Q_x, Q_y, and Q_z of the point Q.
- 2-1 **multiplexor:** Chooses between the components of the initial point P and the components of the intermediate value $\grave{Q} = 2^w Q + iP$.
- **Elliptic curve ALU:** Receives old Q components and iP components to calculate the new components of $\grave{Q} = 2^w Q + iP$
- **Controller:** coordinates data movement between registers and operations of the elliptic curve ALU.

The proposed architecture calculates kP as follows:

1. The controller loads the components of the point P into the RP registers.
2. The controller loads the *scalar* value k into the Rk register.
3. The controller controls the multiplexors to select the outputs of the RP registers.
4. The controller stores the output of the multiplexors into the RQ registers.
5. The RK register outputs a w-bit address of the required iP.
6. The LUT receives the output of the RK register and outputs the required iP.
7. The Elliptic curve ALU receives the outputs of the LUT and the RQ registers and produces $\grave{Q} = 2^w Q + iP$ after some delay.

8. The controller shifts the Rk register to the right by w bits. At the same time the controller controls the multiplexor to select point $\grave{Q}$.

9. Repeat steps 4, 5, 6, 7, and 8 $\lceil l/w \rceil$ times.

We assume that all the internal data buses are m bits wide, whereas the external data buses are W bits wide. Since the external data buses are W bits wide, register Rk requires $\lceil l/W \rceil$ clock cycles to load the *scalar* value k from the bus, whereas register RP requires $3\lceil m/W \rceil$ clock cycles to load the input components, P_x, P_y, and P_z. The results Q_x, Q_y, and Q_z are unloaded in $3\lceil m/W \rceil$ clock cycles also. Since the internal data buses are m bits wide, the elliptic curve ALU requires 6 clock cycles to load the components Q_x, Q_y, Q_z, iP_x, iP_y, and iP_z, whereas performing $2^wQ + iP$ operation requires T_{ALU} clock cycles. Since T_{ALU} depends on w, we evaluate T_{ALU} for each value of w later.

The elliptic curve ALU needs to perform $2^wQ + iP$ operations $\lceil l/w \rceil$ times to calculate the *scalar* multiplication kP. As a result, the *scalar* multiplication architecture requires $\lceil l/W \rceil + 3\lceil m/W \rceil$ clock cycles to load the input registers, $\lceil l/w \rceil (T_{ALU} + 6)$ to calculate the *scalar* multiplication kP, and $3\lceil m/W \rceil$ to unload the output registers. Therefore, the number of clock cycles required to perform the *scalar* multiplication T_{SM} can be written as

$$T_{SM} = \lceil \frac{l}{W} \rceil + 6\lceil \frac{m}{W} \rceil + \lceil \frac{l}{w} \rceil (T_{ALU} + 6) \tag{6.15}$$

6.4.1 Radix-2 Implementation of *scalar* multiplication

Figure 6.3 illustrates the hardware implementation of point doubling according to the López-Dahab technique Equation (6.14), with $a = 1$. Point doubling implementation requires an area of $4A_m + 5A_s + 4A_a$ and a critical path delay of $2T_m + 2T_s + 2T_a$ clock cycles. Table 6.1 lists the critical path delay calculations for point doubling

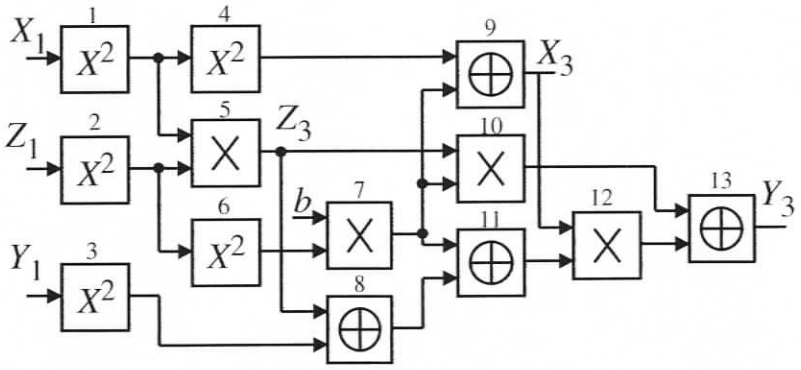


Figure 6.3: Implementation of López-Dahab point doubling. $\bigoplus$: adder in $GF(2^m)$, $\times$: multiplier in $GF(2^m)$, X^2: squarer in $GF(2^m)$.

implementation of Figure 6.3, which is the direct implementation of Equation 6.14.

Figure 6.4 illustrates the hardware implementation of point addition according to the López-Dahab technique (Equation (6.13)) for $a = 1$. Point addition implementation requires an area of $9A_m + 4A_s + 8A_a$ and a critical path delay of $4T_m + T_s + 3T_a$. Table 6.2 lists the critical path delay calculations for point addition implementation of Figure 6.4, which is the direct implementation of Equation 6.13.

The *scalar* multiplication depends strongly on point doubling and adding operations. All of the proposed *scalar* multiplication algorithms (e.g. binary, NAF, w-NAF, etc.) calculate point doubling and point adding separately (i.e., in two separate steps) [28], which in turn requires an area of $13A_m + 9A_s + 12A_a$ and a critical path delay of $6T_m + 3T_s + 5T_a$. If the double-and-add operations are done in a single architecture, the critical path delay could be reduced with the same required area. Figure 6.5 illustrates these two different approaches. The combined double-and-add architecture requires an area of $13A_m + 9A_s + 12A_a$ and a critical path delay of $5T_m + 2T_s + 3T_a$ as detailed in Table 6.3. In Table 6.3, the inputs X_2, Y_2, and Z_2 are the outputs from the point doubling circuit. Therefore, the input delays of these

Table 6.1: Point doubling critical path delay calculations, $\bigoplus$: adder in $GF(2^m)$, $\times$: multiplier in $GF(2^m)$, X^2: squarer in $GF(2^m)$, T_m: multiplication delay, T_s: squaring delay, T_a: addition delay.

Node	Function	Inputs	Output	Input delay	Node Delay	Output delay
1	X^2	X_1	X_1^2	0	T_s	T_s
2	X^2	Z_1	Z_1^2	0	T_s	T_s
3	X^2	Y_1	Y_1^2	0	T_s	T_s
4	X^2	X_1^2	X_1^4	T_s	T_s	$2T_s$
5	$\times$	X_1^2,Z_1^2	Z_3	$Max(T_s, T_s)$	T_m	$T_m + T_s$
6	X^2	Z_1^2	Z_1^4	T_s	T_s	$2T_s$
7	$\times$	Z_1^4,b	bZ_1^4	$\mathrm{Max}(2T_s, 0)$	T_m	$T_m + 2T_s$
8	$\bigoplus$	Z_3,Y_1^2	$Z_3 + Y_1^2$	$\mathrm{Max}(T_m + T_s, T_s)$	T_a	$T_m + T_s + T_a$
9	$\bigoplus$	X_1^4,bZ_1^4	X_3	$\mathrm{Max}(2T_s, T_m + 2T_s)$	T_a	$T_m + 2T_s + T_a$
10	$\times$	Z_3,bZ_1^4	$bZ_1^4Z_3$	$\mathrm{Max}(T_m + T_s, T_m + 2T_s)$	T_m	$2T_m + 2T_s$
11	$\bigoplus$	bZ_1^4,$Z_3 + Y_1^2$	T_1	$\mathrm{Max}(T_m + 2T_s, T_m + T_s + T_a)$	T_a	$T_m + 2T_s + T_a$
12	$\times$	X_3,T_1	X_3T_1	$\mathrm{Max}(T_m + 2T_s + T_a, T_m + 2T_s + T_a)$	T_m	$2T_m + 2T_s + T_a$
13	$\bigoplus$	$bZ_1^4Z_3$,X_3T_1	Y_3	$\mathrm{Max}(2T_m + 2T_s, 2T_m + 2T_s + T_a)$	T_a	$2T_m + 2T_s + 2T_a$

Table 6.2: Point addition critical path delay calculations, $\bigoplus$: adder in $GF(2^m)$, $\times$: multiplier in $GF(2^m)$, X^2: squarer in $GF(2^m)$, T_m: multiplication delay, T_s: squaring delay, T_a: addition delay.

Location	Function	Inputs	Output	Input delay	Node delay	Output delay
1	X^2	Z_2	Z_2^2	0	T_s	T_s
2	$\times$	X_1, Z_2	X_1Z_2	Max(0, 0)	T_m	T_m
3	$\times$	Y_1, Z_2^2	$Y_1Z_2^2$	Max(0, T_s)	T_m	$T_m + T_s$
4	Add	X_1Z_2, X_2	T_2	Max(T_m, 0)	T_a	$T_m + T_a$
5	Add	$Y_1Z_2^2$, Y_2	T_1	Max($T_m + T_s$, 0)	T_a	$T_m + T_s + T_a$
6	$\times$	Z_2, T_2	T_3	Max(0, $T_m + T_a$)	T_m	$2T_m + T_a$
7	X^2	T_2	T_2^2	$T_m + T_a$	T_s	$T_m + T_s + T_a$
8	$\bigoplus$	T_3, Z_2^2	$T_3 + Z_2^2$	Max($2T_m + T_a$, T_s)	T_a	$2T_m + 2T_a$
9	X^2	T_1	T_1^2	$T_m + T_s + T_a$	T_s	$T_m + 2T_s + T_a$
10	$\times$	T_1, T_3	T_5	Max($T_m + T_s + T_a$, $2T_m + T_a$)	T_m	$3T_m + T_a$
11	X^2	T_3	$Z_3 = T_3^2$	$2T_m + T_a$	T_s	$2T_m + T_s + T_a$
12	$\times$	T_2^2, $T_3 + Z_2^2$	T_4	Max($T_m + T_s + T_a$, $2T_m + 2T_a$)	T_m	$3T_m + 2T_a$
13	$\bigoplus$	T_1^2, T_5	$T_1^2 + T_5$	Max($T_m + 2T_s + T_a$, $3T_m + T_a$)	T_a	$3T_m + 2T_a$
14	$\times$	X_2, Z_3	X_2Z_3	Max(0, $2T_m + T_s + T_a$)	T_m	$3T_m + T_s + T_a$
15	$\times$	Y_2, Z_3	Y_2Z_3	Max(0, $2T_m + T_s + T_a$)	T_m	$3T_m + T_s + T_a$
16	$\bigoplus$	$T_1^2 + T_5$, T_4	X_3	Max($3T_m + 2T_a$, $3T_m + 2T_a$)	T_a	$3T_m + 3T_a$
17	$\bigoplus$	X_3 , X_2Z_3	T_6	Max($3T_m + 3T_a$, $3T_m + T_s + T_a$)	T_a	$3T_m + T_s + 2T_a$
18	$\bigoplus$	X_3,Y_2Z_3	T_7	Max($3T_m + 3T_a$, $3T_m + T_s + T_a$)	T_a	$3T_m + T_s + 2T_a$
19	$\times$	T_5,T_6	T_5T_6	Max($3T_m + T_a$, $3T_m + T_s + 2T_a$)	T_m	$4T_m + T_s + 2T_a$
20	$\times$	Z_3,T_7	Z_3T_7	Max($2T_m + T_s + T_a$, $3T_m + T_s + 2T_a$)	T_m	$4T_m + T_s + 2T_a$
21	$\bigoplus$	T_5T_6,Z_3T_7	Y_3	Max($4T_m + T_s + 2T_a$, $4T_m + T_s + 2T_a$)	T_a	$4T_m + T_s + 3T_a$

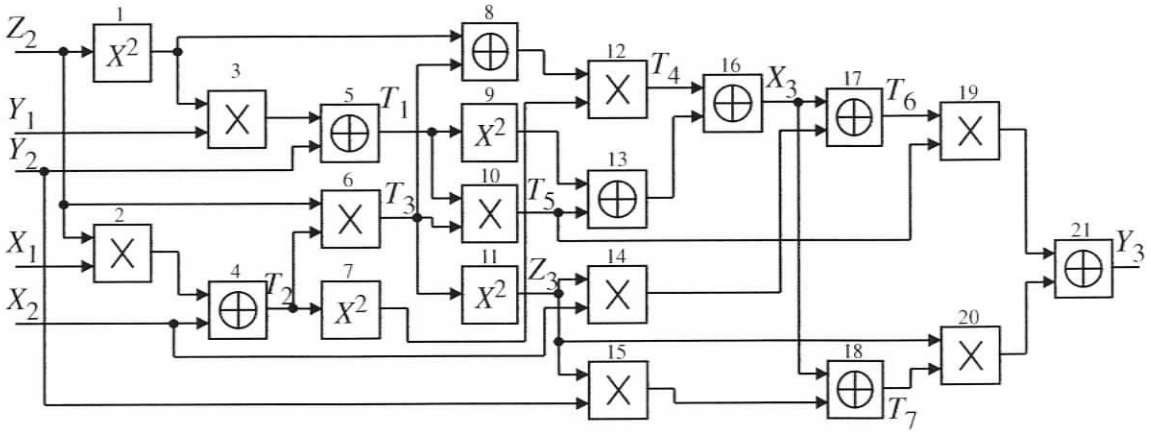


Figure 6.4: Implementation of López-Dahab point addition, $\bigoplus$: adder in $GF(2^m)$, $\times$: multiplier in $GF(2^m)$, X^2: squarer in $GF(2^m)$.

components are the output delays of the point doubling circuit.

6.4.2 Radix-4 Implementation of *scalar* multiplication

Radix-4 *scalar* multiplication requires performing $4Q+iP$ operations. The traditional technique is to perform point doubling ($2Q$) twice followed by a point addition. which in turn requires an area of $17A_m+14A_s+16A_a$ and a critical path delay of $8T_m+5T_s+7T_a$. On the other hand, combining these three steps in one architecture saves some time at the expense of consuming more area. Figure 6.6 illustrates these two different approaches. Using the same approach as in radix-2 (Section 6.4.1), the combined $4Q + iP$ architecture requires an area of $17A_m + 14A_s + 16A_a$ and a critical path delay of $6T_m + 3T_s + 4T_a$.

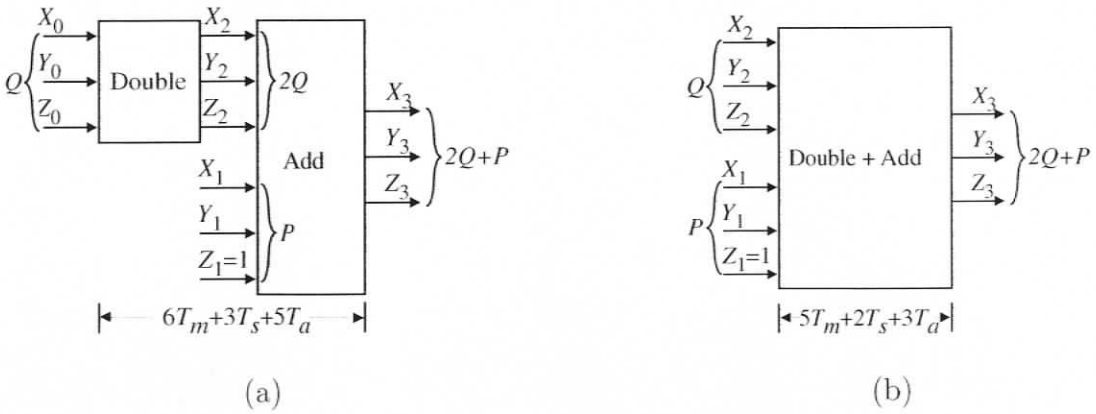


Figure 6.5: Different implementation approaches for calculating $2Q+P$: (a) Separate implementation (b) Combined implementation.

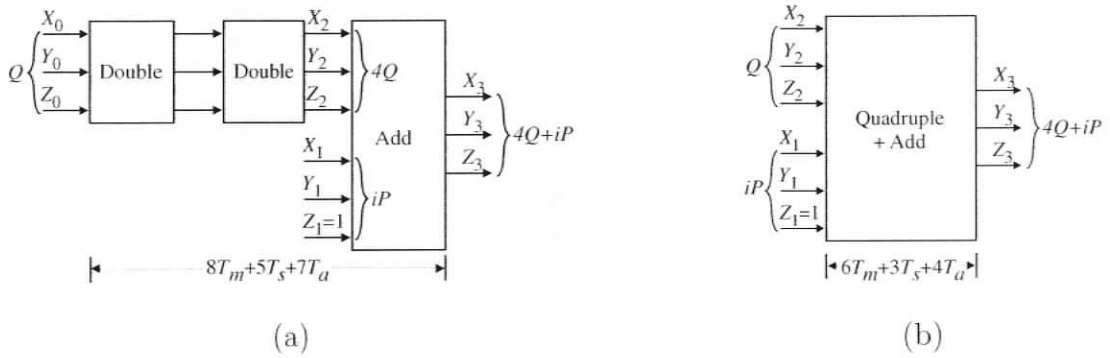


Figure 6.6: Different implementation approaches for calculating $4Q+iP$: (a) Separate implementation (b) Combined implementation.

Table 6.3: Critical path delay calculations for $2Q + P$ operation, $\bigoplus$: adder in $GF(2^m)$, $\times$: multiplier in $GF(2^m)$, X^2: squarer in $GF(2^m)$, T_m: multiplication delay, T_s: squaring delay, T_a: addition delay.

Location	Function	Inputs	Output	Input delay	Node delay	Output delay
1	X^2	Z_2	Z_2^2	$T_m + T_s$	T_s	$T_m + 2T_s$
2	$\times$	X_1, Z_2	X_1Z_2	$\text{Max}(0, T_m + T_s)$	T_m	$2T_m + T_s$
3	$\times$	Y_1, Z_2^2	$Y_1Z_2^2$	$\text{Max}(0, T_m + 2T_s)$	T_m	$2T_m + 2T_s$
4	Add	X_1Z_2, X_2	T_2	$\text{Max}(2T_m + T_s, T_m + 2T_s + T_a)$	T_a	$2T_m + T_s + T_a$
5	Add	$Y_1Z_2^2$, Y_2	T_1	$\text{Max}(2T_m + 2T_s, 2T_m + 2T_s + 2T_a)$	T_a	$2T_m + 2T_s + 3T_a$
6	$\times$	Z_2,T_2	T_3	$\text{Max}(T_m + T_s, 2T_m + T_s + T_a)$	T_m	$3T_m + T_s + T_a$
7	X^2	T_2	T_2^2	$2T_m + T_s + T_a$	T_s	$2T_m + 2T_s + T_a$
8	$\bigoplus$	T_3, Z_2^2	$T_3 + Z_2^2$	$\text{Max}(3T_m + T_s + T_a, T_m + 2T_s)$	T_a	$3T_m + T_s + 2T_a$
9	X^2	T_1	T_1^2	$2T_m + 2T_s + 3T_a$	T_s	$2T_m + 3T_s + 3T_a$
10	$\times$	T_1, T_3	T_5	$\text{Max}(2T_m + 2T_s + 3T_a, 3T_m + T_s + T_a)$	T_m	$4T_m + T_s + T_a$
11	X^2	T_3	$Z_3 = T_3^2$	$3T_m + T_s + T_a$	T_s	$3T_m + 2T_s + T_a$
12	$\times$	T_2^2, $T_3 + Z_2^2$	T_4	$\text{Max}(2T_m + 2T_s + T_a, 3T_m + T_s + 2T_a)$	T_m	$4T_m + T_s + 2T_a$
13	$\bigoplus$	T_1^2, T_5	$T_1^2 + T_5$	$\text{Max}(2T_m + 3T_s + 3T_a, 4T_m + T_s + T_a)$	T_a	$4T_m + T_s + 2T_a$
14	$\times$	X_2, Z_3	X_2Z_3	$\text{Max}(T_m + 2T_s + T_a, 3T_m + 2T_s + T_a)$	T_m	$4T_m + 2T_s + T_a$
15	$\times$	Y_2, Z_3	Y_2Z_3	$\text{Max}(2T_m + 2T_s + 2T_a, 3T_m + 2T_s + T_a)$	T_m	$4T_m + 2T_s + T_a$
16	$\bigoplus$	$T_1^2 + T_5$, T_4	X_3	$\text{Max}(4T_m + T_s + 2T_a, 4T_m + T_s + 2T_a)$	T_a	$4T_m + T_s + 3T_a$
17	$\bigoplus$	X_3 , X_2Z_3	T_6	$\text{Max}(4T_m + T_s + 3T_a, 4T_m + 2T_s + T_a)$	T_a	$4T_m + 2T_s + 2T_a$
18	$\bigoplus$	X_3 , Y_2Z_3	T_7	$\text{Max}(4T_m + T_s + 3T_a, 4T_m + 2T_s + T_a)$	T_a	$4T_m + 2T_s + 2T_a$
19	$\times$	T_5, T_6	T_5T_6	$\text{Max}(4T_m + T_s + T_a, 4T_m + 2T_s + 2T_a)$	T_m	$5T_m + 2T_s + 2T_a$
20	$\times$	Z_3, T_7	Z_3T_7	$\text{Max}(3T_m + 2T_s + T_a, 4T_m + 2T_s + 2T_a)$	T_m	$5T_m + 2T_s + 2T_a$
21	$\bigoplus$	T_5T_6 , Z_3T_7	Y_3	$\text{Max}(5T_m + 2T_s + 2T_a, 5T_m + 2T_s + 2T_a)$	T_a	$5T_m + 2T_s + 3T_a$

6.4.3 High-Radix Implementation of *scalar* multiplication

To show the advantages of our approach in combining point doublings and adding in the same architecture, we provide the critical path delays and the areas for different w values. Following the same approach in calculations as in Section 6.4.1, Table 6.4 lists the critical path delays and the area requirements for $w \in \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Table 6.4 shows that as w increases, the number of clock cycles required to calculate the *scalar* multiplication T_{SM} decreases. However, the area increases dramatically. To reduce the area, we use the pipelining approach, but the required field operations (i.e., addition, squaring, and multiplication) are not identical.

In our work [48] we proposed a high speed, low area processor array field ALU. The proposed field ALU implements field addition, squaring, and multiplication, which are required to perform point doubling and adding operations. By utilizing the proposed field ALU along with pipelining, the required area could be reduced. We noticed from Figure 6.4 that at a given time the maximum required field operations are 4. Therefore, we use four instances of the proposed field ALU to implement an elliptic curve ALU. Figure 6.7 (a) shows the elliptic curve ALU in detail using the field ALU as proposed in [48]. The details of each field ALU is shown in Figure 6.7 (b).

According to [48], the number of clock cycles required to perform field addition (T_a), squaring (T_s), and multiplication (T_m) are

$$\begin{aligned} T_a &= 3\lceil \tfrac{m}{W} \rceil + 1 \\ T_s &= 2\lceil \tfrac{m}{W} \rceil + d + 4 \\ T_m &= 3\lceil \tfrac{m}{W} \rceil + m \end{aligned} \tag{6.16}$$

where W is the field ALU external data bus width, and d is the right hand side maximum power of x in the reduction polynomial $R(x)$ after rewriting it in the form $x^m = x^d + \ldots + 1$ (e.g. $d = 7$ for $GF(2^{163})$) See Table 5.1 for the values of d. For the elliptic curve ALU, all internal data buses are m bits wide. Therefore, the field ALU

Table 6.4: Area and critical path delay for high-radix point doubling and adding, $0 \leq i < 2^w$.

w (bits)	Operation	LUT size (bits)	Area	Separate Arch. Delay (clock cycles)	Combined Arch. Delay T_{ALU} (clock cycles)
1	$2Q + P$	$6m$	$13A_m + 9A_s + 12A_a$	$6T_m + 3T_s + 5T_a$	$5T_m + 2T_s + 3T_a$
2	$4Q + iP$	$12m$	$17A_m + 14A_s + 16A_a$	$8T_m + 5T_s + 7T_a$	$6T_m + 3T_s + 4T_a$
3	$8Q + iP$	$24m$	$21A_m + 19A_s + 20A_a$	$10T_m + 7T_s + 9T_a$	$7T_m + 4T_s + 4T_a$
4	$16Q + iP$	$48m$	$25A_m + 24A_s + 24A_a$	$12T_m + 9T_s + 11T_a$	$8T_m + 6T_s + 5T_a$
5	$32Q + iP$	$96m$	$29A_m + 29A_s + 28A_a$	$14T_m + 11T_s + 13T_a$	$9T_m + 7T_s + 5T_a$
6	$64Q + iP$	$192m$	$33A_m + 34A_s + 32A_a$	$16T_m + 13T_s + 15T_a$	$10T_m + 9T_s + 6T_a$
7	$128Q + iP$	$284m$	$37A_m + 39A_s + 36A_a$	$18T_m + 15T_s + 17T_a$	$11T_m + 10T_s + 6T_a$
8	$256Q + iP$	$768m$	$41A_m + 44A_s + 40A_a$	$20T_m + 17T_s + 19T_a$	$12T_m + 11T_s + 7T_a$

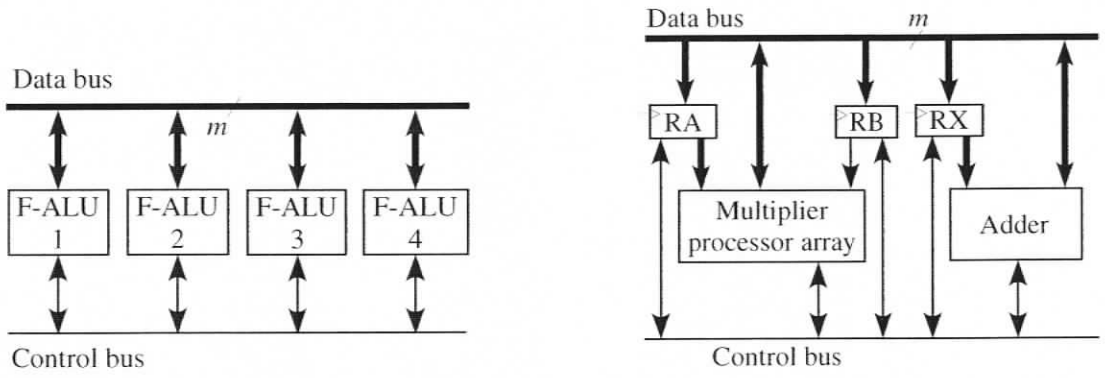


Figure 6.7: Detailed architecture: (a) Elliptic curve ALU (b) Field ALU.

has an external data bus of $W = m$. Thus we can write

$$\begin{aligned} T_a &= 4 \\ T_s &= d + 6 \\ T_m &= m + 3 \end{aligned} \tag{6.17}$$

Using Equation (6.17), the number of clock cycles required to perform point doubling and adding can be written as

$$\begin{aligned} T_{P_D} &= 2T_m + 2T_s + 2T_a = 2m + 2d + 26 \\ T_{P_A} &= 4T_m + T_s + 3T_a = 4m + d + 30 \end{aligned} \tag{6.18}$$

Similarly, the number of clock cycles required to perform $2^w Q + iP$ (T_{ALU}) can be

written as

$$T_{ALU} = \begin{cases} 5m + 2d + 39, & w = 1 \\ 6m + 3d + 52, & w = 2 \\ 7m + 4d + 61, & w = 3 \\ 8m + 6d + 80, & w = 4 \\ 9m + 7d + 89, & w = 5 \\ 10m + 7d + 96, & w = 6 \\ 11m + 10d + 117, & w = 7 \\ 12m + 11d + 130, & w = 8 \end{cases} \tag{6.19}$$

To compare our method to other methods, we calculate the number of clock cycles required to perform the *scalar* multiplication kP using binary, NAF, and w-NAF methods. For the sake of comparison, the number of clock cycles required to perform the *scalar* multiplication calculations using our method is calculated without taking register loading and unloading into consideration, since this is the case for other methods. Table 6.5 compares these methods to our method regarding the number of clock cycles required to perform the *scalar* multiplication calculations. The length l of the *scalar* value k is very crucial in performing the *scalar* multiplication. Table 6.5 shows that, as l increases the number of clock cycles required to calculate the *scalar* multiplication (T_{SM}) increases.

6.5 Implementation Results and Comparison

This section introduces our synthesis approach and the numerical results of our implementation.

Table 6.5: Number of clock cycles required to perform the *scalar* multiplication operation using binary, NAF, w-NAF, and our proposed method (MSWA).

w	T_{SM}								
	NAF			w-**NAF**			**MSWA**		
(bits)	163	283	571	163	283	571	163	283	571
1	596 l	1,008 l	1,963 l	711 l+22	1,203 l+29	2,350 l+26	868 l	1,478 l	2,914 l
2	-	-	-	596 l+366	1,008 l+616	1,963 l+1,188	1,051$\lceil \frac{l}{2} \rceil$	1,786$\lceil \frac{l}{2} \rceil$	3,508$\lceil \frac{l}{2} \rceil$
3	-	-	-	539 l+1,055	910l+1,790	1,769 l+3,512	1,230$\lceil \frac{l}{3} \rceil$	2,090$\lceil \frac{l}{3} \rceil$	4,098$\lceil \frac{l}{3} \rceil$
4	-	-	-	504l+2,433	851l+4,138	1,653 l+8,160	1,426$\lceil \frac{l}{4} \rceil$	2,416$\lceil \frac{l}{4} \rceil$	4,708$\lceil \frac{l}{4} \rceil$
5	-	-	-	481 l+5,189	812 l+8,834	1,576 l+17,456	1,605$\lceil \frac{l}{5} \rceil$	2,720$\lceil \frac{l}{5} \rceil$	5,298$\lceil \frac{l}{5} \rceil$
6	-	-	-	465 l+10,701	784 l+18,226	1,520 l+36,048	1,775$\lceil \frac{l}{6} \rceil$	3,010$\lceil \frac{l}{6} \rceil$	5,876$\lceil \frac{l}{6} \rceil$
7	-	-	-	452 l+21,725	763 l+37,010	1,479 l+73,232	1,980$\lceil \frac{l}{7} \rceil$	3,350$\lceil \frac{l}{7} \rceil$	6,498$\lceil \frac{l}{7} \rceil$
8	-	-	-	443 l+43,773	746 l+74,578	1,446 l+147,600	2,163$\lceil \frac{l}{8} \rceil$	3,658$\lceil \frac{l}{8} \rceil$	7,092$\lceil \frac{l}{8} \rceil$

6.5.1 Experimental Setup

To evaluate the proposed *scalar* multiplication architecture, we implemented it for three different m sizes as recommended by NIST ($m \in \{163, 283, 571\}$).

We modeled these three different implementations using VHDL and implemented them on an FPGA kit. Xilinx ISE 9.1 [189] is used to synthesize the design and provide post-placement area and timing results. The architecture is implemented on a Xilinx XC4VFX100-12 FPGA device.

6.5.2 Results and Comparison

The FPGA synthesis results for different m sizes are listed in Table 6.6. Table 6.6 lists the *scalar* multiplication delay measured in microsecond (μs) for various values of W and l. Table 6.6 shows that the bus width W has a minor effect on the total delay, since the value $\lceil \frac{l}{W} \rceil + 6\lceil \frac{m}{W} \rceil$ is too small in comparison to $\lceil \frac{l}{w} \rceil (T_{ALU} + 6)$. Moreover, as l increases, the *scalar* multiplication delay increases. On the other hand, the *scalar* multiplication delay is inversely proportional to the w. The *scalar* multiplication delay is directly proportional to the amount $\lceil l/w \rceil$. The minimum delays occurred when l/w is an integer (i.e., $l/w \in \{1, 2, 3, \ldots\}$). Figure 6.8 (a)-(c) illustrates the relation between w and the *scalar* multiplication delay. Figure 6.8 (d) illustrates the relationship between w and the LUT size. The size of the LUT required to store the pre-computed values of iP increases with increased values of w.

Table 6.7 compares radix-2^8 implementation of our architecture to other selected designs. Since previous implementations have different m lengths, it is hard to conduct a precise comparison with our architecture. However, we prove that we achieved better results. For example, our implementation for $m = 163$ is 22 times faster than the architecture implemented for $m = 113$ proposed in [146], and 41 times faster than the architecture implemented for $m = 113$ proposed in [190]. Although

Table 6.6: Performance results of our proposed architectures implementations.

m (bits)	W (bits)	l (bits)	**Area** (slice)	f_{Max} (MHz)	T_{SM} (μs) w 1	2	3	4	5	6	7	8
163	32	8	3,568	253	28	17	15	11	13	14	16	9
		16			55	34	29	23	26	21	24	17
		32			111	67	54	45	45	42	39	34
163	64	8	3,568	253	28	17	15	11	13	14	16	9
		16			55	33	29	23	26	21	24	17
		32			111	67	54	45	45	42	39	34
163	128	8	3,568	253	28	17	15	11	13	14	16	9
		16			55	33	29	23	26	21	24	17
		32			111	67	54	45	45	42	39	34
283	32	8	6,128	157	76	46	40	31	35	39	43	24
		16			151	91	80	62	70	58	64	47
		32			302	182	147	123	122	115	107	94
283	64	8	6,128	157	75	46	40	31	35	39	43	23
		16			151	91	80	62	69	58	64	47
		32			301	182	147	123	121	115	107	93
283	128	8	6,128	157	75	46	40	31	35	38	43	23
		16			151	91	80	62	69	58	64	47
		32			301	182	147	123	121	115	107	93
571	32	8	12,894	127	184	111	98	75	84	93	103	57
		16			368	222	194	149	168	140	154	112
		32			734	442	355	297	293	278	256	224
571	64	8	12,894	127	184	111	97	75	84	93	103	56
		16			367	221	194	149	167	139	154	112
		32			734	442	355	297	292	278	256	224
571	128	8	12,894	127	184	111	97	74	84	93	102	56
		16			367	221	194	148	167	139	154	112
		32			734	442	355	297	292	278	256	223

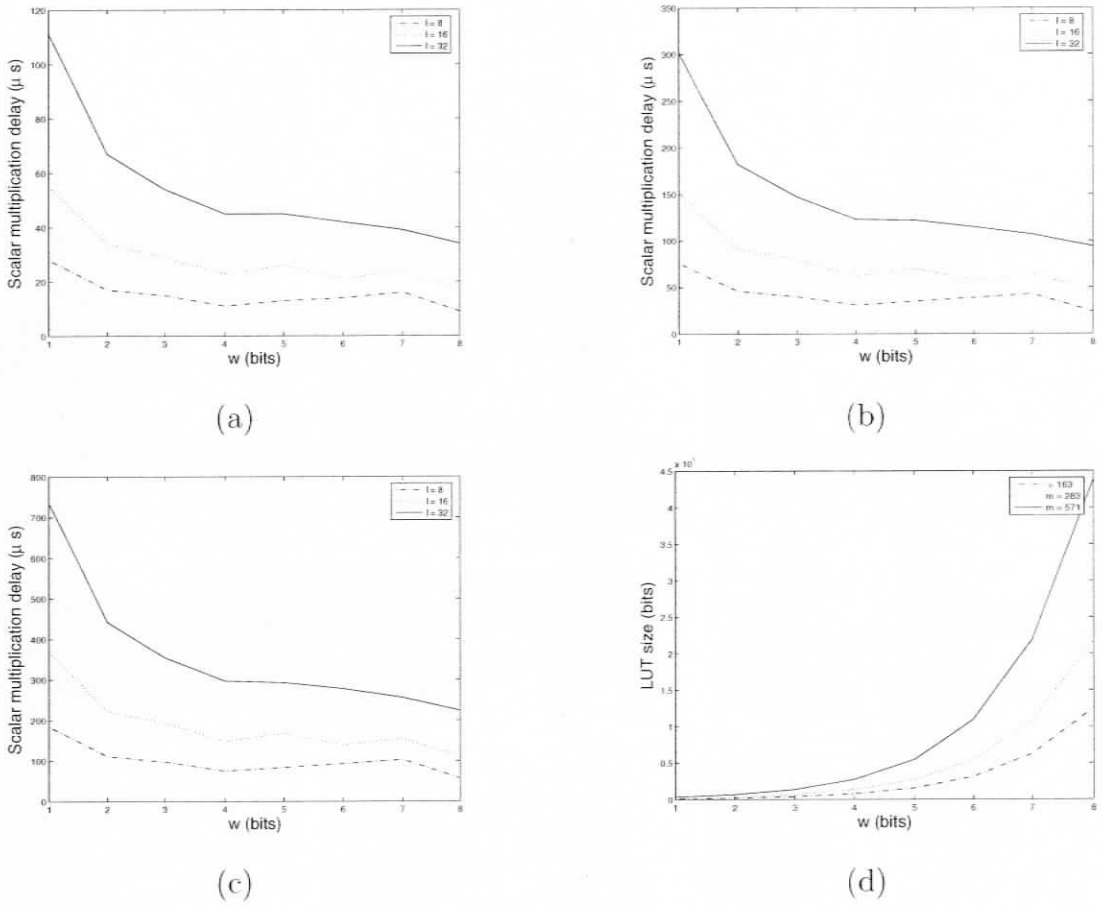


Figure 6.8: Effect of increasing w (a) $m = 163$, (b) $m = 283$, (c) $m = 571$, (d) LUT size.

ASIC implementations are traditionally much faster than FPGA implementations, we achieved speed up of 1.1, 5.6, and 6.1 in comparison to the ASIC implementations in [174], [165], and [175], respectively. Table 6.8 shows the speed up of the proposed architecture in comparison to the previous implementations. It is clear from Tables 6.7 and 6.8 that our architecture is faster than all prior FPGA and ASIC implementations. Although the *scalar* multiplication delay could be reduced by using a higher radix, the size of the LUT required to store the pre-computed values of iP will increase, which in turn requires more memory. Therefore, a compromise between area and speed should be taken into consideration depending on the required application.

6.6 Chapter Summary

In this chapter, a novel, area-speed efficient, high-speed ECC architecture over $GF(2^m)$ has been proposed. The critical path delay of the *scalar* multiplication has been reduced by using a high-radix implementation, based on a newly proposed algorithm and an optimized processor array field ALU. The proposed architecture utilizes 4 parallel field ALUs in order to speed up the *scalar* multiplication operation. The proposed architecture is implemented for $GF(2^{163})$, $GF(2^{283})$, and $GF(2^{571})$ on a Xilinx XC4VFX100-12 device to verify its function and measure its performance. We achieved a maximum frequency of 253 MHz, which allows the architecture to calculate $GF(2^{163})$ *scalar* multiplication for radix-2^8 in 9 μs. Although we compared our worst case results with the previous implementations, the proposed architecture for $GF(2^{163})$ achieves a speed up range from 1.5 to 326 in comparison to the previous FPGA implementations, and a speed up range from 1.1 to 5.6 in comparison to the previous ASIC implementations. The architecture could achieve further speed up by increasing the radix at the expense of larger LUTs.

In the next chapter, we conclude the dissertation and restate the contributions

Table 6.7: Comparison between our implementations and previous implementations.

Design	Technology	Area (slice/gates)	f_{Max} (MHz)	m (bits)	Time (μs)	**Our architecture**			
						m (bits)	f_{Max} (MHz)	Time (μs)	Speed up
Leong [146] (serial)	XCV1000	2,148 Slice	28	173	11,100	253	163	34	326
Leong [146] (parallel)	XCV1000	8,753 Slice	31	113	750	253	163	34	22
Gura [179]	XCV2000E	15,768 Slice	66.4	163	140	253	163	34	4
Ernst [190]	Atmel AT94K40	-	12	113	1,400	253	163	34	41
Bednara [180] (LFSR)	XCV1000	-	50	191	2,270	157	283	94	24
Bednara [180] (parallel)	XCV1000	-	50	191	270	157	283	94	2.9
Nguyen [182]	XC2V6000	-	100	233	3,350	157	283	94	35.6
Satoh [165]	0.13- μm ASIC	117.5 Kgates	510.2	160	190	253	163	34	5.6
Lutz [181]	XCV2000E	5,009 Slice	66	163	75	253	163	34	2.2
Mentens [184]	XCV800	-	47	160	3,810	253	163	34	112
Batina [185]	XCV800	11,881	47.7	211	763	157	283	94	8.1
Sozzani [175]	0.13- μm ASIC	90 Kgates	416.7	163	209	253	163	34	6.1
Cheung [147]	XC4VFX20-11	-	-	163	50	253	163	34	1.5
Sakiyama et al. [174]	0.13-μm ASIC	244 Kgates	292	163	76	253	163	34	2.2
Sakiyama et al. [174]	0.13-μm ASIC	244 Kgates	292	283	202	157	283	94	2.1
Sakiyama [174]	0.13-μm ASIC	244 Kgates	292	571	1,545	127	571	224	6.9
Sakiyama [174]	0.13-μm ASIC	115 Kgates	500	163	38	253	163	34	1.1

Table 6.8: Speed up of the proposed architecture in comparison to previous implementations.

m (bits)	**Speed up**			
	FPGA		**ASIC**	
	Minimum	Maximum	Minimum	Maximum
163	1.5	326	1.1	5.6
283	2.9	35.6	2.1	2.1
571	-	-	6.9	6.9

of this dissertation towards improving spam control.

Chapter 7

Conclusions and Future Work

Securing VoIP is a challenge since VoIP users are increasing every moment. We presented four approaches to improve VoIP security based on hardware acceleration and optimization. We do not claim that our proposed approaches will secure VoIP traffic completely. However, our approaches can complement existing efforts to secure VoIP. We could not cover all possible development scopes for effective VoIP security. Nonetheless, we hope that our proposed approaches to secure VoIP could lead to a better secure VoIP.

7.1 Summary

We summarize research work presented in this dissertation below.

In the first part of the research work, presented in Chapter 3, we proposed a reconfigurable, high throughput hardware implementation for the different block cipher operational modes.

In the second part of the research work, presented in Chapter 4, we proposed a high speed, deep-pipelined architecture for AES algorithm based on the composite

field approach targeting VoIP applications.

In the third part of the research work, presented in Chapter 5, we proposed a high-speed, low-area ALU to perform field operations required for cryptographic applications.

In the fourth part of the research work, presented in Chapter 6, we proposed a high-speed ECC architecture based on a high-radix *scalar* multiplication targeting VoIP applications.

7.2 Contributions

The contribution of this research work is summarized below.

7.2.1 A Unified, Reconfigurable Architecture Implementing Block Cipher Operational Modes

For the first contribution, we proposed a reconfigurable, high throughput hardware implementation for the different block cipher operational modes. The proposed architecture is unified; it combines multiple related functions on the same architecture. In other words, it has the ability to encrypt/decrypt a plaintext/ciphertext efficiently using ECB, CBC, CFB, OFB, and CTR operational modes. Moreover, it has the ability to prove data integrity using CBC-MAC, CCM, or RMAC operational modes. We have demonstrated that using the unified architecture saves a lot of area at the expenses of minor delay. This work has been published in brief in [37], and is submitted in full to [38].

7.2.2 A High-Speed, Fully-Pipelined Architecture for Real-Time AES

For the second contribution, we proposed a high speed, deep-pipelined architecture for the AES algorithm based on the composite field approach targeting VoIP applications. As a result of using novel techniques, we achieved a processing throughput of $49.401 Gbps$, which is twice the speed of the fastest design introduced before in previous studies [34, 40–44]. Moreover, it is designed to allow concurrent encryption and decryption, which facilitates full duplex encryption/decryption for VoIP applications. This work has been published in brief in [45], and is submitted in full to [46].

7.2.3 A Unified, Memoryless Processor Array Architecture for Basic Field Operations over $GF(2^m)$

For the third contribution, we proposed a high-speed, low-area ALU to perform field operations required for cryptographic applications. Although the proposed architecture works for any cryptographic application, we targeted an ECC implementation for VoIP applications. We achieved a maximum frequency of $264 MHz$, which allows the architecture to calculate $GF(2^{163})$ multiplication in $640 ns$ and inversion in $14.357 \mu s$. This work is submitted in brief to [47], and is submitted in full to [48].

7.2.4 A High-Speed, High-Radix, Processor Array Architecture for Real-Time Elliptic Curve Cryptography Over $GF(2^m)$

For the fourth contribution, we proposed a high speed ECC architecture based on a high-radix *scalar* multiplication targeting VoIP applications. We achieved a

maximum frequency of $253MHz$, which allows the architecture to calculate $GF(2^{163})$ *scalar* multiplication for radix 2^8 in $9\mu s$. Our worst case results for $GF(2^{163})$, showed a speed up that ranged from 1.5 to 326 in comparison to previous FPGA implementations and a speed up ranged from 1.1 to 5.6 in comparison to previous ASIC implementations. This work is submitted in brief to [49], and is submitted in full to [191].

7.3 Directions for Future Works

The work in this dissertation can not claim to solve the VoIP security problems completely. Nonetheless, securing VoIP with hardware support show promising potential. This research work can be extended along the following research directions.

7.3.1 VoIP Security Coprocessor Integration

The VoIP security processor requires that the security modules (i.e., encryption/decryption, message authentication, user authentication, and key management) be integrated together on the same architecture for further area and time saving.

7.3.2 Optimization for Hand Held Devices

As hand held communication devices becomes more popular, it is recommended to secure voice conversation transmitted between these devices. Since the hand held devices has limited power, the proposed hardware architectures need to be optimized for low power consumption to be used with hand held devices.

7.3.3 Investigate alternative approaches

As we stated in Section 2.2.4, securing VoIP increases the length of the VoIP packets by 11% to 70%, which contributes to degrading QoS. Therefore, investigating the idea of compressing the packet header to make it more efficient and consume less bandwidth needs to be done.

Bibliography

[1] "Voice over Internet protocols (VoIP) security technical implementation guide," Defence Information System Agency, Field security Operations, Gaithersburg, MD, USA, Jan. 2004, VoIP STIG, V1R1.

[2] W. C. Hardy, *VoIP Service Quality : Measuring and Evaluating Packet-Switched Voice*. New York, NY, USA: McGraw-Hill Professional, 2003.

[3] P. Drew and C. Gallon, "Next-generation VoIP network architecture," Mar. 2003, MSF-TR-ARCH-001. [Online]. Available: http://www.msforum.org/techinfo/reports/MSF-TR-ARCH-001-FINAL.pdf. Last accessed: June 2007.

[4] T. Walsh and D. Kuhn, "Challenges in securing voice over IP," *IEEE Security & Privacy Magazine*, vol. 3, no. 3, pp. 44–49, May 2005.

[5] S. Landau, "Security, wiretapping, and the Internet," *IEEE Security & Privacy Magazine*, vol. 3, no. 6, pp. 26–33, Nov. 2005.

[6] D. R. Kuhn, T. J. Walsh, and S. Fries, "Security consideration for voice over IP systems," National Institute for Standards and Technology (NIST), Gaithersburg, MD, USA, Jan. 2005, NIST SP 800-38A.

[7] K. Werbach, "Using VoIP to compete," Harvard Business Review, Sept. 2005.

[8] W. Stofega and T. S. Valovic, "U.S. residential VoIP services 2005-2009 forecast and analysis: Miles to go before we sleep," Mar. 2005, doc #32991.

[9] B. Alfonsi, "Alliance addresses VoIP security," *IEEE Security & Privacy Magazine*, vol. 3, no. 4, pp. 8–8, July 2005.

[10] M. Balaban, "VoIPong," EnderUNIX. [Online]. Available: http://www.enderunix.org/voipong/. Last accessed: June 2007.

[11] N. Provos, "VOMIT - voice over misconfigured Internet telephones." [Online]. Available: http://vomit.xtdnet.nl/. Last accessed: June 2007.

[12] F. Robles, "The VoIP dilemma GIAC security essentials practical assignment," SANS Institute, June 2004. [Online]. Available: http://www.sans.org/rr/whitepapers/voip/1452.php. Last accessed: June 2007.

[13] S. Kent and R. Atkinson, "Security architecture for the Internet protocol," Network Working Group, Nov. 1998, RFC 2401. [Online]. Available: http://www.ietf.org/rfc/rfc2401.txt. Last accessed: June 2007.

[14] R. Thayer, N. Doraswamy, and R. Glenn, "IP security document roadmap," Network Working Group, Nov. 1998, RFC 2411. [Online]. Available: http://www.ietf.org/rfc/rfc2411.txt. Last accessed: June 2007.

[15] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, "The secure real-time transport protocol (SRTP)," Network Working Group, Mar. 2004, RFC 3711. [Online]. Available: http://www.ietf.org/rfc/rfc3711.txt. Last accessed: June 2007.

[16] R. Barbieri, D. Bruschi, and E. Rosti, "Voice over IPSec: analysis and solutions," in *Proceedings of the* 18^{th} *Annual Computer Security Applications Conference (ACSAC)*, Dec. 9–13, 2002, pp. 261–270.

[17] O. Arkin, "Why E. T. can not phone home? security risk factors with IP telephony based networks," The Sys-Security Group, Nov. 2002. [Online]. Available: http://www.sys-security.com/archive/papers/Security_Risk_Factors_with_IP_Telephony_based_Networks.pdf. Last accessed: June 2007.

[18] ITU, "Recommendation ITU-T G.114 - one-way transmission time," International Telecommunication Union, Geneva, Feb. 1996.

[19] M. Collier, "Understanding delay in packet voice networks," Cisco Systems, Feb. 2006. [Online]. Available: http://www.cisco.com/warp/public/788/voip/delay-details.html. Last accessed: June 2007.

[20] B. Goode, "Voice over Internet protocol (VoIP)," *Proceedings of the IEEE*, vol. 90, no. 9, pp. 1495–1517, Sept. 2002.

[21] M. D. Collier, "Enterprise telecom security threats," SecureLogic, Oct. 2004. [Online]. Available: http://www.securelogix.com/. Last accessed: June 2007.

[22] Internet Security System, "VoIP: The evolving solution and the evolving threat," 2006. [Online]. Available: http://documents.iss.net/whitepapers/ISS_VoIP_White_paper.pdf. Last accessed: June 2007.

[23] R. Sinha, C. Spence, and T. Verrall, "Quality campus VoIP: An intel case study," *Intel Technology Journal*, vol. 10, no. 1, pp. 1495–1517, Feb. 2006.

[24] N. Shah, "Understanding network processors," Master's thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, USA, Sept. 2001.

[25] J. Daemen and V. Rijmen, "Advanced encryption standard (AES)," National Institute for Standards and Technology (NIST), Nov. 2001. [Online].

Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Last accessed: June 2007.

[26] IEEE, *IEEE Standard Specifications for Public-Key Cryptography*. New York, NY, USA: IEEE Computer Society, Aug. 2000.

[27] IEEE, *IEEE Standard Specifications for Public-Key Cryptography, Amendment 1: Additional Techniques*. New York, NY, USA: IEEE Computer Society, Sept. 2004.

[28] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, 1st ed. New York, NY, USA: Springer-Verlag, 2004.

[29] D. Hankerson, J. López, and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields," in *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems - (CHES'00)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 1965. Worcester, MA, USA: Springer-Verlag Berlin, Germany, Aug. 2000, pp. 1–24.

[30] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2006.

[31] "Data encryption standard (DES)," Oct. 1999. [Online]. Available: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf. Last accessed: June 2007.

[32] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC press, 1997.

[33] Xilinx, "Virtex-II platform FPGAs: Complete data sheet," Mar. 2005. [Online]. Available: http://www.xilinx.com/partinfo/ds031.pdf. Last accessed: June 2007.

[34] X. Zhang and K. Parhi, "High-speed VLSI architectures for the AES algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 9, pp. 957–967, 2004.

[35] S. Mangard, M. Aigner, and S. Dominikus, "A highly regular and scalable AES hardware architecture," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 483–491, 2003.

[36] A. Hodjat, P. Schaumont, and I. Verbauwhede, "Architectural design features of a programmable high throughput AES coprocessor," in *Proceedings of International Conference on Information Technology: Coding and Computing (ITCC'04)*, Apr. 5–7, 2004, pp. 498–502.

[37] M. Fayed, M. W. El-Kharashi, E. Khan, and F. Gebali, "A unified, configurable architecture implementing block cipher operational modes," in *Proceedings of the IEEE Third International Conference on Information and Communications Technology (ICICT'05)*, Cairo, Egypt, Dec. 4-6, 2005, pp. 883–895.

[38] M. Fayed, M. W. El-Kharashi, E. Khan, and F. Gebali, "A unified, configurable architecture implementing block cipher operational modes," *Submitted to IEEE Transaction on Consumer Electronics*, May 2006.

[39] C. Paar, "Efficient VLSI architecture for bit-parallel computation in Galois field," Ph. D. Dissertation, Institute of Experimental Mathematics, University of Essan, Essan, Germany, June 1994.

[40] K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä, "A fully pipelined memoryless 17.8 Gbps AES-128 encryptor," in *Proceedings of the* 11^{th} *international symposium on Field programmable gate arrays: (FPGA'03)*. New York, NY, USA: ACM Press, Feb. 23–25, 2003, pp. 207–215.

[41] A. Hodjat and I. Verbauwhede, "A 21.54 Gbits/s fully pipelined AES processor on FPGA," in 12^{th} *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, Apr. 20–23, 2004, pp. 308–309.

[42] G. P. Saggese, A. Mazzeo, N. Mazzocca, and A. G. M. Strollo, "An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm," in *Proceedings of the* 13^{th} *International Conference Field-Programmable Logic and Applications: (FPL'03)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2778. Lisbon, Portugal: Springer-Verlag Berlin, Germany, Sept. 1–3, 2003, pp. 292–302.

[43] F. Standaert, G. Rouvroy, J. Quisquater, and J. Legat, "Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs," in *Cryptographic Hardware and Embedded Systems - (CHES'03)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2779. Springer-Verlag Berlin, Germany, 2003, pp. 334–350.

[44] M. McLoone and J. V. McCanny, "Rijndael FPGA implementation utilizing look-up tables," in *IEEE Workshop on Signal Processing Systems*, London, UK, Sept. 26–28, 2001, pp. 349–360.

[45] M. Fayed, M. W. El-Kharashi, and F. Gebali, "A high-speed, fully-pipelined VLSI architecture for real-time AES." in *Proceedings of the ITI Fourth International Conference on Information and Communication Technology*

(ICICT'06), M. A. Salem and M. T. El-Hadidi, Eds., Cairo, Egypt, Dec. 10–12, 2006, pp. 429–448.

[46] M. Fayed, M. W. El-Kharashi, and F. Gebali, "A high-speed, fully-pipelined VLSI architecture for real time AES," *Submitted to IEEE Transaction on Circuits & Systems*, May 2006.

[47] M. Fayed, M. W. El-Kharashi, and F. Gebali, "A unified, memoryless processor array, architecture for basic field operations over $GF(2^m)$," *Submitted to the IEEE International Conference on Microelectronics (ICM'07)*, June 2007.

[48] M. Fayed, M. W. El-Kharashi, and F. Gebali, "A unified, memoryless processor array, architecture for basic field operations over $GF(2^m)$," *Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, June 2007.

[49] M. Fayed, M. W. El-Kharashi, and F. Gebali, "A high-speed, high-radix, processor array architecture for real-time elliptic curve cryptography over $GF(2^m)$," *Submitted to the IEEE Symposium on Signal Processing and Information Technology (ISSPIT'07)*, June 2007.

[50] Cisco, "IP telephony/voice over IP (VoIP) understanding packet voice protocols," Cisco Systems. [Online]. Available: http://www.cisco.com/. Last accessed: June 2007.

[51] Y. Boger, "Fine-tuning voice over packet services," RADCOM Ltd. [Online]. Available: http://www.protocols.com/pbook/pdf/voip.pdf. Last accessed: June 2007.

[52] R. Dhamankar, "Intrusion prevention: The future of VoIP security," Tipping Point a division of 3COM, June 2005. [Online]. Available:

http://www.crtgroupinc.com/docs/VoIPSecurity.pdf. Last accessed: June 2007.

[53] C. Metz, "Internet multimedia: Answering basic questions," *IEEE Internet Computing*, vol. 9, no. 4, pp. 51–55, July 2005.

[54] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session initiation protocol," Network Working Group, Mar. 1999, RFC 2543.

[55] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session initiation protocol," Network Working Group, June 2002, RFC 3261. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt. Last accessed: June 2007.

[56] R. Sparks, "Problems identified associated with the session initiation protocol's (SIP)," Network Working Group, Jan. 2006, RFC 4321. [Online]. Available: http://www.ietf.org/rfc/rfc4321.txt. Last accessed: June 2007.

[57] P. Rowe, "VoIP - extra threats in the converged environment," *Network Security*, vol. 2005, no. 7, pp. 12–16, July 2005.

[58] E. Edelson, "Voice over IP: Security pitfalls," *Network Security*, vol. 2005, no. 2, pp. 4–7, Feb. 2005.

[59] M. Collier, "Basic vulnerability issues for SIP security," SecueLogix Corporation, Mar. 2005. [Online]. Available: http://download.securelogix.com/library/SIP_Security030105.pdf. Last accessed: June 2007.

[60] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 2003.

[61] M. K. Ranganathan and L. Kilmartin, "Performance analysis of secure session initiation protocol based VoIP networks," *Computer Communications*, vol. 26, no. 6, pp. 552–565, Apr. 2003.

[62] A. Steffen, D. Kaufmann, and A. Stricker, "SIP security," in *DFN-Arbeitstagung über Kommunikationsnetze*, ser. LNI, vol. 55, 2004, pp. 397–412.

[63] P. Judge, "IPSec and SSL: Complementary VPN technologies for universal remote access," Nortel Networks, Sept. 2004. [Online]. Available: http://www.nortel.com/promotions/whitepaper/ssl/collateral/nn102260-091304.pdf. Last accessed: June 2007.

[64] Cisco, "An introduction to IP security (IPSec) encryption," Cisco, Mar. 2005. [Online]. Available: http://www.cisco.com/warp/public/105/IPSECpart1.pdf. Last accessed: June 2007.

[65] H. Schulzrinne, T. S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," Network Working Group, July 2003, RFC 3550. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3550.txt. Last accessed: June 2007.

[66] C. Huitema, "Real time control protocol (RTCP) attribute in session description protocol (SDP)," Network Working Group, Oct. 2003, RFC 3605. [Online]. Available: http://www.ietf.org/rfc/rfc3605.txt. Last accessed: June 2007.

[67] T. Friedman, R. Caceres, and A. Clark, "RTP control protocol extended reports (RTCP XR)." Network Working Group, Nov. 2003, RFC 3611. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3611.txt. Last accessed: June 2007.

[68] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," Network Working Group, Feb. 1997, RFC 2104. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt. Last accessed: June 2007.

[69] T. Uhl, "Quality of service in VoIP communication," *AEU - International Journal of Electronics and Communications*, vol. 58, no. 3, pp. 178–182, Mar. 2004.

[70] R. Wang and H. Xiaorui, "VoIP development in china," *Computer*, vol. 37, no. 9, Sept. 2004.

[71] T. Shimotoyodome, T. Kamiyama, Y. Mizukoshi, and H. Kumr, "Development of dedicated LSIs for VoIP telephones (ML7074/ML7084) and communications protocol control software." *OKI Technical Review Issue 196*, vol. 70, no. 4, Oct. 2003.

[72] T.-K. Chua and D. Pheanis, "Bandwidth-conserving real-time VoIP teleconference system," in *Proceedings of the Third International Conference on Information Technology: New Generations (ITNG'06)*, Apr. 10–12, 2006, pp. 535–540.

[73] C. Ho, T. Tang, C. Lee, C. Chen, H. Tu, C. Wu, C. Chang, and C. Huang, "H.323 VoIP telephone implementation embedding a low-power SOC processor," in *IEEE Conference on Electron Devices and Solid-State Circuits*, Dec. 16–18, 2003, pp. 163–166.

[74] A. Hashad, K. Shehata, M. Dahab, and K. El-Dars, "Design and implementation of a signaling unit for a simple IP phone," in *Proceedings of the* 46^{th} *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS'03)*, Dec. 27–30, 2003, pp. 851–845.

[75] D. Y. Kim and J. W. Park, "A design and implementation of ASIC for high-quality VoIP terminal over wireless LAN," in *The Eighth International Conference on Advanced Communication Technology (ICACT'06)*, Feb. 20–22, 2006, pp. 1645–1648.

[76] Cisco, "Voice and IP communications compare products and solutions," Cisco. [Online]. Available: http://www.cisco.com/. Last accessed: June 2007.

[77] Broadcom, "Ethernet IP phone solutions," BROADCOM. [Online]. Available: http://www.broadcom.com/products/Voice-Over-IP-(VoIP) /Ethernet-IP-Phone-Solutions. Last accessed: June 2007.

[78] M. Collier, "Voice over IP solutions intel IXP421 and IXP425 network processors," Intel. [Online]. Available: ftp://download.intel.com/design/ network/appbrf/25293901.pdf. Last accessed: June 2007.

[79] Intel, "Intel IXP423 network processor," Intel, 2005. [Online]. Available: ftp: //download.intel.com/design/network/ProdBrf/30566102.pdf. Last accessed: June 2007.

[80] Intel, "Intel IXP421 network processor," Intel, 2003. [Online]. Available: ftp: //download.intel.com/design/network/ProdBrf/25249502.pdf. Last accessed: June 2007.

[81] Intel, "Intel IXP425 network processor," Intel, 2004. [Online]. Available: ftp: //download.intel.com/design/network/ProdBrf/27905104.pdf. Last accessed: June 2007.

[82] G. Lisha and L. Junzhou, "Performance analysis of a P2P-based VoIP software," in *Proceedings of International Conference on Internet and Web Applications*

and Services/Advanced Telecommunications (AICT-ICIW'06), Feb. 19-25, 2006, pp. 11–16.

[83] P. Judge, "Cambridge prof warns of skype botnet threat," TechWorld, Jan. 2006. [Online]. Available: http://www.techworld.com/mobility/news/index.cfm?NewsID=5232. Last accessed: June 2007.

[84] S. Frankel, R. Glenn, and S. Kelly, "RFC 3602: The AES-CBC cipher algorithm and its use with IPSec," Sept. 2003. [Online]. Available: http://rfc3602.x42.com/. Last accessed: June 2007.

[85] R. Housley, "RFC 3686: Using advanced encryption standard (AES) counter mode with IPSec encapsulating security payload (ESP)," Jan. 2004. [Online]. Available: http://rfc3686.x42.com/. Last accessed: June 2007.

[86] R. Housley, "Using AES CCM mode with IPSec ESP," Nov. 2003. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ciph-aes-ccm-05.txt. Last accessed: June 2007.

[87] M. Dworkin, "Recommendation for block cipher modes of operation," National Institute for Standards and Technology, Gaithersburg, MD, USA, Dec. 2001, NIST SP 800-38A.

[88] M. Dworkin, "Recommendation for block cipher modes of operation: The RMAC authentication mode methodes and techniques," National Institute for Standards and Technology, Gaithersburg, MD, USA, Nov. 2001, NIST SP 800-38B.

[89] M. Dworkin, "Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality," National Institute for Standards and Technology, Gaithersburg, MD, USA, Sept. 2003, NIST SP 800-38C.

[90] E. Cole, R. Krutz, and J. W. Conley, *Network Security Bible.* Crosspoint Boulevard Indianapolis, IN, USA: Wiley Publishing, Inc., Jan. 2005.

[91] I. Papaefstathiou, V. Papaefstathiou, and C. Sotiriou, "Design-space exploration of the most widely used cryptography algorithms," *Microprocessors and Microsystems*, vol. 28, no. 10, pp. 561–571, 2004.

[92] F. Arts, P. Barri, I. Clemminck, A. Niemegeers, B. Pauwels, G. Taildeman, and M. Vrana, "Network processor requirements and benchmarking," *Computer Networks*, vol. 41, no. 5, pp. 549–562, 2003.

[93] J. Goodman and A. P. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1808–1820, 2001.

[94] J. Daemen and V. Rijmen, "AES proposal: Rijndael," Sept. 1999. [Online]. Available: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf. Last accessed: June 2007.

[95] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A compact Rijndael hardware architecture with s-box optimization," in *Advances in Cryptology - (ASIACRYPT'01): Proceedings of the Seventh International Conference on the Theory and Application of Cryptology and Information Security*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2248. Gold Coast, Australia: Springer-Verlag Berlin, Germany, Dec. 9–13, 2001, pp. 239–254.

[96] M. Macchetti and G. Bertoni, "Hardware implementation of the Rijndael sbox: a case study," *ST Journal of System Research*, pp. 84–91, July 2003.

[97] I. Verbauwhede, P. Schaumont, and H. Kuo, "Design and performance testing of a 2.29-Gb/s Rijndael processor," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 3, pp. 569–572, Mar. 2003.

[98] C. Lu and S. Tseng, "Integrated design of AES (advanced encryption standard) encryptor and decryptor," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, vol. 2, July 17–19, 2002, pp. 277–285.

[99] N. Sklavos and O. Koufopavlou, "Architectures and VLSI implementations of the AES-proposal Rijndael," *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1454–1459, 2002.

[100] N. M. Kosaraju, M. Varanasi, and S. P. Mohanty, "A high-performance VLSI architecture for advanced encryption standard (AES) algorithm," in *Proceedings of the* 19^{th} *International Conference on VLSI Design*, Jan. 3–7, 2006, pp. 481–484.

[101] M. H. Jing, Y. H. Chen, Y. T. Chang, and C. H. Hsu, "The design of a fast inverse module in AES," in *Proceedings of International Conferences on Info-tech and Info-net(ICII)*, vol. 3, Oct. 29–Nov. 1, 2001, pp. 298–303.

[102] A. V. Dinh, R. J. Bolton, and R. Mason, "A low latency architecture for computing multiplicative inverses and divisions in $GF(2^m)$," *IEEE Transactions on Circuits on Circuits Systems II: Analog & Digital Signal Processing*, vol. 48, no. 8, pp. 789–793, Aug. 2001.

[103] V. Rijmen, "Efficient implementation of the Rijndael s-box," 2000. [Online]. Available: http://homes.esat.kuleuven.be/~rijmen/rijndael/sbox.pdf. Last accessed: June 2007.

[104] C. Su, T. Lin, C. Huang, and C. Wu, "A high-throughput low-cost AES processor," *IEEE Communications Magazine*, vol. 41, no. 12, pp. 86–91, Dec. 2003.

[105] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, "Efficient Rijndael encryption implementation with composite field arithmetic," in *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2162. Paris, France: Springer-Verlag Berlin, Germany, May 14–16, 2001, pp. 171–184.

[106] C. O'Driscoll, "Hardware implementation aspects of the Rijndael block cipher," Master's thesis, National University of Ireland, Oct. 2001.

[107] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, "A systematic evaluation of compact hardware implementations for the Rijndael S-Box," in *Topics in Cryptology (CT-RSA'05): The Cryptographers Track at the RSA Conference*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 3376. San Francisco, CA, USA: Springer-Verlag Berlin, Germany, Feb. 14–18, 2005, pp. 323–333.

[108] P. Prasithsangaree and P. Krishnamurthy, "Analysis of tradeoffs between security strength and energy savings in security protocols for WLANs," in *IEEE* 60^{th} *on Vehicular Technology Conference (VTC'04)-Fall*, Sept. 26-29, 2004, pp. 5219–5223.

[109] M. Feldhofer, S. Dominikus1, and J. Wolkerstorfer, "Strong authentication for RFID systems using the AES algorithm," in *Proceedings of the Sixth International Workshop on Cryptographic Hardware and Embedded Systems (CHES'04)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol.

LNCS 3156. Cambridge, MA, USA: Springer-Verlag Berlin, Germany, Aug. 11–13, 2004, pp. 239–254.

[110] D. Greenstreet and S. Scoggins, "Building residential VoIP gateways: A tutorial, part four: VoIP security implementation." Dec. 2004, VoIP Business Unit, Texas Instruments Incorporation. [Online]. Available: www.analogzone.com/nett0913.pdf. Last accessed: June 2007.

[111] T. A. Gulliver, "Irreducible and primitive polynomial of degree 2 whose coefficients are elements of $GF(2^4)$ [personal communication]," Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada.

[112] P. Chodowiec, P. Khuon, and K. Gaj, "Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining," in *Proceedings of the 2001 ACM/SIGDA Ninth international symposium on Field programmable gate arrays: (FPGA'01)*. New York, NY, USA: ACM Press, 2001, pp. 94–102.

[113] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 4th ed. Englewood Cliffs, NJ: Prentice-Hall, 2005.

[114] NIST, "Digital signature standard," National Institute for Standards and Technology (NIST), Gaithersburg, MD, USA, Jan. 2000, FIPS PUB 186-2. [Online]. Available: http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf. Last accessed: June 2007.

[115] S. Kwon, C. H. Kim, and C. P. Hong, "Compact linear systolic arrays for multiplication using a trinomial basis in $GF(2^m)$ for high speed cryptographic processors," in *Proceedings of Computational Science and Its Applications (ICCSA'05)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 3480. Springer-Verlag Berlin, Germany, May 2005, pp. 508–518.

[116] A. Reyhani-Masoleh and M. A. Hasan, "Low complexity bit parallel architectures for polynomial basis multiplication over $GF(2^m)$," *IEEE Transaction on Computers*, vol. 53, no. 8, pp. 945–959, Aug. 2004.

[117] M. Hasan, M. Wang, and V. Bhargava, "A modified massey-omura parallel multiplier for a class of finite fields," *IEEE Transaction on Computers*, vol. 42, no. 10, pp. 1278–1280, Oct. 1993.

[118] T. Zhang and K. K. Parhi, "Systematic design of original and modified mastrovito multipliers for general irreducible polynomials," *IEEE Transaction on Computers*, vol. 50, no. 7, pp. 734–749, July 2001.

[119] A. Halbutogullar and C. K. Koc, "Mastrovito multiplier for general irreducible polynomials," in *Proceedings of the* 13^{th} *International Symposium on Algebraic Algorithms and Error-Correcting Codes: AAECC-13*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 1719. Honolulu, Hawaii, USA: Springer-Verlag Berlin, Germany, Nov. 1999, pp. 498–507.

[120] E. D. Mastrovito, "VLSI designs for multiplication over finite fields $GF(2^m)$," in *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 357. Springer-Verlag Berlin, Germany, 1989, pp. 297–309.

[121] C. Wu and M. Chang, "Bit-level systolic arrays for finite-field multiplications," *The Journal of VLSI Signal Processing*, vol. 10, no. 1, pp. 85–92, June 1995.

[122] C. Lee, Y. Chen, C. Chiou, and J. Lin, "Unified parallel systolic multiplier over $GF(2^m)$," *Journal of Computer Science and Technology*, vol. 22, no. 1, pp. 28–38, Jan. 2007.

[123] J. Guo and C. Wang, "Digit-serial systolic multiplier for finite fields $GF(2^m)$," *IEE Proceedings on Computers & Digital Techniques*, vol. 145, no. 2, pp. 143–148, Mar. 1998.

[124] M. Mekhallalati, M. Ibrahim, and A. Ashur, "New low complexity bidirectional systolic structures for serial multiplication over the finite field $GF(2^m)$," *IEE Proceedings on Circuits, Devices & Systems*, vol. 145, no. 1, pp. 55–60, Feb. 1998.

[125] C. Lee, E. Lu, and J. Lee, "Bit-parallel systolic multipliers for $GF(2^m)$ fields defined by all-one and equally spaced polynomials," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 385–393, May 2001.

[126] S. T. J. Fenn, M. Benaissa, and O. Taylor, "Dual basis systolic multipliers for $GF(2^m)$," *IEE Proceedings on Computers & Digital Techniques*, vol. 144, no. 1, pp. 43–46, Jan. 1997.

[127] C. Lee, C. W. Chiou, and J. Lin, "Concurrent error detection in a bit-parallel systolic multiplier for dual basis of $GF(2^m)$," *Journal of Electronic Testing*, vol. 21, no. 5, pp. 539–549, Oct. 2005.

[128] X. Wu, H. Chen, Y. Sun, and W. Gai, "A fully-pipeline linear systolic architecture for modular multiplier in public-key crypto-systems." *The Journal of VLSI Signal Processing*, vol. 33, no. 1-2, pp. 191–197, Oct. 2004.

[129] C. H. Kim, S. Kwon, C. P. Hong, and H. Kim, "A new systolic array for least significant digit first multiplication in $GF(2^m)$," in *Computational Science and Its Applications (ICCSA'04)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 3045. Springer-Verlag Berlin, Germany, Apr. 2004, pp. 656–666.

[130] G. Bertoni, J. Guajardo, and G. Orlando, "Systolic and scalable architectures for digit-serial multiplication in fields $GF(2^m)$," in *Computational Science and Its Applications (ICCSA'04)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2904. Springer-Verlag Berlin, Germany, Feb. 2003, pp. 349–362.

[131] S. Kwon, "Low complexity bit serial systolic multipliers over $GF(2^m)$ for three classes of finite fields," in *Proceedings of Information and Communications Security: Fourth International Conference, (ICICS'02)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2513. Singapore: Springer-Verlag Berlin, Germany, Dec. 2002, pp. 209–216.

[132] H.-S. Kim and I.-S. Jeon, "Semi-systolic architecture for modular multiplication over $GF(2^m)$," in *Proceedings of Computational Science (ICCS'05)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 3516. Springer-Verlag Berlin, Germany, May 2005, pp. 912–915.

[133] R. K. Satzoda and C.-H. Chang, "VLSI performance evaluation and analysis of systolic and semisystolic finite field multipliers," in *Proceedings of Advances in Computer Systems Architecture*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 3740. Springer-Verlag Berlin, Germany, Oct. 2005, pp. 693–706.

[134] C. Yeh, I. S. Reed, and T. K. Truong, "Systolic multipliers for finite fields $GF(2^m)$," *IEEE Transaction on Computers*, vol. C-33, no. 4, pp. 357–360, Apr. 1984.

[135] J. Guajardo and C. Paar, "Efficient algorithms for elliptic curve cryptosystems," in *Advances in Cryptology - (CRYPTO'97): 17^{th} Annual International Cryptology Conference*, ser. Lecture Notes in Computer Science, Springer,

Berlin, vol. LNCS 1294. Santa Barbara, California, USA: Springer-Verlag Berlin, Germany, Aug. 1997, pp. 342–356.

[136] M. Aydos, E. Savas, and C. K. Koc., "Implementing network security protocols based on elliptic curve cryptography," in *Proceedings of the Fourth Symposium on Computer Networks*, Istanbul, Turkey, May 20–21, 1999, pp. 130–139.

[137] M. C. Rosner, "Elliptic curve cryptosystems on reconfigurable hardware," Ph. D. Dissertation, Worcester Polytechnic Institute, Worcester, MA, USA, May 1998.

[138] J. Guo and C. Wang, "Systolic array implementation of euclid's algorithm for inversion and division in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1161–1167, Oct. 1998.

[139] C.-H. Wu, C.-M. Wu, M.-D. Shieh, and Y.-T. Hwang, "High-speed, low-complexity systolic designs of novel iterative division algorithms in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 375–380, Mar. 2004.

[140] J.-H. Guo and C.-L. Wang, "Hardware-efficient systolic architecture for inversion and division in $GF(2^m)$," *IEE Proceedings on Computers & Digital Techniques*, vol. 145, no. 3, pp. 272–278, July 1998.

[141] Z. Yan, D. V. Sarwate, and Z. Liu, "High-speed systolic architectures foe finite field inversion," *Integration, the VLSI Journal*, vol. 38, no. 3, pp. 383–398, Jan. 2005.

[142] Z. Yan and D. V. Sarwate, "New systolic architectures for inversion and division in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1514–1519, Nov. 2003.

[143] H. Brunner, A. Curiger, and M. Hofstetter, "On computing multiplicative inverses in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1010–1015, Aug. 1993.

[144] A. Daly, W. Marnane, T. Kerins, and E. Popovici, "Fast modular division for application in ECC on reconfigurable logic," in *Field-Programmable Logic and Applications*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2778. Springer-Verlag Berlin, Germany, Sept. 2003, pp. 786–795.

[145] E. Savas and C. K. Koc, "Architectures for unified field inversion with applications in elliptic curve cryptography," in *Proceedings of the Ninth International Conference on Electronics, Circuits and Systems*, vol. 3, Sept. 15–18, 2002, pp. 1155–1158.

[146] P. H. W. Leong and I. K. H. Leung, "A microcoded elliptic curve processor using FPGA technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 5, pp. 550–559, Oct. 2002.

[147] R. C. C. Cheung, N. J. Telle, W. Luk, and P. Y. K. Cheung, "Customizable elliptic curve cryptosystems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 9, pp. 1048–1059, Sept. 2005.

[148] S.-W. Wei, "A systolic power-sum circuit for $GF(2^m)$," *IEEE Transactions on Computers*, vol. 43, no. 2, pp. 226–229, Feb. 1994.

[149] S. K. Jain, L. Song, and K. K. Parhi, "Efficient semisystolic architectures for finite-field arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 1, pp. 101–113, Mar. 1998.

[150] C.-L. Wang and J.-H. Guo, "New systolic arrays for $C + AB^2$, inversion, and division in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 49, no. 10, pp. 1120–1125, Oct. 2000.

[151] H.-S. Kim and K.-Y. Yoo, "Area efficient exponentiation using modular multiplier/squarer in $GF(2^m)$," in *Proceedings of Computing and Combinatorics : Seventh Annual International Conference, (COCOON'01)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2108. Guilin, China: Springer-Verlag Berlin, Germany, Aug. 20–23, 2001, pp. 262–267.

[152] C. Y. Lee, E.-H. Lu, and L.-F. Sun, "Low-complexity bit-parallel systolic architecture for computing $AB^2 + C$ in a class of finite field $GF(2^m)$," *IEEE Transactions on Circuits & Systems II: Analog & Digital Signal Processing*, vol. 48, no. 5, pp. 519–523, May 2001.

[153] C.-Y. Lee, C. W. Chiou, A.-W. Deng, and J.-M. Lin, "Low-complexity bit-parallel systolic architectures for computing $A(x)B^2(x)$ over $GF(2^m)$," *IEE Proceedings on Circuits, Devices & Systems*, vol. 153, no. 4, pp. 399–406, Aug. 2006.

[154] J. Lutz and A. Hasan, "High performance elliptic curve cryptographic co-processor," Center for Applied Cryptographic Research, the University of Waterloo, Waterloo, Ontario, Canada, Tech. Rep. CACR 2004-06, 2004. [Online]. Available: http://www.cacr.math.uwaterloo.ca/techreports/2004/cacr2004-06.ps. Last accessed: June 2007.

[155] W.-Y. Lin, "Implementation of elliptic curve cryptography," Department of Mathematics, University of Wisconsin, Madison, WI, USA, Tech. Rep. ECE 842, Dec. 2004. [Online]. Available: http://www.math.wisc.edu/~boston/lin1.pdf. Last accessed: June 2007.

[156] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases," *Information and computing*, vol. 78, no. 3, pp. 171–177, 1998.

[157] F. ElGuibaly and A. Tawfik, "Mapping 3D IIR digital filter onto systolic array," *Multidimensional systems and Signal Processing*, vol. 7, no. 1, pp. 7–26, Jan. 1996.

[158] A. N. M. Refiq and F. Gebali, "Processor array architectures for deep packet classification." *IEEE Transaction on Parallel & Distributed Systems*, vol. 17, no. 3, pp. 241–252, Mar. 2006.

[159] N.-Y. Kim, H.-S. Kim, and K.-Y. Yoo, "Computation of $A(x)B^2(x)$ multiplication in $GF(2^m)$ using low-complexity systolic architecture," *IEE Proceedings Circuits, Devices & Systems*, vol. 150, no. 2, pp. 119–123, Apr. 2003.

[160] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A design perspective*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2002.

[161] M. Morales-Sandoval and C. Feregrino-Uribe, "On the hardware design of an elliptic curve cryptosystem," in *Proceedings of the Fifth Mexican International Conference in Computer Science (ENC'04)*, Sept. 20–24, 2004, pp. 64–70.

[162] S. B. Örs, L. Batina, B. Preneel, and J. Vandewalle, "Hardware implementation of a montgomery modular multiplier in a systolic array," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, Apr. 22–26, 2003.

[163] F. Crowe, A. Daly, and W. Marnane, "A scalable dual mode arithmetic unit for public key cryptosystems," in *Proceedings of the International Conference on*

Information Technology: Coding and Computing (ITCC'05), vol. 1, Apr. 4–6, 2005, pp. 568–573.

[164] M. A. Garcia-Martinez, R. Posada-Gomez, G. Morales-Luna, and F. Rodriguez-Henriquez, "FPGA implementation of an efficient multiplier over finite fields $GF(2^m)$," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs: (ReConFig'05)*, Sept. 28–30, 2005.

[165] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 449–460, Apr. 2003.

[166] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, Jan. 1987.

[167] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology CRYPTO'85*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 218. Springer-Verlag Berlin, Germany, 1986, pp. 417–428.

[168] J. López and R. Dahab, "Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation," in *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems: (CHES'99)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 1717. Worcester, MA, USA: Springer-Verlag Berlin, Germany, Aug. 1999, pp. 317–327.

[169] M. Brown, D. Hankerson, J. López, and A. Menezes, "Software implementation of the NIST elliptic curves over prime fields," in *Progress in Cryptology (CT-RSA'01): The Cryptographers Track at the RSA*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2020. San Francisco, CA, USA: Springer-Verlag Berlin, Germany, Apr. 8–12, 2001, pp. 250–265.

[170] D. Hankerson, J. López, and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields," in *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems - (CHES'00)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 1965. Worcester, MA, USA: Springer-Verlag Berlin, Germany, Aug. 2000, pp. 243–267.

[171] G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for $GF(2^m)$," in *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems - (CHES'00)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 1965. Worcester, MA, USA: Springer-Verlag Berlin, Germany, Aug. 2000, pp. 41–56.

[172] G. Orlando and C. Paar, "A scalable $GF(p)$ elliptic curve processor architecture for programmable hardware," in *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems: (CHES'01)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2162. Paris, France: Springer-Verlag Berlin, Germany, May 14–16, 2001, pp. 348–363.

[173] G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "A fast elliptic curve cryptosystem," in *Advances in Cryptology - EUROCRYPT'89: Workshop on the Theory and Application of Cryptographic Techniques*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 434. Houthalen, Belgium: Springer-Verlag Berlin, Germany, Apr. 1989, pp. 706–708.

[174] S. K. B. Lejla, P. Bart, and V. Ingrid, "Multi-core curve-based cryptoprocessor with reconfigurable modular arithmetic logic units over GF(2^n)." *IEEE Transactions on Computers : Accepted for future publication*, vol. PP, no. 99, 2007.

[175] F. Sozzani, G. Bertoni, S. Turcato, and L. Breveglieri, "A parallelized design for an elliptic curve cryptosystem coprocessor," in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, vol. 1, Apr. 4–6, 2005, pp. 626–630.

[176] A. D. Woodbury, D. V. Bailey, and C. Paar, "Elliptic curve cryptography on smart cards without coprocessor," in *Proceedings of the Fourth Smart Card Research and Advanced Applications Conference (CARDIS 2000)*, Bristol, UK, Sept. 20–22, 2000.

[177] G. M. de Dormale, R. Ambroise, D. Bol, J.-J. Quisquater, and J.-D. Legat, "Low-cost elliptic curve digital signature coprocessor for smart cards," in *Proceedings of International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, Boston, Massachusetts, USA, Sept. 2006, pp. 347 – 353.

[178] M. Bednara, M. Daldrup, J. Teich, J. Gathen, and J. Shokrollahi, "Tradeoff analysis of FPGA based elliptic curve cryptography," in *IEEE International Symposium on Circuits and Systems (ISCAS'02)*, vol. 5, May 26–29, 2002, pp. 797–800.

[179] N. Gura, S. Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila, "An end-to-end systems approach to elliptic curve cryptography," in *Proceedings of the Fourth International Workshop on Cryptographic Hardware and Embedded Systems - (CHES'02)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2523. Redwood Shores, CA, USA: Springer-Verlag Berlin, Germany, Aug. 13–15, 2002, pp. 15–22.

[180] M. Bednara, M. Daldrup, J. Gathen, J. Shokrollahi, and J. Teich, "Reconfigurable implementation of elliptic curve crypto algorithms," in *Proceedings of the*

International Symposium on Parallel and Distributed Processing (IPDPS'02), 2002, pp. 157–164.

[181] J. Lutz and A. Hasan, "High performance FPGA based elliptic curve cryptographic co-processor," in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, vol. 2, Apr. 26–29, 2004, pp. 486–492.

[182] N. Nguyen, K. Gaj, D. Caliga, and T. El-Ghazawi, "Implementation of elliptic curve cryptosystems on a reconfigurable computer," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT'03)*, Dec. 15–17, 2003, pp. 60–67.

[183] S. B. Ors, L. Batina, B. Preneel, and J. Vandewalle, "Hardware implementation of an elliptic curve processor over $GF(p)$," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'03)*, June 24–26, 2003, pp. 433–443.

[184] N. Mentens and S. B. rs, "An FPGA implementation of an elliptic curve processor over $GF(2^m)$," in *Proceedings of the* 14^{th} *edition of ACM Great Lakes Symposium on VLSI (GLSVLSI'04)*, Boston, Massachusetts, USA, Apr. 26–28, 2004.

[185] L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede, "Balanced point operations for side-channel protection of elliptic curve cryptography," *IEE Proceedings Information Security*, vol. 152, no. 1, pp. 57–65, Oct. 2005.

[186] J. López and R. Dahab, "Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation," in *First International Workshop: (CHES'99)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 1717,

Worcester, MA, USA: Springer-Verlag Berlin, Germany, Aug. 1999, pp. 316–327.

[187] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*, 1st ed., ser. London Mathematical Society (Lecture Note Series 265). Cambridge, UK: Cambridge University Press, 2000.

[188] J. López and R. Dahab, "Improved algorithms for elliptic curve arithmetic in $GF(2^n)$," in *Selected Areas in Cryptography: Fifth Annual International Workshop, SAC'98*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 1556. Kingston, Ontario, Canada: Springer-Verlag Berlin, Germany, Aug. 1998, pp. 201–212.

[189] Xilinx, "Virtex-IV platform FPGAs: Complete data sheet," May 2007. [Online]. Available: http://direct.xilinx.com/bvdocs/publications/ds112.pdf. Last accessed: June 2007.

[190] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blmel, "A reconfigurable system on chip implementation for elliptic curve cryptography over $GF(2^n)$," in *Proceedings of the Fourth International Workshop on Cryptographic Hardware and Embedded Systems - (CHES'02)*, ser. Lecture Notes in Computer Science, Springer, Berlin, vol. LNCS 2523. Redwood Shores, CA, USA: Springer-Verlag Berlin, Germany, Aug. 13–15, 2002, pp. 175–192.

[191] M. Fayed, M. W. El-Kharashi, and F. Gebali, "A high-speed, high-radix, processor array architecture for real-time elliptic curve cryptography over $GF(2^m)$," *Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, June 2007.

Appendix A

Multiplicative Inverse in Composite Fields

In composite field $GF((2^4)^2)$, an element $E \in GF(2^8)$ can be expressed as

$$E(x) = e_h x + e_l \tag{A.1}$$

where $e_h,\ e_l \in GF(2^4)$, $e_h,\ e_l$ represents the higher and lower four bits respectively, and x is the root of the irreducible polynomial $P(x)$ defined in Equation (4.10). Let $I(x)$ be the multiplicative inverse of $E(x)$, i.e., $I(x) = E^{-1}(x)$ then

$$I(x) = i_h x + i_l \tag{A.2}$$

By definition, we have

$$\begin{aligned} E(x).I(x) &= 1 \\ &= e_h i_h x^2 + (e_h i_l + e_l i_h)x + e_l i_l \end{aligned} \tag{A.3}$$

But, from Equation (4.10) we have: $P(x) = 0$, or

$$x^2 = Ax + B \tag{A.4}$$

Combining Equation (A.3) and (A.4) we obtain

$$(e_h i_l + e_l i_h + e_h i_h A)x + (e_l i_l + e_h i_h B) = 0x + 1 \tag{A.5}$$

Equating coefficients in Equation (A.5) we obtain

$$e_h i_l + e_l i_h + e_h i_h A = 0 \tag{A.6}$$

$$e_l i_l + e_h i_h B = 1 \tag{A.7}$$

From Equation (A.6) we have

$$i_h = e_h i_l (e_l + e_h A)^{-1} \tag{A.8}$$

$$i_l = e_h^{-1} i_h (e_l + e_h A) \tag{A.9}$$

Solving Equations (A.7), (A.8), and (A.9) together we get

$$i_h = \frac{e_h}{e_l^2 + e_h e_l A + e_h^2 B} \tag{A.10}$$

$$i_l = \frac{e_h A + e_l}{e_l^2 + e_h e_l A + e_h^2 B} \tag{A.11}$$

Hence, the resulting $I(x) = E^{-1}(x)$ will be

$$(i_h x + i_l) = \frac{e_h x + (e_h A + e_l)}{e_l^2 + e_h e_l A + e_h^2 B} \tag{A.12}$$

Since we have $A = \omega^2$ and $B = 1$, Equation (A.12) will be

$$(i_h x + i_l) = \frac{e_h x + (e_h A + e_l)}{e_l^2 + e_h e_l A + e_h^2} \tag{A.13}$$