Hardware Processors for Pairing-Based Cryptography

Robert Ronan B.E.

September 13, 2016

A Thesis Submitted to the
National University of Ireland
in Fulfilment of the Requirements for
the Degree of Doctor of Philosophy

Supervisor: Dr. Colin Murphy
Head of School: Prof. Nabeel A. Riza

Department of Electrical and Electronic Engineering, School of Engineering,
National University of Ireland, Cork.

Abstract

Bilinear pairings can be used to construct cryptographic systems with very desirable properties. A pairing performs a mapping on members of groups on elliptic and genus 2 hyperelliptic curves to an extension of the finite field on which the curves are defined. The finite fields must, however, be large to ensure adequate security. The complicated group structure of the curves and the expensive field operations result in time consuming computations that are an impediment to the practicality of pairing-based systems.

The Tate pairing can be computed efficiently using the η_T method. Hardware architectures can be used to accelerate the required operations by exploiting the parallelism inherent to the algorithmic and finite field calculations. The Tate pairing can be performed on elliptic curves of characteristic 2 and 3 and on genus 2 hyperelliptic curves of characteristic 2. Curve selection is dependent on several factors including desired computational speed, the area constraints of the target device and the required security level.

In this thesis, custom hardware processors for the acceleration of the Tate pairing are presented and implemented on an FPGA. The underlying hardware architectures are designed with care to exploit available parallelism while ensuring resource efficiency. The characteristic 2 elliptic curve processor contains novel units that return a pairing result in a very low number of clock cycles. Despite the more complicated computational algorithm, the speed of the genus 2 processor is comparable. Pairing computation on each of these curves can be appealing in applications with various attributes. A flexible processor that can perform pairing computation on elliptic curves of characteristic 2 and 3 has also been designed. An integrated hardware/software design and verification environment has been developed. This system automates the procedures required for robust processor creation and enables the rapid provision of solutions for a wide range of cryptographic applications.

Contents

List of Figures

List of Tables

List of Algorithms

Declaration

I certify that this material is my own work and has not been submitted for any other qualification.

Signed:

Date:

Dedication

This thesis is dedicated to Mam, Dad and Gary. Without your love, support and strength this Ph.D. and many other things would never have been possible.

Acknowledgements

"*The hardest arithmetic to master is that which enables us to count our blessings.*"
– Eric Hoffer

I would first like to thank my supervisor Colin Murphy. Quite aside from his technical expertise, he has always demonstrated an enduring patience and personal understanding. Liam Marnane's open door and help throughout my time in UCC have also meant a lot. I would like to thank Colm Ó hÉigeartaigh, Maurice Keller, Andrew Byrne and Tim Kerins for their collaborative work.

The technical lessons that I learned from the research described in this thesis pale in comparison to the knowledge that I gained about myself and, in particular, about the people around me. I will never be able to put into words the gratitude that I feel for my friends and family for helping me, for picking me up when I've fallen and for giving me confidence in myself and a lifelong belief in the enduring kindness of others. I'd like to mention and thank a few people by name in no meaningful order. If I've left your name out please forgive me, but hopefully you know that it was only in error.

To Heaphy for his constant ability to make me laugh (and allow me to laugh at him) and to make me comfortable with myself. To Rick and Kerry for their kindness and their unbelievable warmth down the years. To Kev and Monika for the fun nights out and their never ending ability to make me laugh. Thanks to Twomey, a close friend even if there usually is a pond between us. I want to thank Ronan and Anne Marie for their close friendship over the last few years and, of course, for the Tuesday (anything but Russian) DVD nights. My postgrad friends Ray and Steve have seen all of me, and its because of this that it feels so natural to be around them. I went to school and college with Eoin and Darren and I hope our ability to ruthlessly slag each other one minute and laugh about old times the next will never leave us. Although he's been living in a different country since we properly talked first, a natural and strong friendship has grown between myself and Alan. Sully and Elaine have always had a kind word, a smile and a funny story handy. Thanks to Shane for the craic and for his company at gigs and Ireland matches. Sarah's provided Gary with great happiness and her ability to take my Dad down a peg or two is always worth seeing in action. Thanks to all the Ronans and Melsops who have always been so loving.

I'd like to save my final thanks for Gary, for my mother and for my father. They've moved mountains and more for me over the last 13 years. They've shown me strength and drive when I forgot what it looked like. I wouldn't be sitting here writing this on a rainy Sunday in September without them. My hope is that they'll always know just how much love I hold for them.

"*Our virtues and our failings are inseparable, like force and matter. When they separate, man is no more.*"
– Nikola Tesla

Explanatory Note

The research work presented in this thesis was performed between the years 2003 and 2007 inclusive. Chapters 1-6 are written in the context of the state of the art of pairing-based cryptography at that time. Chapter 7 reviews the advances and changes in the research area between the years 2008 and 2015 inclusive.

Abbreviations

AES	Advanced Encryption Standard
ABE	Attribute-Based Encryption
ALU	Arithmetic Logic Unit
APB	Advanced Peripheral Bus
API	Application Programming Interface
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ASIP	Application Spefic Instruction Set Processor
AT	Area-Time
BDH	Bilinear Diffie-Hellman Problem
BN	Barreto-Naehrig
BRAM	Block Random Access Memory
CA	Certification Authority
CAU	Configurable Arithmetic Unit
CDH	Computational Diffie-Hellman Problem
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DAG	Direct Acyclic Graph
DES	Data Encryption Standard
DLP	Discrete Logarithm Problem
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
EEA	Extended Euclidean Algorithm
FCC	Fast Carry Chain
FPGA	Field Programmable Gate array
FSM	Finite State Machine
GCD	Greatest Common Divisor
GPS	Global Positioning System

HAN	Home Area Network
HCC	Hyperelliptic Curve Cryptography
HCDLP	Hyperelliptic Curve Discrete Logarithm Problem
HMM	Hybrid Montgomery Multiplier
IBE	Identity-Based Encryption
IBS	Identity-Based Signature
ID-NIKDS	Identity-Based Non-Interactive Key Distribution Scheme
LUT	Lookup Table
MAC	Multiply-And-Accumulate
MIRACL	Multiprecision Integer and Rational Arithmetic C Library
MM	Montgomery Method
NAF	Non-Adjacent Form
NFC	Near Field Communication
NoC	Network on Chip
PBC	Pairing-Based Cryptography
PKC	Public Key Cryptography
PKG	Private Key Generator
PKI	Public Key Infrastructure
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
RNS	Residue Number System
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
SWAR	SIMD Within A Register
TMVP	Toeplitz Matrix-Vector Product
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WSN	Wireless Sensor Network

Chapter 1

Introduction

1.1 Motivation

The earliest cryptographic schemes were employed to ensure the secrecy of a message in physical transit. Methods included substitution cyphers, in which each letter of the message was replaced with a letter offset by a fixed number of positions in a repeating alphabet. Julius Caesar famously employed this scheme to communicate with his generals during his military campaigns. The secret key in this case was the number of letters through which the message was shifted, but once this was known, decryption was trivial. The importance of privacy has grown at a staggering rate and cryptographic systems have become increasingly robust to meet the requirements of parties that wish to communicate.

The rotor machines of World War II are perhaps the most well known cryptographic devices of the early 20th century. The rotors controlled a path through an electrical system. Each letter of the plaintext resulted in a rotor movement, which meant that even repeated letters within a message were encrypted to different letters of the ciphertext through the constantly changing pathways.

In the modern age, cryptography has moved from traditional government and military applications to the realm of commerce and the general public. Meaningful communication would be unacceptably constrained without protection of user information and a defined probability of maintaining security. Currently, security relies on the difficulty of solving certain mathematical problems, known as *one-way functions*. These functions should be efficiently computable by verified users of the scheme. Reversal of the function by an adversary should, however, be intractable (in practice) without some added information, usually referred to as a key.

The constant advance of processing power means that existing cryptographic schemes are ever more susceptible to attack. If a known attack on a scheme becomes realistic, the difficulty of the mathematical problem must be increased. Unfortunately, this means that computations required by users of the system also become more expensive. As communication volume grows the viability of modern systems is becoming increasingly reliant on the efficient and fast computation of mathematical functions by users of the system.

Singh presents an interesting and engaging overview of the evolution of cryptography in [4].

1.2 Thesis Aims

Bilinear pairings have been used to construct many protocols with very desirable properties. An overview of some some such schemes is provided in [5]. Their fast and efficient computation is vital if pairing-based schemes are to be used in real-world scenarios. Pairings are computed on algebraic curves, known as supersingular elliptic and genus 2 hyperelliptic curves. These curves are defined on finite fields. Pairing algorithms can be expressed in terms of arithmetic operations on these finite fields and their extensions.

The Tate pairing has traditionally been the most efficiently computable pairing. It can be computed relatively quickly by performing an η_T pairing calculation followed by a suitable exponentiation. In this thesis, this is referred to the η_T method for Tate pairing computation. However, even with the algorithmic operations available to date, Tate pairing calculation remains a complex, expensive and time consuming operation.

Custom hardware processors can provide substantial accelerations in pairing computation times when compared to implementations on general purpose serial processors. They enable the exploitation of available parallelism in the required operations. Two types of parallelism are possible in the case of pairings. Firstly, the underlying finite field arithmetic operations can, in many cases, be performed in terms of parallel operations on a smaller field. Secondly, hardware units can be designed so that the main computational stages of the pairing algorithms can be performed in parallel. With careful scheduling, this can result in a significant reduction in computation time.

A number of factors should be considered when devising a pairing-based scheme. The system should be created with a desired computation speed and security level in mind. The area constraints of the target device(s) should also be considered. Each of the curves has a *security multiplier* that determines the field size required for a particular level of security. Elliptic curves of characteristic 2 and 3 have security multipliers of 4 and 6 respectively. Characteristic 2 genus 2 hyperelliptic curves have a security multiplier of 12. This means that, for a particular security level, the smallest field size can be used in the genus 2 case. If the required security level increases, the genus 2 implementation will also scale more efficiently due to the large security multiplier. Pairing computation on genus 2 curves is, however, more complicated than on elliptic curves. This added complexity should be considered during the designation of a pairing-based system. A quantitative comparison between the two elliptic cases is not straightforward. Some mathematical attributes of the characteristic 3 Tate pairing, along with the higher values of the security multipliers, mean that a smaller finite field size can be used in comparison to the characteristic 2 case. The implementation of characteristic 3 systems is, however, more expensive than environments that rely on characteristic 2 computations.

The hardware implementation of the Tate pairing using the η_T method has not been published to date. Furthermore, there have been no previous implementations of hardware processors for any type of pairing on genus 2 hyperelliptic curves. In this work, the feasibility of accelerators for the Tate pairing using the η_T method is examined in the two elliptic cases and in the genus 2 case. Efficient hardware units that perform many of the required operations in parallel are discussed. Pairing processors that use these units to return a Tate pairing computation in a very low number of clock cycles are presented. A processor that uses a programmable unit for finite field computation is also discussed. This processor is flexible and can used in a wide range of environments, from high speed systems that are relatively unconstrained in terms of area to small, low power devices. The processors are implemented on reconfigurable devices known as Field Programmable Gate Arrays (FPGAs). FPGAs contain a large number of lookup tables, memory and programmable interconnect. They are relatively inexpensive and their ease of reconfigurability means that they are an ideal target technology on which to implement these processors. A Very High Speed Integrated Circuit (VHSIC) language is used to describe the processors. Architectures are defined at the Register Transfer Level (RTL) using VHSIC Hardware Description Language (VHDL). The processors are designed so that the finite field size on which the curves are defined can be changed at will. This means that the processors can be regenerated with ease if the security level of a system must be changed.

Once an environment and target application have been chosen, curve and pairing selection are not trivial as many interconnected factors must be considered. This problem has not yet been well explored in the literature. A custom software design suite has been created that automatically generates a wide range of hardware architectures according to user specifications. This suite communicates with the FPGA and enables rapid prototyping and benchmarking. It can be used to reduce the effort required to create low level instruction sequences. Verification can also be performed automatically. This means that, once a pairing-based protocol has been defined, a system designer can examine the suitability of various processors with various area constraints, desired computation speed and security levels in mind.

1.3 Thesis Outline & Contributions

In this section, a thesis outline is provided and the contributions of this work are discussed.

Chapter 2 introduces the background necessary for an understanding of the subject matter of this research. A brief description of private key and public key systems is provided. The use of bilinear pairings in cryptography is discussed to provide context for this work. The theory of finite fields is outlined. The mathematics of elliptic and genus 2 hyperelliptic curves is presented. Computation of the group operation of these curves is explained. Security considerations for curve-based systems are also discussed. Finally, some benefits of the use of elliptic and hyperelliptic curves in cryptography are outlined.

Chapter 3 is concerned with the computation of the Tate and η_T pairings. The η_T pairing is presented and Tate pairing computation, using the η_T method, discussed. Considerations for the use of pairings in cryptography are outlined. The concepts underpinning Identity-Based Encryption (IBE) are discussed and the Boneh-Franklin IBE scheme presented. The methodology used for the hardware implementation of the Tate pairing in this work is discussed. A design system that has been created for the automatic generation, implementation, benchmarking and verification of pairing processors is presented. This can be used to generate processors according to various security, area and speed requirements. It can also be used to aid in curve and pairing selection on setup or modification of a cryptographic scheme.

In Chapter 4, a dedicated processor for pairing computation on elliptic curves of characteristic 2 is presented. The algorithms and operations required for Tate pairing computation using the η_T method are discussed. The hardware modules that implement $\mathbb{F}_{2^m}$ arithmetic are described. Techniques for the efficient compution of $\mathbb{F}_{2^{4m}}$ arithmetic are discussed and efficient extension field hardware architectures presented. Hardware units that perform the various steps of Tate pairing computation are discussed. These units are designed in a manner that enables calculations to be performed in parallel during the most expensive computational loop of the pairing algorithm. Finally, the top level processor that utilises these architectures to return a fast Tate pairing result is presented. The efficient scheduling of operations in the processor is also discussed. Results demonstrate that the processor can return a pairing result in a low number of clock cycles. The subject matter of this chapter was published in [6], [7] and [8].

The hardware implementation of the genus 2 hyperelliptic curve Tate pairing is discussed in Chapter 5. A design strategy for the computation of the Tate pairing, using the η_T method, in this case is outlined. The large extension degree means that care must be taken to ensure that arithmetic units are not too large or complex. A tower of extensions is used to reduce arithmetic operations on $\mathbb{F}_{2^{12m}}$ to operations on $\mathbb{F}_{2^{6m}}$. These subfield operations can be performed either serially or in parallel, depending on their complexity and resource availability. In practice, dedicated units for each of the computational steps of the algorithms cannot be used in the genus 2 case as the use of all available parallelism would result in an extremely large area footprint. Instead, custom hardware units are presented that share the most costly algorithmic operations. These units are designed to minimise the impact of resource sharing on computation time while maximising efficiency. The top level architecture of the pairing processor is presented. Results show that the genus 2 processor can return a pairing in a similar time to the characteristic 2 elliptic curve processor. Furthermore, it can return comparable results while utilising fewer resources. The topics discussed and architectures described in this chapter were published in [9] and [10].

A flexible processor for pairing computation on elliptic curves of both characteristic 2 and 3 is presented in Chapter 6. At this point in the thesis, characteristic 2 arithmetic has already been detailed in previous chapters. Characteristic 3 arithmetic is discussed and hardware modules that implement the required operations are presented. The processor described in this chapter does not contain any extension field arithmetic units. Instead, several subfield arithmetic modules operate in parallel. The most costly arithmetic operation is multiplication. For this reason, the number of multipliers can be varied at will and the design regenerated automatically using the platform that has been developed. This means that the pairing processor can be implemented on devices with extremely small area and strict power constraints by limiting the number of multipliers. A design subsystem that can be used to quickly generate the instruction sequences required to implement pairing algorithms is also discussed. Results returned show that the flexible processor has a very high level of efficiency. The topics covered in this chapter were published in [11], [12] and [13].

In Chapter 7, a review of modern pairing-based cryptography is provided. Several pairings with features that are attractive for the security requirements of contemporary schemes are described. Currently known computational and side channel attacks are outlined, along with countermeasures that can be used to render the attacks infeasible. The software computation of pairings is discussed. This is an area that has received much attention in recent years as some suggested systems rely on the efficient computation of pairings on small devices such as microprocessors. The state of the art of pairing implementation using dedicated hardware architectures is also discussed. Some interesting modern applications of pairings are described. Finally, some future work in the area of pairing-based cryptography is suggested.

Chapter 2

Background

2.1 Introduction

The concepts required for an understanding of this research are discussed in this chapter. Private and public key cryptography are described in Section 2.2. An introduction to cryptography based on curves and pairings is also provided. Cryptographically suitable curves have coordinates that exist on finite fields. These fields are discussed in Section 2.3. The mathematical theory underpinning hyperelliptic curves is provided in Section 2.4. Some curves are not suitable for cryptographic purposes since they are vulnerable to practical attacks. Two types of hyperelliptic curves, known as elliptic and genus 2 curves, can be used securely and are discussed in Section 2.5. Security considerations for curve-based schemes are outlined in Section 2.6. The merits of hyperelliptic curve cryptography are also discussed. Finally, a hyperelliptic curve protocol is described in detail to provide some context to the subject matter of this chapter.

2.2 Cryptography

This section provides a brief introduction to cryptography. Some terminology is first defined. The message to be sent is known as the *plaintext*. The disguised version of the message, to be sent across an insecure channel, is known as the *ciphertext*. *Encryption* is the means of converting the plaintext to ciphertext, and *decryption* is the procedure used to convert it back again. The intended viewers of the plaintext are called the *recipients*. Unintended viewers of the message while it is in transit are called *eavesdroppers*. A *key* is a tool that can be used to encrypt or decrypt a message. A *cryptosystem* is the finite set of possible plaintexts, possible ciphertexts, possible keys and algorithms for encryption and decryption. An overview of modern cryptography, the algorithms used, and the practical implementation of cryptographic systems is available in [14].

2.2.1 Private Key Cryptography

In private key cryptography, a single key is used for both encryption and decryption. The sender encrypts a message using a key and sends the ciphertext to the recipient. The recipient then decrypts the ciphertext using the same key. This is known as symmetric key cryptography. In this case the same key must be made available to both the recipient and the sender, but must remain hidden from eavesdroppers. An illustration of this type of system is provided in Figure 2.1.

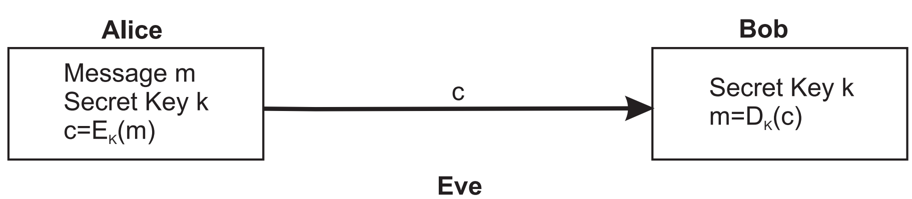

Figure 2.1: Private Key Cryptographic System

Here, Alice uses the encryption algorithm E, along with the secret key k, to encrypt a message m. This produces the ciphertext c, which is sent across the insecure channel.

On receipt of c, Bob uses the decryption algorithm D and the secret key to retrieve the original plaintext.

Private key schemes can be implemented using *block ciphers* and *stream ciphers*. Block ciphers operate on a fixed length of plaintext and produce a ciphertext of the same fixed size. The *Advanced Encryption Standard* uses block ciphers [15]. Stream ciphers encrypt a small segment of the plaintext at a time using a pseudorandom stream of small keys. These ciphers are suited to applications in which the size of the plaintext is not known in advance. In practice, block ciphers are regularly used in conjunction with stream ciphers in private key cryptographic schemes. A good introduction to ciphers is available in [16].

Private key schemes have the capacity for very high data throughput. Key distribution is, however, a significant problem. Keys cannot be transferred over the insecure channel as, if intercepted by an adversary, all communication will be compromised. In the recent past, some keys were sent by physical means such as a private courier. It is clear that this is not feasible in a modern society in which a large number of connections are established every second. In systems with a large number of users, key management is also problematic as a large number of unique keys is required.

2.2.2 Public Key Cryptography

Public key cryptography can help to solve the problems associated with key distribution, management and authentication. In 1976 Diffie and Hellman proposed a system that allows users to safely share a secret key over an insecure channel [17]. The shared key can then be used in a high throughput private key scheme. This is the first published example of a public key cryptographic scheme.

To aid in the understanding of public key cryptography, consider the case in which Alice and Bob wish to generate a shared key for communication over an insecure channel. Alice and Bob are first each assigned a private key and a public key. Security does not depend on the inaccessibility of the public keys and they can be made freely available. Alice and Bob must keep their private keys secret. Let G be a cyclic group of order n with a generating element g. Alice chooses a random integer $1 \leq a < n$. This is her private key. She then computes her public key g^a. Bob chooses a random integer $1 \leq b < n$. This is his private key. He then computes his public key g^b. After this, Alice retrieves Bob's public key and computes $(g^b)^a$. Bob computes $(g^a)^b$. Now both Alice and Bob have computed g^{ab}, which is the value of the shared secret key. An eavesdropper has to solve the *Diffie-Hellman Problem* (DHP) to obtain the shared secret key. This is the problem of computing the value of g^{ab} given g, g^a and g^b. The intractability of this problem relies on the difficulty of the *Discrete Logarithm Problem* in the group. In practice, the group order n must be very large to ensure adequate security.

The scheme is, however, susceptible to a *man in the middle* attack. An eavesdropper can tamper with or replace the public keys of Alice and Bob if she intercepts them before the shared secret key has been generated. She can then pretend to be either party or simply listen to the conversation even though the parties believe that their communication is secure. To prevent this, Diffie and Hellman suggested the use of *digital signatures.* Alice can sign the message using her own private key and encrypt it using Bob's public key. Bob can now be sure that the key was created by Alice and not altered in transit by decrypting using her public key. A central *Public Key Infrastructure* (PKI) is usually required. A PKI is used to generate, manage and store *digital certificates.* These certificates are tied to keys so that users can be assured of their validity.

In 1978, Rivest, Shamir and Adleman devised RSA, the first usable public key encryption scheme [18]. In RSA, public keys are used for encryption and private keys for decryption. System security relies on the intractability of factoring the product of two large prime numbers. In 1985, Elgamal proposed an encryption scheme that, instead, relies on the intractability of the discrete logarithm problem in a finite field $\mathbb{F}_q$, where q is the number of elements in the field. Sub-exponential attacks can be used against the integer factorisation problem and the DLP on a finite field. This means that very large key sizes must be used to ensure an adequate level of security.

In 1986, Miller [19] and Koblitz [20] suggested that elliptic curves can be used in public key cryptographic schemes. Security is manifested in the intractability of the DLP in the group of elliptic curve points with coordinates in $\mathbb{F}_q$. This is a more difficult problem than in the finite field case and smaller key sizes can, therefore, be used. The group operation is, however, more complicated. In 1989, Koblitz showed that curves of any genus can be used [21]. Genus 2 curves are invulnerable to some attacks on elliptic curves and even smaller key sizes can be used. However, the group operation is more complex than in the elliptic case.

An introduction to public key cryptography and its infrastructures is provided in [22]. A comprehensive overview of elliptic curve cryptography is available in [23].

2.2.3 Cryptography Based on Pairings

In 2000, Joux devised a one round tripartite Diffie-Hellman key agreement scheme using the bilinearity property of pairings [24]. This was the first example of the constructive use of pairings (previous to this, they had been used for the purposes of cryptographic attack). Identity-Based Encryption (IBE) schemes are the most well known application of pairings in cryptography. The concept of IBE was first proposed by Shamir in 1984 [25]. In his proposal, the generation of a shared secret key is not required before communication can commence. The identities of users of the scheme can be utilised to send messages securely. The creation of a fully usable IBE scheme remained an open problem for many years as a secure implementation method could not be found. In 2001, however, Boneh and Franklin used the bilinearity and other desirable properties of pairings to construct the first fully functional IBE system [26]. Public keys are directly linked to the identities of the users of an IBE scheme (email addresses or even real names can be used as keys). If Alice wishes to send a message to Bob she encrypts the message using Bob's identity and some other system parameters that are available to all users. On receipt of the message Bob decrypts the message using his private key. The steps required to perform the Boneh-Franklin IBE scheme are detailed in Subsection 3.4.2.

For comparative purposes, first consider a conventional public key scheme (which can be implemented using, for example, RSA or curve-based cryptography). The PKI registers users of the network. Users generate their own public keys from their private keys. All public keys must be sent to the PKI to be validated before they can be used. Digital certificates are tied to validated keys by a *Certification Authority* (CA). A typical validation and certification process is illustrated in Figure 2.2 (Network World [1]).

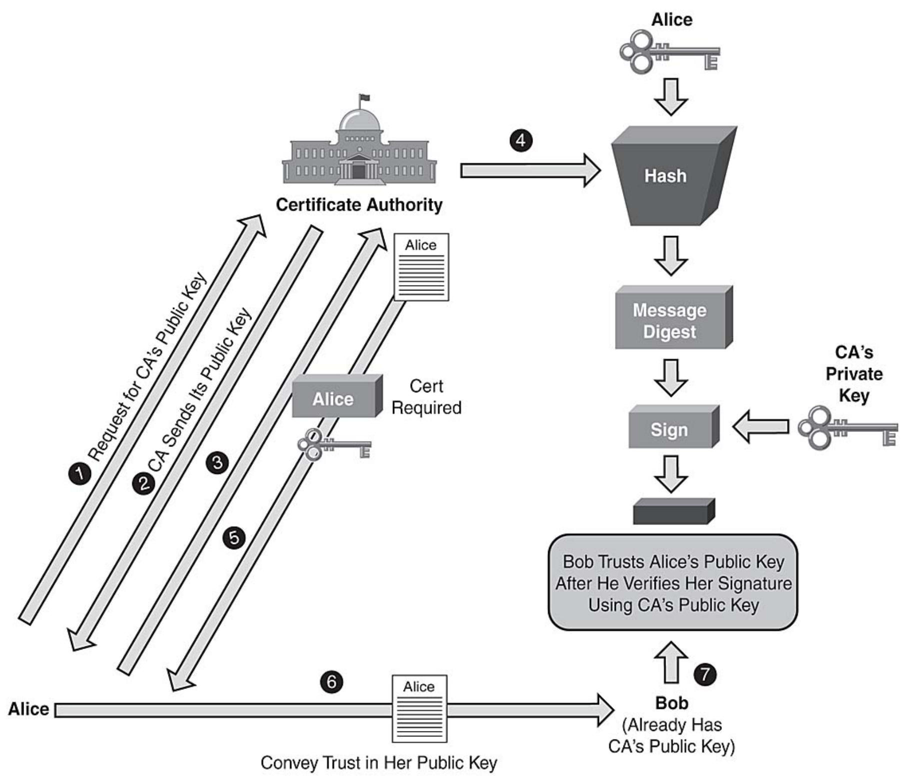

Figure 2.2: Conventional Public Key Validation and Certification Process (Source: Network World [1])

In order to establish trust with Bob, Alice must follow a number of steps. She first requests the public key of the CA, which is known as the root certificate. On receipt of this, she generates a certificate request containing her identity information and her public key, signs it using the root certificate and sends it to the CA. The CA verifies Alice's identity and creates a digital certificate for Alice by binding her identity and public key. The CA then signs this digital certificate using its own private key and issues the final certificate to Alice, to be used as her identity certificate. Finally, Alice presents this to Bob, who follows the digital signature verification process to establish trust in her public key. Once trust has been established, secure information exchange can be initiated. As the number of users grows, the key and certificate management system may become increasingly complex. The resources required to store the keys and certificates can also be costly.

A basic example of an IBE system is illustrated in Figure 2.3 (Wikipedia [2]). All private keys are generated from a shared master key by the Private Key Generator (PKG). Alice and Bob first obtain their own private keys from the PKG. The security of this transfer is paramount. To ensure secrecy, the key transfer can be performed off-line if desired. To send a message to Bob, Alice generates a public key using Bob's identity. She then encrypts and signs the message using her own private key, the public key and parameters that are available to all users of the system. On receipt of the message, Bob decrypts the message using Alice's public key, his own private key, and the publicly available parameters of the system.

The elimination of the PKI can result in significant savings in terms of resources and system complexity. Digital certificates are not required to establish public keys. This removes the necessity for a central public key and management environment. To further improve the security of an IBE system, the shared master key can be destroyed if no more users are to be added. The designers of IBE schemes must, however, endeavour to minimise the impact of the necessity for a secure private key transmission channel. Some schemes require the generation and communication of private keys only at setup. Private keys can also be transferred by physical means.

An example of a particularly useful application of IBE is in the area of Wireless Sensor Networks (WSNs). WSN systems usually consist of a large number of sensor devices, which communicate with each other to relay information to a central server. Before deployment,

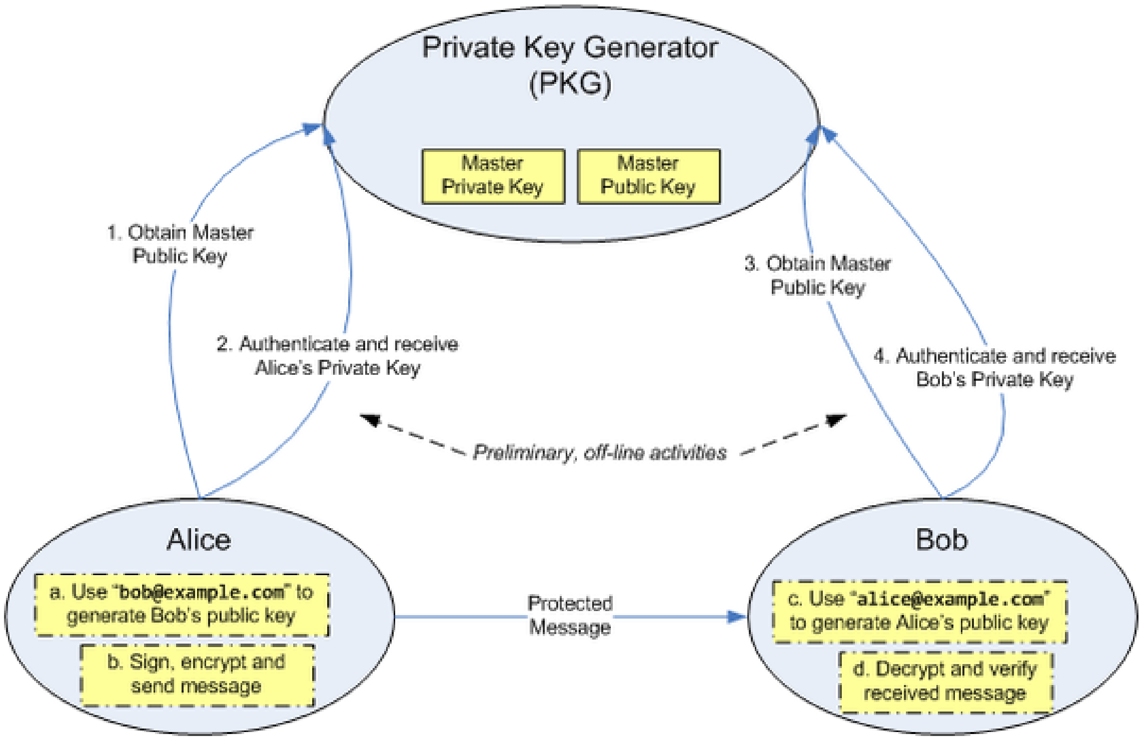

Figure 2.3: Identity-Based Encryption (Source: Wikipedia [2])

private keys can be transferred by physical means. Once deployed, low power consumption is vital since the devices may be located in environments from which they are difficult to retrieve. The devices are usually relatively inexpensive and the highest energy cost is commonly that of communication. IBC can be used to reduce the number of transmissions required to establish trust and initiate secure information exchange. An *Identity-Based Non-Interactive Key Distribution Scheme* (ID-NIKDS), described by Sakai et al. in [27], is very suitable for use in WSNs.

Canetti et al. [28] discuss how IBE can be used to construct a simple and efficient public key encryption scheme that provides security against chosen-cyphertext attacks. Pairings have also been used in the implementation of non identity-based cryptographic applications. Several signature schemes with very desirable properties have been proposed and implemented using pairings. Other applications include threshold schemes, in which private information can be distributed among several parties to reduce the impact of the failure of a single point in the system. A survey of such schemes is available in [5].

The advantages of ECC and IBE for wireless security are briefly outlined in [29]. A concise introduction to pairing-based cryptography can be found in the Master's thesis of Maas [30]. A detailed treatment on the various aspects of identity-based cryptography is available in [31].

2.3 Finite Field Theory

The work described in this thesis is based on computations on groups of hyperelliptic curve elements. The curves are defined on *finite fields*. This subsection provides an overview of the mathematics of these fields. A comprehensive resource for the theory of finite fields is available in [32]. Wong provides an overview of finite fields and their applications to cryptography in [33].

2.3.1 Groups and Rings

Before finite fields can be discussed, a brief introduction to groups and rings is necessary. Some definitions are also provided.

Definition 2.3.1. ***(group)*** *A group is an algebraic structure consisting of a set of elements together with a binary operation. Given a binary operator '+' (this notation is for convenience, the operator is not necessarily addition) then the following properties must be satisfied for all $a, b, c \in G$:*

- ***Closure:*** *If $d = a + b$, then d must also be a member of G.*
- ***Associativity:*** $(a + b) + c = a + (b + c)$.

- ***Identity:*** *For each a there exists an element $0 \in G$ such that $0 + a = a + 0 = a$.*

- ***Inverse:*** *For each $a \in G$, there exists some element $-a \in G$ such that $a + (-a) = (-a) + a = 0$*

Definition 2.3.2. ***(abelian group)*** *A group G is abelian if, for all $a, b \in G$, then $a+b = b + a$. An abelian group is also known as a commutative group.*

Definition 2.3.3. ***(finite group)*** *A group G is finite if it contains a finite number of elements.*

Definition 2.3.4. ***(group order)*** *The order of a group G, denoted $\#G$, is the number of elements in that group.*

Definition 2.3.5. ***(subgroup)*** *A non empty subset G' of G is a subgroup of G if G' forms a group under the same binary operation and inverse element as G.*

Definition 2.3.6. ***(cyclic group, generator, scalar multiplication)*** *A group G is cyclic if, for all elements $a \in G$, each associated with some integer $k_a < \#G$, there exists a single element $g \in G$ such that $[k_a]a = g$. The element g is known as the generator of the group. The operation $[k]a$, for any integer k, is known as scalar multiplication and is performed by adding a to itself k times. All cyclic groups are abelian.*

Definition 2.3.7. ***(order of a group member, torsion group)*** *Consider a group G and $a \in G$. The order of a is the lowest value of n such that $[n]a = 0$. The subgroup of elements with order n in G is known as an n-torsion group, denoted $G[n]$.*

Note that the order of a group generator is always equal to the group order.

Consider a group G with binary operator '$+$' and a group H with binary operator '$*$'. These groups can be written as $(G, +)$ and $(H, *)$ respectively.

Definition 2.3.8. ***(group homomorphism)*** *A group homomorphism from $(G, +)$ to $(H, *)$ is a mapping function $\phi : G \to H$ such that for all elements $a, b \in G$.*

$$\phi(a + b) = \phi(a) * \phi(b) \tag{2.1}$$

There are various mapping functions.

- **Injective function:** There will never be more than one element of G mapped to a single element of H. Note that there may be elements of H that do not have a corresponding element in G after the mapping.
- **Surjective function:** Every element of H has one (or more) corresponding element(s) in G after the mapping. An element of G may be mapped to more than one element of H.
- **Bijective function:** Every element of G is paired exactly with one element of H and every element of H is paired exactly with one element of G.

These mapping functions give rise to different types of group homomorphism.

1. **Group Monomorphism:** A homomorphism that is injective. Distinctness on G is preserved through the mapping.
2. **Group Epimorphism:** A homomorphism that is surjective. The mapping of elements from G reaches every element in H.
3. **Group Isomorphism:** A homomorphism that is bijective. The groups G and H are identical in operation and differ only in the notation of their elements.
4. **Group Endomorphism:** This is a homomorphism between G and itself, $\phi : G \to G$.
5. **Group Automorphism:** This is an endomorphism that is bijective. It is also an isomorphism. The set of all automorphisms of a group G forms a group known as the automorphism group of G.

Definition 2.3.9. *(**ring**) A ring R is a set of elements with two binary operators, usually referred to as addition and multiplication, '$+$' and '$*$'. The ring must satisfy the following properties for all $a, b, c \in R$:*

- ***Additive associativity:*** $(a + b) + c = a + (b + c)$
- ***Additive commutativity:*** $a + b = b + a$
- ***Additive identity:*** *For all a there exists an element $0 \in R$ such that $0+a = a+0 = a$.*
- ***Additive Inverse:*** *For every a, there exists some element $-a \in R$ such that $a + (-a) = (-a) + a = 0$*
- ***Left and right distributivity:*** $a*(b+c) = a*b+a*c$ *and* $(b+c)*a = (b*a)+(c*a)$
- ***Multiplicative associativity:*** $(a * b) * c = a * (b * c)$

Definition 2.3.10. ***(commutative ring)*** *A ring R is commutative if its multiplicative operation is commutative, i.e. $a * b = b * a$ for all $a, b \in R$*

An element $a \in R$ is *invertible* if $a * a^{-1} = 1$ where a^{-1} is some other element in R.

Definition 2.3.11. *(**subring**) A ring R' is a subring of R if R' is a subset of R and is a ring under the same addition and multiplication operations and identity elements.*

2.3.2 Finite Fields

Definition 2.3.12. *(**field**) A field F is a commutative ring in which all elements, excluding 0, have a multiplicative inverse.*

Definition 2.3.13. *(**subfield, extension field**) A field F' is a subfield of F if F' is a subset of F and is a field with respect to the same binary operations and identity elements. F is called an extension field with respect to F'. The extension field may be written as F/F' for clarity.*

Definition 2.3.14. ***(finite field)*** *A finite field is a field in which the number of elements is finite. A finite field is written as $\mathbb{F}_q$, where q is the field order (the number of elements in the field).*

To satisfy its axioms, the order of a finite field will always have the form $q = p^m$, where p is a prime and m is a positive integer. The prime p is known as the *characteristic* of the field and m is the *dimension* of the field. Fields with $p = 2, 3$ and $m > 1$ are considered in this research since pairings, when computed on curves that are defined on these fields, are very suitable for hardware implementation.

Polynomials can be used to represent elements of a finite field. Polynomial representation has many advantageous features for the computation of finite field arithmetic. A *generator polynomial*, denoted $f(x)$, is used to construct the field. This polynomial is of degree m and has, with the exclusion of constants, no roots in the field. It can be written as $f(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$, where all $f_i \in \mathbb{Z}$ and $f_i \in \{1-p, p-1\}$. The relationship $f(x) = 0 \in \mathbb{F}_q$ holds, and is useful for computation of field arithmetic. The generator polynomial is also known as the *irreducible polynomial.*

In this work, irreducible polynomials with the lowest number of non-zero coefficients are used to generate the field in use (in practice, trinomials and pentanomials are always available). When there is a choice between irreducible polynomials with the same number of coefficients, the polynomial which has the lowest $(m-1)^{th}$ coefficient is selected. This ensures that the hardware resources required to perform finite field arithmetic are minimised.

Given an element $a \in \mathbb{F}_q$, $q = p^m$, the polynomial representation of a in the variable x is

$$a(x) = \sum_{i=0}^{m-1} a_i x^i = a_0 + a_1 x + \ \ldots \ + a_{m-1} x^{m-1} \tag{2.2}$$

where all $a_i \in \mathbb{F}_p$ and $\mathbb{F}_p$ is the ring of integers modulo p . Note that the degree of the polynomial, denoted $\delta(a)$, has maximum value $m-1$.

Addition of two elements $a(x)$, $b(x) \in \mathbb{F}_q$ with degree $\delta(a)$ and $\delta(b)$, respectively, is performed coefficient-wise as

$$a(x) + b(x) = \sum_{i=0}^{n} (a_i + b_i) x^i \tag{2.3}$$

where $n = max(\delta(a), \delta(b))$. The coefficients are members of $\mathbb{F}_p$ and addition is, therefore, performed modulo p.

Multiplication of the polynomials $a(x)$ and $b(x)$ in a conventional manner may result in a polynomial with degree larger than $m-1$. This polynomial cannot be a member of $\mathbb{F}_q$. The generator $f(x)$ is used to reduce the result in this case. Multiplication proceeds as follows.

Consider the polynomials $a(x)$, $b(x) \in \mathbb{F}_q$ with degrees $\delta(a)$ and $\delta(b)$, respectively. Conventional polynomial multiplication returns

$$c'(x) = \sum_{i=0}^{\delta(a)} \sum_{j=0}^{\delta(b)} (a_i . b_j) x^{i+j}. \tag{2.4}$$

where $a_i, b_j \in \mathbb{F}_p$.

The degree of $c'(x)$ is $\delta(c') = \delta(a) + \delta(b)$. If $\delta(c')$ is larger than $m-1$ then the polynomial $c'(x)$ cannot be a member of $\mathbb{F}_q$ and must be reduced modulo the irreducible polynomial $f(x)$. Two methods can be employed to perform this reduction:

1. $c'(x)$ is divided by $f(x)$ using regular polynomial division (with subtraction being performed modulo p). If the remainder of this division, $r(x)$, has $\delta(r) \leq m-1$ then reduction is complete. If this is not the case, then $r(x)$ is repeatedly divided by $f(x)$ until $\delta(r) < m$. The final value of the remainder $r(x)$ is the multiplicative result and is a member of $\mathbb{F}_q$.

2. Since $f(x) = 0 \in \mathbb{F}_q$, then $x^m = \sum_{i=0}^{m-1} f_i x^i$. This relationship is used to remove the terms of the composition polynomial that are of degree greater than $m-1$.

As an example, consider the field $\mathbb{F}_{2^4}$, generated by $f = f(x) = x^4 + x + 1$. Let $a = a(x) = x^3 + x^2 + 1$ represent the element $(1101)_2$ and let $b = b(x) = x^3 + 1$ represent

$(1001)_2$. Multiplication of a and b begins with the composition stage, which produces the polynomial $c' = x^6 + x^5 + x^2 + 1$. The first reduction method involves dividing c' by f until the degree of the remainder is less than 4. Two rounds are required: the first returns $r = x^5 + x^3 + 1$, whilst the second returns $r = x^3 + x^2 + x + 1$. Since $\delta(r) \leq m - 1$ at this point, the value of $a.b \in \mathbb{F}_q$ is, therefore, given by r, which can also be written as $(1111)_2$. The second reduction method uses the relationship $x^4 = x + 1$. From this, $x^5 = x.x^4 = x(x+1) = x^2+x$. Also, $x^6 = x.x^5 = x(x^2+x) = x^3+x^2$. The reduced result is then given by $c = x^3 + x^2 + x^2 + x + x^2 + 1 = x^3 + x^2 + x + 1$.

Notation is required to represent the construction of $\mathbb{F}_q$ using the irreducible polynomial $f(x)$. Let $\mathbb{F}_p[x]$ be the ring of all possible polynomials with coefficients in $\mathbb{F}_p$. The field $\mathbb{F}_q = \mathbb{F}_{p^m}$ is an *m-dimensional extension* of $\mathbb{F}_p$. The field $\mathbb{F}_{p^m}$ is generated by the degree m polynomial $f(x) \in \mathbb{F}_p[x]$. This is written as

$$\mathbb{F}_q = \mathbb{F}_{p^m} \equiv F_p[x]/f(x) \text{ where } f(x) = \sum_{i=0}^{m} f_i x^i \text{ for all } f_i \in \mathbb{F}_p \text{ and } f_m \neq 0 \tag{2.5}$$

Put simply, $\mathbb{F}_{p^m}$ is the set of all possible polynomials with coefficients in $\mathbb{F}_p$ reduced modulo $f(x)$.

The field $\mathbb{F}_q$ can itself be extended. Let $\mathbb{F}_q[y]$ be the ring of all polynomials with coefficients in $\mathbb{F}_q$. If a k-dimensional extension of $\mathbb{F}_q$ is required, another polynomial $g(y)$ that is irreducible in $\mathbb{F}_{q^k}$ is selected and used to generate $\mathbb{F}_{q^k}$. This is written as

$$\mathbb{F}_{q^k} = \mathbb{F}_{(p^m)^k} \equiv F_q[y]/g(y) \text{ where } g(y) = \sum_{i=0}^{k} g_i y^i \text{ for all } g_i \in \mathbb{F}_q \text{ and } g_k \neq 0 \tag{2.6}$$

A series of extensions of the same field is known as a *tower of extensions.* It is possible to perform arithmetic on an extension field in terms of arithmetic on one or several of its subfields if towers are used. These subfield operations can be performed in parallel in hardware in many cases.

A useful automorphism exists on finite fields and their extensions. This automorphism can simplify the exponentiation of elements and can be used to reduce the complexity of finite field multiplication, as will be seen later in this thesis. Fermat's little theorem states that for any integer a and prime p then $a^p \equiv a \mod p$. On a prime field $\mathbb{F}_p$ this means that $a^p = a$ for every element a of the field. The mapping of a to a^p is called the *Frobenius automorphism* of the field. It can be written as a mapping $F : a \to a^p$ or as a function $F(a) = a^p$. The field is cyclic under this mapping and the Frobenius is said to *fix* the field. It is clear that this function respects multiplication on $\mathbb{F}_p$. Given $r, s \in \mathbb{F}_p$ then $F(rs) = (rs)^p = r^p s^p$ under the associative rule. The Frobenius also respects the addition rule. Given $F(r + s) = (r + s)^p$ it can be shown (using the binomial theorem) that $(r+s)^p = r^p + s^p$ on a prime field and the addition rule is, therefore, satisfied. These properties also hold for all extensions of $\mathbb{F}_p$.

Now, consider a prime power field $\mathbb{F}_q = \mathbb{F}_{p^m}$. The order of $\mathbb{F}_q$ is q, so if the field is to be cyclic then $b^q = b$ for all $b \in \mathbb{F}_q$. The required mapping in this case is $F : b \to b^q$. Since $q = p^m$, the mapping can be written as $F : b \to b^{p^m}$. This can be achieved by performing m iterations of $b \to b^p$. The Frobenius of $\mathbb{F}_q$ is known as the m-th order Frobenius and fixes the field $\mathbb{F}_p$ inside $\mathbb{F}_q$, i.e. $\mathbb{F}_q \equiv \mathbb{F}_q/\mathbb{F}_p$.

Consider a k-dimensional extension of $\mathbb{F}_q$. The field $\mathbb{F}_{q^k}$ has order q^k so a mapping $F : c \to c^{q^k}$ is required. Since $\mathbb{F}_{q^k}$ is an extension of $\mathbb{F}_q$, this can be written as $c \to c^{q^k}$. The mapping on this field can, therefore, be performed using k iterations of the m-th order Frobenius. This map fixes $\mathbb{F}_q$ inside $\mathbb{F}_{q^k}$, i.e. $\mathbb{F}_{q^k} \equiv \mathbb{F}_{q^k}/\mathbb{F}_q/\mathbb{F}_p$. The same properties hold for all extensions of $\mathbb{F}_q$. These Frobenius mappings, which are relatively easy to perform, prove very useful for pairing computation, as will be seen in the next chapter.

Some further definitions concerning extension fields are required before hyperelliptic curves can be introduced.

Definition 2.3.15. ***(algebraic element)*** *If a field L is an extension of a field K then an element $a \in L$ is algebraic over K if a has a root in some non-zero polynomial with coefficients in K, i.e. $b(a) = 0$ for some polynomial $b(x) \in K$.*

Definition 2.3.16. ***(algebraic closure of a field)*** *The algebraic closure of a field K, denoted $\bar{K}$, is a field in which every element has a root in a polynomial with coefficients* *in K.*

For a finite field with $q = p^m$, the algebraic closure of $\mathbb{F}_q$ is the union of all finite fields with order q^n, for all positive integers n.

$$\bar{\mathbb{F}}_q = \bigcup_{n=1}^{\infty} \mathbb{F}_{q^n} \tag{2.7}$$

2.4 The Mathematics of Hyperelliptic Curves

Hyperelliptic curves contain additive groups with properties that are advantageous to the construction of cryptographic protocols. The mathematics of these groups is outlined in this section. Note that discussion will remain general; the specific cases of elliptic and genus 2 hyperelliptic curves will not be discussed until Section 2.5. A brief overview of the mathematics of hyperelliptic curves is provided in [34]. A more comprehensive treatment of elliptic and hyperelliptic curves and their group arithmetic is available in [35].

2.4.1 Curve Definition

Definition 2.4.1. ***(hyperelliptic curve)*** *A hyperelliptic curve C on a field K is defined by*

$$C : y^2 + h(x)y = f(x) \tag{2.8}$$

where (x, y) are members of the algebraic closure $\bar{K}$, $h(x) \in K[x]$ is a polynomial of degree less than or equal to an integer g and $f(x) \in K[x]$ is a monic polynomial of degree $2g+1$. The value of g is known as the genus of the curve.

For cryptographic use, the curve should have no singular points. This means that there should be no point on C that satisfies both partial derivatives

$$2y + h(x) = 0 \tag{2.9}$$

$$h'(x)y - f'(x) = 0 \tag{2.10}$$

where $h'(x) = \frac{\delta h(x)}{\delta x}$ and $f'(x) = \frac{\delta f(x)}{\delta x}$.

A point P on C is represented by an (x, y) pair, where x and y are elements of the algebraic closure $\bar{K}$. The set of points, together with a special point at infinity ∞, is denoted $C(\bar{K})$. For convenience, $C(\bar{K})$ is written as C.

The opposite of a point $P \in C$ with $(x, y) \in \bar{K}$ is given by

$$-P = (-x, -y - h(x)) \tag{2.11}$$

Note that if $P = \infty$ then $-P = \infty$.

2.4.2 Divisors

The computations relevant to this research are performed on *divisors*.

Definition 2.4.2. ***(divisor)*** *A divisor D is a finite formal sum of points on C such that*

$$D = \sum_{P \in C} m_p P \tag{2.12}$$

where m_p is an integer and only a finite number of m_p are non-zero.

The notation m_p is used to describe the number of instances a particular point P exists in the divisor construction.

Definition 2.4.3. ***(degree of a divisor)*** *Given a divisor $D = \sum_{P \in C} m_p P$, the degree of D is the sum of all values of m_P*

$$deg(D) = \sum m_P \tag{2.13}$$

Definition 2.4.4. ***(support of a divisor)*** *The support of a divisor is the set of all points at which $m_p \neq 0$.*

Definition 2.4.5. ***(degenerate divisor)*** *A divisor with a single point in its support is known as a degenerate divisor.*

The set of all divisors on C, denoted $\mathbb{D}$, forms an abelian group under an addition law, which is defined as follows: Given two divisors $D_1 = \sum_{P \in C} m_P P$ and $D_2 = \sum_{P \in C} n_P P$ then

$$D_1 + D_2 = \sum_{P \in C} (m_P + n_P) P \tag{2.14}$$

The identity element of $\mathbb{D}$ is $\sum 0P = 0$ and the additive inverse of a divisor $D = \sum_{P \in C} m_p P$ is $-D = \sum_{P \in C} (-m_P) P$.

Definition 2.4.6. ***(greatest common divisor (gcd))*** *The greatest common divisor of $D_1 = \sum_{m_P} P$ and $D_2 = \sum_{n_P} P$ is given by*

$$gcd(D_1, D_2) = \sum min(m_P, n_P) P - \sum min(m_P, n_P) \infty \tag{2.15}$$

Note that $gcd(D_1, D_2)$ will be a divisor of degree 0.

2.4.3 Polynomial and Rational Functions

Definition 2.4.7. ***(coordinate ring, polynomial function)*** *Given a hyperelliptic curve C and an algebraic closure $\bar{K}$, the coordinate ring of C over $\bar{K}$ is the set of all polynomials in $\bar{K}$ reduced modulo C. It is a quotient ring defined as*

$$\bar{K}[C] = \bar{K}[x, y]/(y^2 + h(x)y - f(x)) \tag{2.16}$$

An element of $\bar{K}[C]$ is called a polynomial function of C.

Definition 2.4.8. ***(rational function)*** *Given polynomial functions $G, H \in \bar{K}[C]$ and $H \neq 0$ a rational function R is given by*

$$R = G/H \tag{2.17}$$

The field of rational functions on C is written as $\bar{K}(C)$.

The somewhat subtle difference between the notation used for rings and field should be noted. If R is a ring then $R[x]$ denotes the set of polynomials in x with coefficients from R. If L is a field then $L(x)$ denotes the set of polynomials in x with coefficients from L.

Definition 2.4.9. ***(uniformising parameter, intersection multiplicity)*** *Let P be a point on C. Let d be an integer and $U, S \in \bar{K}(C)$ such that $U(P) = 0$ and $S(P) \neq 0, \infty$. Each polynomial $G \in \bar{K}[C]$ satisfies the relationship $G = U^d S$ at P. The function U is known as the uniformising parameter of G. The integer d is known as the intersection multiplicity of G at P and does not depend on the choice of U.*

Definition 2.4.10. ***(order of a polynomial function)*** *Consider a polynomial function $G \in \bar{K}[C]$. The order of G at a point P is equal to the intersection multiplicity d of G at P and is given by*

$$ord_P(G) = d \tag{2.18}$$

Definition 2.4.11. ***(divisor of a polynomial function)*** *The divisor of a polynomial function $G \in \bar{K}[C]$ is given by*

$$div(G) = \sum_{P \in C} [ord_P(G)]P \tag{2.19}$$

Definition 2.4.12. ***(divisor and order of a rational function)*** *Let $R = G/H$, where $R \in \bar{K}(C)$ and $G, H \in \bar{K}[C]$. The divisor of R is defined as*

$$div(R) = \sum_{P \in C} [ord_P(R)]P \tag{2.20}$$

and also satisfies

$$div(R) = div(G) - div(H) \tag{2.21}$$

The order of the divisor of a rational function is, therefore, always equal to 0.

A divisor of a rational function is usually known as a *principal divisor*, defined formally as follows.

Definition 2.4.13. ***(principal divisor)*** *A divisor D is a principal divisor if it is a divisor of some rational function $R \in \bar{K}(C)$. All principal divisors are, therefore, of degree 0. The set of all principal divisors is written as $\mathbb{P}$.*

The set of degree zero divisors is written as $\mathbb{D}^0$. The set $\mathbb{P}$ forms an important subgroup of $\mathbb{D}^0$.

2.4.4 The Jacobian

The group on which the hyperelliptic curve discrete logarithm problem is constructed is known as the *Jacobian* of the curve.

Definition 2.4.14. ***(Jacobian)*** *The Jacobian of a hyperelliptic curve C is the quotient group of the degree zero divisors modulo the principal divisors, so that*

$$J_C = D^0/\mathbb{P} \tag{2.22}$$

The elements of the Jacobian are defined according to an *equivalence relation* between degree zero divisors, denoted $\sim$.

Definition 2.4.15. ***(equivalent divisors)*** *Consider two divisors $D_1, D_2 \in \mathbb{D}^0$ on a hyperelliptic curve. D_1 and D_2 are known as equivalent divisors if $D_1 - D_2$ is a principal divisor, i.e.*

$$D_1 \sim D_2 \text{ if } D_1 - D_2 \in \mathbb{P} \tag{2.23}$$

The Jacobian group is partitioned into subsets such that each subset contains divisors that are related to each other under the equivalence relation. These subsets are known as *equivalence classes*. All of the equivalence classes are cosets with respect to each other. The Jacobian is an *equivalence class group* under this partition. The equivalence classes can be represented using divisors with particular properties.

Definition 2.4.16. ***(semi-reduced divisor)*** *A semi-reduced divisor is a degree 0 divisor of form*

$$D = \sum m_i P - \sum m_i \infty \tag{2.24}$$

with the following properties:

1. *All $m_i \geq 0$ and all of the points must be finite.*
2. *If $m_i \neq 0$ at a point P and $P \neq -P$ then the divisor can contain either P or $-P$ but not both.*
3. *If a point $P = -P$ then $m_i \leq 1$ at that point.*

Every divisor in $\mathbb{D}^0$ has an equivalent semi-reduced divisor.

Definition 2.4.17. ***(reduced divisor)*** *A reduced divisor is a semi-reduced divisor that also satisfies the property $\sum m_i \leq g$, where g is the genus of the curve.*

Using the Riemann-Roch theorem [36], it can be shown that every equivalence class of the Jacobian contains exactly one reduced divisor that is unique to that class. Each class can, therefore, be represented by its own reduced divisor. The Jacobian forms an abelian group under the addition of these reduced divisors. This additive operation is very important in the context of curve-based cryptography.

Mumford [37] proposed a method for writing each reduced divisor in terms of two related polynomials. This representation is convenient for addition on the Jacobian.

2.4.5 Mumford Notation

From this point forward, discussion will be restricted to curves on finite fields since maintaining generality would result in additional complexity whilst providing no benefit. The hyperelliptic curve $C : y^2 + h(x)y = f(x)$ of genus g is now defined on a finite field $\mathbb{F}_q$ and has points $P = (x, y)$ with $x, y \in \bar{\mathbb{F}_q}$. Note that if a hyperelliptic curve is defined over $\mathbb{F}_q$ then it is also defined on all extensions of $\mathbb{F}_q$.

Mumford provides a way to represent reduced divisors as follows.

Definition 2.4.18. ***(Mumford notation)*** *All reduced divisors $D = \sum_{m_i} P_i - \sum_{m_i} \infty$ on a hyperelliptic curve $C : y^2 + h(x)y = f(x)$ can be represented by two polynomials $u(x)$ and $v(x)$ such that*

$$u(x) = \prod_i (x - x_i)^{m_i} \tag{2.25}$$

$$v(x_i) = y_i \tag{2.26}$$

where $P_i = (x_i, y_i)$. These polynomials must satisfy the following properties:

1. *u is monic*
2. *$deg(v) < deg(u) \leq g$*
3. *u divides $v^2 + vh - f$*

A divisor is written as $D = [u, v]$ when using Mumford notation.

Up until now the coordinates of the points have been defined on $\bar{\mathbb{F}}_q$. However, $\bar{\mathbb{F}}_q$ may be a very large field in comparison to $\mathbb{F}_q$. Fortunately, an automorphism can be used so that the coordinates can be defined on $\mathbb{F}_q$.

Definition 2.4.19. ***($\mathbb{F}_q$-rational divisor)*** *Consider a divisor $D = \sum_{m_P} P$ where $D \in \mathbb{D}^0$. D is said to be defined over $\mathbb{F}_q$ if, for all automorphisms σ of $\bar{\mathbb{F}}_q$ over $\mathbb{F}_q$, then $D^\sigma = \sum_{m_P} P^\sigma = \sum_{m_P}(\sigma(x), \sigma(y))$ where $D^\sigma = D$. D is known as an $\mathbb{F}_q$-rational divisor.*

On fields with suitable automorphisms, points with coordinates $(x, y) \in \mathbb{F}_q$ can be used to construct reduced divisors on the Jacobian. The Mumford polynomials are considerably smaller as a result.

An algorithm that uses Mumford notation to perform divisor addition on the Jacobian was proposed by Cantor in 1989 [36].

2.4.6 Cantor's Algorithm

Cantor's algorithm is used to add two reduced divisors on the Jacobian. Elements of the Jacobian form an abelian group under this operation. The algorithm is performed in two steps, respectively known as composition and reduction. Given the reduced divisors D_1 and D_2, a semi-reduced divisor D' is first calculated such that $D' \sim D1 + D2$. The divisor D' is then reduced to return $D_3 = D_1 + D_2$, where D_3 is a reduced divisor.

Given a hyperelliptic curve $y^2 + h(x)y = f(x)$, the algorithm begins with the composition step, which is described by Algorithm 1. For convenience, $f(x)$ and $h(x)$ are written as f and h. Note that steps 1 and 2 correspond to $d_1 \leftarrow gcd(u_1, u_2)$ and $d \leftarrow gcd(d_1, v_1 + v_2 + h)$ respectively.

The reduction step is then used to reduce the result and return $D_3 = D_1 + D_2$. This step is described by Algorithm 2.

Algorithm 1 Composition step of Cantor's algorithm

INPUT: Reduced divisors $D_1 = [u_1, v_1]$ and $D_2 = [u_2, v_2]$

OUTPUT: Semi-reduced divisor $D' = [u', v']$ such that $D' \sim D_1 + D_2$

1: $d_1 \leftarrow e_1 u_1 + e_2 u_2$
2: $d \leftarrow c_1 d_1 + c_2(v_1 + v_2 + h)$
3: $s_1 \leftarrow c_1 e_1, s_2 \leftarrow c_1 e_2, s_3 \leftarrow c_2$
4: $u' \leftarrow (u_1 u_2)/(d^2)$
5: $v' \leftarrow \big((s_1 u_1 v_2 + s_2 u_2 v_1 + s_3(v_1 v_2 + f))/d\big) \mod u'$

RETURN: $[u', v']$

Algorithm 2 Reduction step of Cantor's algorithm

INPUT: Semi-reduced divisor $D' = [u', v']$

OUTPUT: Reduced divisor $D_3 = [u_3, v_3]$

1: $u_3 \leftarrow (f - v'h - {v'}^2)/u'$
2: $v_3 \leftarrow (-h - v') \mod u_3$
3: **if** $deg(u_3) > g$ **then**
4: $\quad u' \leftarrow u_3, v' \leftarrow v_3$
5: $\quad$ Go to step 1
6: **end if**
7: Make u_3 monic by dividing it by its leading coefficient

RETURN: $[u_3, v_3]$

2.5 Elliptic and Genus 2 Hyperelliptic Curves

There is an attack, first described in [38], that can be used to compromise the security of some systems that use hyperelliptic curves. The time required by this attack decreases as the genus of the curve is increased. At the time of this research, attacks of this nature were considered very costly on curves of genus 1 and 2. These curves are used exclusively in this research for this reason. A more detailed overview of attacks on hyperelliptic curves is available in Section 2.6. Curves of genus 1 are more commonly known as elliptic curves. This section discusses these curves and their group operation. The use of elliptic and genus 2 hyperelliptic curves in cryptography are discussed in [39] and [40]. An introduction to the group operation on elliptic and hyperelliptic curves is provided by Sadanandan in [41].

2.5.1 Elliptic Curves

Definition 2.5.1. ***(elliptic curve)*** *An elliptic curve on a finite field $\mathbb{F}_q = \mathbb{F}_{p^m}$ is the set of points $P = (x, y)$ with $x, y \in \mathbb{F}_q$, together with a point at infinity ∞, that satisfy the equation:*

$$E(\mathbb{F}_q) : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.27}$$

where all $a_i \in \mathbb{F}_q$.

Definition 2.5.2. ***(opposite of a point)*** *The opposite of a point $P = (x, y)$ is $-P = (x, -y)$.*

Recall from Definition 2.4.17 that members of the Jacobian on a hyperelliptic curve C are reduced divisors of the form $D = \sum m_iP - \sum m_i\infty$, where $\sum m_i \leq g$. Since $g = 1$ for elliptic curves, reduced divisors only have one point in their support, i.e. $D = P - \infty$. Every member of the Jacobian can, therefore, be represented by a distinct point on $E(\mathbb{F}_q)$. Members of $E(\mathbb{F}_q)$ form a finite abelian group under the addition of these points. The identity element of this group is ∞, which exists at $(1, 0)$. This group is isomorphic to the Jacobian, which means that it can be used for cryptographic purposes. There exists a method for adding points on elliptic curves that is more efficient than Cantor's algorithm.

When used for cryptographic purposes, elliptic curve points are members of $E(F_q)$. The addition process can, however, be viewed more clearly when the elliptic curve is defined over the field of real numbers $\mathbb{R}$, as illustrated in Figure 2.4 (G. C. Kessler [3])

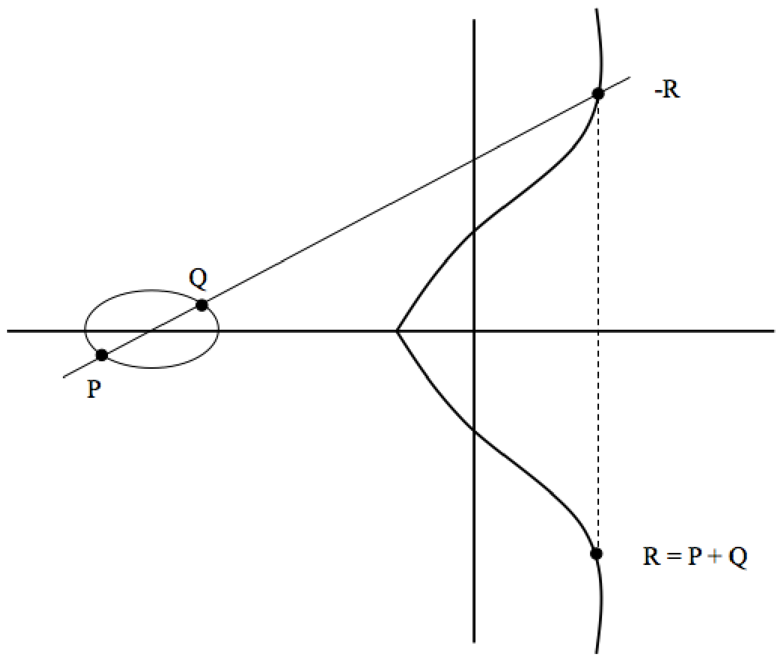

Figure 2.4: Elliptic Curve Addition over $\mathbb{R}$ (Source: G. C. Kessler [3])

Consider the addition $R = P+Q$, where $P, Q, R \in E(\mathbb{R})$ and $P \neq Q$. A function describing a straight line between P and Q is first defined. The line intercepts the curve at a third point, $-R$, the value of which can be retrieved by solving for the line function and curve equation. A vertical line function is then defined that describes a vertical line through $-R$. This line intercepts the curve at another point, which is R. This is the final result $R = P + Q$. Doubling is performed in a similar manner. The value of $S = P + P$, where $P, S \in E(\mathbb{R})$ is computed by first defining a function describing a tangent to the curve at P. This tangent intercepts the curve at a point $-S$. A vertical line function is defined at $-S$. The line function intercepts the curve at S, which is equal to $P + P$. The addition of a point to multiples of itself is known as *point scalar multiplication*. Doubling of a point is written as $[2]P$, while point tripling is written as $[3]P$, and so forth.

When an elliptic curves is defined on $\mathbb{F}_q$, all arithmetic required by the addition and doubling operations can be performed on $\mathbb{F}_q$. More details on the elliptic curve group operation are available in [39].

The security level of systems that employ elliptic and hyperelliptic curves is directly related to the order of the group in question. This order lies within the Hasse-Weil bound [40].

Definition 2.5.3. ***(elliptic curve group order, trace of Frobenius)*** *Consider an elliptic curve E defined on a finite field $\mathbb{F}_q$. The order of $E(\mathbb{F}_q)$ is given by*

$$\#E(\mathbb{F}_q) = q + 1 - t \tag{2.28}$$

where t is known as the trace of the Frobenius of the curve and $|t| \leq 2\sqrt{q}$.

For groups large enough to be of cryptographic interest, the value of $|t|$ is much smaller than q so $\#E(\mathbb{F}_q) \approx q$.

Subsets of points of the same order are known as sets of torsion points.

Definition 2.5.4. ***(set of n-torsion points on an elliptic curve)*** *The set of n-torsion points on an elliptic curve E is the set of points on the curve for which $[n]P = \infty$, where n is some positive integer. This set is written as $E[n]$.*

Supersingular curves are useful in the context of pairing-based cryptography as pairings can be computed very efficiently on them.

Definition 2.5.5. ***(supersingular elliptic curve, ordinary elliptic curve)*** *An elliptic curve $E(\mathbb{F}_q)$, where $q = p^m$, is supersingular if there are no points of order p on the curve, i.e. if $E[p]$ is empty. If this is not the case the curve is known as ordinary. The trace of Frobenius of a supersingular elliptic curve is always divisible by p.*

2.5.2 Genus 2 Hyperelliptic Curves

As seen in the previous section, a hyperelliptic curve of genus g is described by $C: y^2 + h(x)y - f(x) = 0$, where $h(x)$ is of degree less than or equal to g and $f(x)$ is monic and of degree $2g+1$. This means that in the genus 2 case the degree of $h(x)$ is either 1 or 2 and the degree of $f(x)$ is 5.

Members of the Jacobian on a genus 2 curve are represented by divisors of form $D = \sum m_i P - \sum m_i \infty$. Using Mumford notation, the divisors can be represented by polynomial pairs $[u, v]$ where u is monic, $deg(v) < deg(u) \leq 2$ and $u|(v^2 + vh - f)$ (from Definition 2.4.18). Cantor's algorithm can be used to perform addition on the Jacobian, as before. There is, however, a more efficient way to perform the group operation.

Consider the Mumford polynomial pairs $[u_1, v_1]$ and $[u_2, v_2]$ corresponding to, respectively, the reduced divisors D_1 and D_2 on the Jacobian. Addition can be performed efficiently by exploiting the relationships between the polynomials. Explicit addition formulae for the cases where $\{deg(u_1) = 0\}$, $\{deg(u_1) = 1, deg(u_2) = 1\}$, $\{deg(u_1) = 1, deg(u_2) = 2\}$ and $\{deg(u_1) = 2, deg(u_2) = 2\}$ can be derived. Note that the degrees of the v polynomials are fixed by their relationships to the u polynomials. Harley provides explicit formulae for odd characteristic genus 2 curves in [42]. Some performance results are published in [43]. In [44], Lange derives formulae for even characteristic curves. The most common case, in which $\{deg(u_1) = 2, deg(u_2) = 2\}$ requires two inversions, three squarings and 24 multiplications on $\mathbb{F}_q$. This performance is much better than the average number of operations required by Cantor's algorithm. Addition formulae do not have to be employed exclusively. A good compromise is to use the explicit addition formulae for the most common cases and to employ Cantor's algorithm otherwise.

In [45], Lange further refines formulae for characteristic 2 and 3 curves using *affine*, *projective* and *weighted projective* coordinates. Points are represented by (x, y) pairs in an affine coordinate system. The explicit formulae require multiplications, squarings, additions and one inversion. Finite field inversion can be costly in comparison to other field operations. A projective coordinate system employs (x, y, z) triples. While inversion is not required, more multiplications must be performed. The extra coordinate also increases the complexity of the group arithmetic. Lange introduces a weighted projective system that provides a comparably efficient way to perform doubling on the Jacobian. An affine coordinate system is, however, used in this research. The introduction of a third coordinate is not justified since inversions can be performed relatively quickly using a dedicated hardware component.

The Hasse-Weil bound in the genus 2 case states that the order of the Jacobian is restricted to

$$(\sqrt{q}-1)^4 \leq \#J \leq (\sqrt{q}+1)^4 \tag{2.29}$$

In practice, this means that $\#J \approx q^2$. Recall that on elliptic curves the Jacobian is of order q. This means that if a field size of order q is used in the elliptic case then genus 2 curves can be defined on a field of order $\sqrt{q}$, while maintaining the same level of security. This reduction in the underlying field size results in faster field arithmetic and reduced storage requirement. The benefit that this provides may, however, be offset by the cost of a more complicated Jacobian addition in the genus 2 case.

The set of n-torsion divisors on a genus 2 curve can be defined in the same manner as in the elliptic case.

Definition 2.5.6. ***(set of n-torsion divisors on genus 2 curves)*** *The set of n-torsion divisors on the Jacobian, J, of a genus 2 hyperelliptic curve is the set of degree 0 divisors on J for which $[n]D = 0$, where n is some positive integer. The set is written as $J[n]$.*

Pairings can be calculated very efficiently on certain supersingular genus 2 hyperelliptic curves.

Definition 2.5.7. ***(supersingular genus 2 curve)*** *A supersingular genus 2 hyperelliptic* *curve has no divisors of order p, where p is the characteristic of the finite field on which the curve is defined.*

2.6 Hyperelliptic Curves in Cryptography

The cryptographic use of hyperelliptic curves is discussed in this section. Known attacks on curve-based schemes that use group addition are described. The properties of fields and curves that should be avoided to prevent the success of these attacks are outlined. The rationale for the cryptographic use of hyperelliptic curves is also discussed. Finally, the *Elgamal Encryption system* [46], when implemented using an elliptic curve, is described in detail to provide some context to the topics discussed in this chapter. An overview of curve-based cryptography is available in [39]. An introduction to the security issues relating to curve-based cryptography is provided by Scholten and Vercauteren in [47].

The *Computational Diffie-Hellman Problem* (CDH) can be used to construct many schemes based on finite fields, elliptic curves and hyperelliptic curves.

Definition 2.6.1. ***(Computational Diffie-Hellman Problem)*** *Consider a group G of order n with generator g. Let a, b be positive integers smaller than n and $g^a, g^b \in G$. The Computational Diffie-Hellman Problem is: given g, g^a, g^b, find the element $h \in G$ such that $h = g^{ab}$.*

The difficulty of the CDH is reliant on the difficulty of the *Discrete Logarithm Problem* in the group in question.

Definition 2.6.2. ***(Discrete Logarithm Problem (DLP))*** *Consider a group G of order n with generator g. Let $h = g^a$, where $h \in G$ and a is a positive integer smaller than n. The Discrete Logarithm Problem is the problem of finding the value of a, given g and h.*

The CDH can be solved if the DLP is solved in the group. It is shown in [48] that the CDH and the DLP have the same complexity in a group of prime order. They also have the same complexity in the relevant groups on elliptic and hyperelliptic curves. Attacks on the CDH come in the form of attacks on the DLP of the group in question. An overview of the CDH and its intractability is available in [39].

Attacks on the DLP become more difficult as the group order is increased. To provide adequate security for modern cryptographic applications the size of a finite field must be in the order of several thousand bits when the DLP is built on the field alone. Computations on fields of this size are too costly and too slow to be of benefit in the majority of systems. Attacks on the DLP of groups on elliptic and hyperelliptic curves are, however, more difficult. This means that a smaller underlying finite field size can be used to provide the required security level.

2.6.1 Security Considerations

Some good overviews of the subject matter of this subsection can be found in [40], [47], [30] and [49].

The following terms are used to define the time requirement of an attack on the DLP or its curve-based extensions:

- **Polynomial:** The required time is polynomial in the number of digits in the group order.
- **Exponential:** The required time is exponential in the number of digits in the group order.
- **Sub-exponential:** There is no agreed definition of sub-exponential time. In general terms, a problem with sub-exponential properties is still considered intractable but

not as difficult as a problem requiring exponential time.

The Discrete Logarithm Problem (DLP)

Consider a group with an order of n bits that has members g and h such that $h = g^a$. The most basic attack on the DLP is known as a *brute force attack*. This involves guessing values of a until a solution to the problem is found. This attack requires a running time of $O(n)$ and is, therefore, exponential in nature.

There is a family of generic algorithms that can be used to attack the DLP when implemented on any group. This includes groups defined on curves. One such attack uses the *baby-step giant-step* algorithm [50], usually in combination with a lookup table, to perform a modified brute force attack. Pollard's rho algorithm [51] returns a result in similar time, but with a smaller memory requirement. In [52], Shoup proves that all generic attacks on groups of prime order require a running time of $O(\sqrt{n})$. It should be noted that even if the order of the group is not prime, the *Pohlig-Helman* algorithm [53] can be used to reduce the running time to $O(\sqrt{p})$, where p is the number of digits in the largest prime factor of the group order. This means that generic attacks are exponential in the number of digits in the prime order, or in the number of digits in the largest prime factor of the order. The DLP should therefore be constructed on a group with large prime order or on a group with an order that has a large prime factor.

The DLP, when implemented on the multiplicative groups of finite fields, is susceptible to the *Index Calculus Algorithm* [54]. This algorithm exploits relationships between small prime members of the field and returns a result in probabilistic sub-exponential time. The order of the group should be sufficiently large to render this attack infeasible. The brute force, generic and index calculus attacks mean that multiplicative finite field must be of very large order.

The Elliptic Curve Discrete Logarithm Problem (ECDLP)

The ECDLP was independently suggested for cryptographic use by Miller [19] and Koblitz [20] in 1985.

Definition 2.6.3. ***(Elliptic Curve Discrete Logarithm Problem(ECDLP))*** *Consider an elliptic curve $E(\mathbb{F}_q)$ containing a point P of order l. Let $< P >$ be the cyclic subgroup of points generated by P. Let $Q = [a]P$, where $1 \leq a < l$ and $Q \in < P >$. The elliptic curve discrete logarithm problem is the problem of finding the value of a given P and Q.*

The point $Q = [a]P$ is the result of the addition of P to itself a times. This is known as point scalar multiplication.

The index calculus attack cannot be applied to elliptic curves. This means that a much smaller field size can be used to provide the same security as systems based on the DLP alone. There are, however, certain curves that should be avoided since their properties introduce a susceptibility to other attacks.

Anomalous elliptic curves have a trace of Frobenius that is equal to 1. An anomalous curve $E(\mathbb{F}_q)$, where $q = p^m$, has exactly p^m points. In [55], it is shown that, in this case, the ECDLP can be reduced to the DLP in an additive group on $\mathbb{F}_p$. The DLP in this group is trivial and can be solved in linear time. All anomalous elliptic curves should be avoided.

Menezes et al. [56] describe an attack, known as the MOV reduction, that reduces the ECDLP to the DLP on some extension of the underlying finite field on which the elliptic curve is defined. The dimension of this extension is known as the *embedding degree* of the curve (see Definition 2.6.5 for the definition of embedding degree in the general hyperelliptic case, which holds here). The index calculus attack can then be used on the DLP of the extension field and, if its order is insufficient, a successful attack may be feasible.

An insignificant subset of elliptic curves have an embedding degree that is small enough to introduce vulnerability to a MOV reduction. Since it is necessary to randomly select a curve only during the construction of a system or at the beginning of a particular session, the MOV reduction is, in general, not a troublesome problem.

Supersingular elliptic curves are vulnerable to MOV reduction since they have a maximum embedding degree of 6. These curves should not be used in DLP systems unless this vulnerability is considered. Supersingular curves do, however, have properties that allow for efficient pairing computation on them. Many novel pairing-based schemes have been devised that use supersingular curves in combination with other difficult problems. The calculation of pairings on supersingular curves, and their use in cryptographic schemes, is discussed in the next chapter.

The Hyperelliptic Curve Discrete Logarithm Problem (HCDLP)

The HCDLP was generalised to groups on hyperelliptic curves by Koblitz in 1989 [21].

Definition 2.6.4. ***(Hyperelliptic Curve Discrete Logarithm Problem (HCDLP))*** *Consider a hyperelliptic curve $C(\mathbb{F}_q)$ with Jacobian J and $D_1 \in J$. Let D_2 be a member of $< D_1 >$, where $< D_1 >$ is the cyclic subgroup of divisors in J generated by D_1. The Hyperelliptic Curve Discrete Logarithm Problem is, given D_1 and D_2, finding the positive integer $a < (\#J)$ such that $D_2 = [a]D_1$.*

An Index Calculus attack on hyperelliptic curves, faster than Pollard's rho algorithm when the genus of the curve is larger than 2, is described by Gaudry et al. in [57]. Curves with genus larger than 2 should, therefore, be avoided.

The Pohlig-Hellman algorithm can be used to reduce the DLP on the Jacobian to the DLP on a group with order equal to the largest prime divisor of $\#J$. This means that $\#J$ should be prime, or the largest prime divisor of the group order should be sufficiently large, to avoid vulnerability to this attack.

If the largest prime divisor, r, of $\#J$ is equal to the characteristic of $\mathbb{F}_q$ then the curve is anomalous and the DLP is trivial in the additive group of $\mathbb{F}_q$, as in the elliptic case.

The MOV reduction was generalised from elliptic curves to hyperelliptic curves of arbitrary genus by Frey and Rück in [58]. In this case, the Tate pairing is used to perform a reduction from the HCDLP to the DLP on a k-dimensional extension of the finite field, where k is the embedding degree of the hyperelliptic curve.

Definition 2.6.5. ***(embedding degree)*** *Consider a hyperelliptic curve C with a divisor group of order r on J. The embedding degree of C is the smallest value of k such that $r \mid (q^k - 1)$.*

If the embedding degree is sufficiently small an Index Calculus attack on the DLP in the multiplicative group $\mathbb{F}^*_{q^k}$ may be possible. Fortunately, very few genus 2 curves have a value of k that is small enough to be problematic. As in the elliptic case, however, supersingular hyperelliptic curves have low embedding degree. Genus 2 curves defined on fields of characteristic 2, for example, have a maximum embedding degree of 12. These curves are, however, again useful in the implementation of pairing-based cryptography, as will be seen in the next chapter.

2.6.2 The Benefits of Hyperelliptic Curve Cryptography

The benefits of the cryptographic use of elliptic and genus 2 hyperelliptic curves are discussed in this subsection. In public-key schemes based on the DLP (and the ECDLP and the HCDLP) the keys are members of the finite field upon which the problem is constructed. This means that the keys are the same size as the order of the finite field in use. The difficulty of the DLP on the multiplicative group of a finite field has the same complexity as the problem of integer factorisation, which is used by conventional public-key schemes such as RSA. This means that, for example, a conventional public-key scheme with 1024-bit keys corresponds to a DLP scheme with a finite field size with a 1024-bit order.

It was shown in Section 2.5 that, on elliptic curves, the group operation is performed on the group of points in $E(\mathbb{F}_q)$, where $\#E(\mathbb{F}_q) \approx q$. The Index Calculus attack is not problematic in the elliptic case. This means that a smaller field size can be used to provide the same level of security as conventional public-key schemes. In the genus 2 case, the Hasse-Weil bound states that the Jacobian group, on which the HCDLP is constructed, has order $\#J \approx q^2$. The larger group order means that the curve can be defined on an even smaller finite field than in the elliptic case if the genus 2 curve is carefully chosen.

In [49], the times required by successful attacks on the DLP, the ECDLP and the HCDLP are calculated and used to compare the field sizes required for various levels of security. These are compared with each other and also with the key sizes required by conventional public-key schemes (such as RSA) in Table 2.1.

Table 2.1: Comparison of key sizes (in bits) required by conventional public-key schemes and public-key schemes using the DLP, the ECDLP and the HCDLP

Conventional	**DLP**	**ECDLP**	**HCDLP**
1024	1024	174	87
2048	2048	234	117
4096	4096	313	157
8192	8192	417	209

It can be seen that elliptic and genus 2 implementations can provide a large reduction in key size when compared to conventional schemes, resulting in a reduced bandwidth requirement. Arithmetic on the smaller fields is less expensive. Furthermore, if an increase in security level is necessary, the key sizes required by curve-based solutions grow at a smaller rate than systems based on RSA or the DLP on the multiplicative groups of finite fields. The genus 2 case holds an added advantage over the elliptic case in this respect.

As the security level is increased, the key length of a genus 2 implementation increases at a smaller rate than in the elliptic case.

The advantages of the smaller finite field sizes of the ECDLP and HCDLP are, however, offset by the more complicated group operations of the elliptic and hyperelliptic curves. A hardware implementation can provide a very favourable solution to this problem since many of the group operations can be performed in parallel, if carefully scheduled. The extension field computations can also be implemented in terms of sub-field operations, which can be performed in parallel in hardware. This can result in a large reduction in computation time.

2.6.3 Protocol Example: The Elgamal Encryption Scheme

Many cryptographic systems based on the DLP can be implemented using elliptic and hyperelliptic curves. Several of these schemes are discussed in [39]. The *Elgamal Encryption System* [46] is an asymmetric public-key encryption scheme. Security is dependent on the intractability of the CDH in the group on which it is constructed. It was adapted for implementation on elliptic curves by Koblitz in 1987 [20]. In the following example, the Elgamal encryption system is implemented on an elliptic curve. It can also be implemented on the Jacobian of genus 2 hyperelliptic curves.

The following steps must be followed if Alice wishes to securely send a message, denoted m, to Bob using the Elgamal Encryption System.

Setup

- A suitable elliptic curve, E, defined on a finite field, $\mathbb{F}_q$, is selected. A point $P = (x_p, y_p)$, of order n, is chosen as the generator of a cyclic subgroup $< P >$. The security of the cryptosystem does not depend on these parameters and they can be made available to all users.

- Bob chooses a random positive integer k_b, where $1 \leq k_b < n$. This is his private key and is not shared. He then calculates and publishes $Q = [k_b]P$. The point Q is his public key and is available to Alice.

Encryption

- Alice first selects a random positive integer k_a, $1 \leq k_a < n$.
- She calculates $R = [k_a]P$ and $S = [k_a]Q$, where $S = (x_s, y_s)$.
- Alice then calculates $t = x_s.m \mod q$, where m is the message to be sent.
- Alice sends the ciphertext tuple (R, t) to Bob.

Decryption

- Bob calculates $U = [k_b]R$, where U has coordinates (x_u, y_u).
- Bob retrieves the message by calculating $m = t/x_u \mod q$.

The Elgamal cryptosystem is viable since the calculations required of Alice and Bob are relatively easy to perform. An eavesdropper would, however, have to solve the ECDLP to decrypt the message. The ECDLP in this group should sufficiently difficult so that such an attack is infeasible.

2.7 Conclusions

The mathematics underpinning curve-based cryptography have been introduced in this chapter. The theory that is required for an understanding of the subject matter of this thesis is summarised here for clarity. The computation of arithmetic on finite fields is particularly important as all operations required for pairing-based cryptography are performed on members of abelian groups on these fields. The group on which the hyperelliptic curve discrete logarithm problem is based is known as the *Jacobian*. The *set of divisors*, which are finite formal sums of points on the curve, form an abelian group under *the addition law of the Jacobian*. These divisors can be represented by a set of two polynomials using *Mumford Notation*. Divisors on the Jacobian of elliptic curves have only one point in their support, which means that the abelian additive group on the Jacobian consists of points. This significantly reduces the complexity of the group operation. The Jacobian of genus 2 hyperelliptic curves contains divisors with at most two points in their support, complicating the group operation. Addition can be performed using *Cantor's algorithm*, explicit formulae, or a combination of the two.

Attacks on the security of schemes that employ these curves have been outlined and the removal of vulnerabilities through appropriate field and curve selection discussed. It has been shown that curve-based implementations of cryptographic schemes require smaller key sizes, when compared to other similar public-key systems. The use of hyperelliptic curves can also offer several advantages if a cryptographic scheme must be implemented in a constrained environment, i.e. quicker computation of finite field arithmetic and a reduced memory requirement. FPGAs are suitable for such an implementation since, if a change in the security level of the scheme is required, a simple reprogramming can be performed.

The next chapter introduces the theory of mathematical pairings on hyperelliptic curves and the rest of this thesis is concerned with their efficient computation. This chapter has provided a foundation for that work. It must be noted, however, that the hardware to be discussed in later chapters can easily be adjusted to implement the more conventional curve-based schemes that have been described up to this point.

Chapter 3

The Tate and η_T Pairings

3.1 Introduction

In 1994, bilinear pairings were introduced to cryptography in the form of attacks on the security of certain curves. The MOV reduction [56] employed the Weil pairing whilst the Frey-Ruck (FR) attack [58] used the Tate pairing to reduce the DLP on an elliptic or genus 2 hyperelliptic curve to the DLP on an extension of the finite field on which the curve is defined. *Supersingular curves* are particularly vulnerable to these attacks and their use was avoided for a time. Supersingular curves are, however, suited to schemes that use pairings in a constructive manner since pairings can be computed very efficiently on them. The Tate pairing is usually used in cryptographic applications as it can be computed faster than the Weil pairing (the Weil computation requires two applications of an algorithm that can itself be used once to compute the Tate pairing).

As discussed in Subsection 2.2.3, the first constructive use of pairings was described by Joux in 2000 [24]. A one round tripartite key agreement scheme, devised using the bilinearity of pairings, is described. In 2001, Boneh and Franklin [26] devised an IBE scheme that was made possible by pairings. Since then the research area has garnered much attention and many optimisations for pairing computation have been suggested, the most important of which are briefly outlined here.

As will be seen in this chapter, pairings are computed by performing finite field operations within an iterative loop. This is followed by an exponentiation to ensure that a unique result is returned. The loop is constructed using Miller's algorithm [19], [59]. In 2002, Barreto et al. [60] provided efficient algorithms for pairing computation on characteristic 3 elliptic curves. Prior to this, pairings were performed on members of extensions of the finite fields on which the relevant curves were defined. They show that if one of the pairing inputs is restricted to the base field, then many computations required during the Miller loop can be avoided. Galbraith et al. made similar observations in [61]. They also provide efficient computation methods for arithmetic on characteristic 2 and 3 extension fields.

In 2003, Duursma and Lee [62] described several optimisations for a subset of hyperelliptic curves with points on $\mathbb{F}_{p^m}$, where $p > 2$. In 2007, Barreto et al. [63] generalised these optimisations to a large number of characteristic 2 and 3 elliptic curves and to characteristic genus 2 hyperelliptic curves. They define a notation for the use of these optimisations and show that a bilinear pairing can be computed using a Miller's loop with half as many iterations as previously required. This method does not return a Tate pairing but has all the properties required for use in pairing-based cryptosystems. The pairing is known as the truncated pairing, denoted η_T. It is closely related to the Tate pairing and a conversion can be performed by exponentiating the η_T result. Tate pairing computation using the η_T method returned the fastest software result at the time of this research, and the rest of this thesis discusses its implementation.

It should be noted that this research does not consider pairings on curves that are defined over fields of large prime characteristic. A review of the optimisations to pairing computation on these curves is, however, provided in Section 7.1.

The Tate pairing is defined and some relevant properties outlined in Section 3.2. Computation using Miller's Algorithm is described. Further optimisations to Tate pairing computation are also discussed. The η_T pairing is introduced in Section 3.3. A discussion on the use of pairings in cryptography is provided in Section 3.4. The Boneh-Franklin identity-based encryption scheme [26] is outlined to provide context to the subject matter of this chapter. The methodology for the hardware implementation of the Tate pairing in this work is described in Section 3.5. An automated design environment has been created to facilitate the rapid design and verification of hardware pairing processors and will also be discussed.

The η_T is an optimisation of methods previously used for Tate pairing calculation. The properties of certain groups and curves are exploited to return a very fast computation. Although the pairings are closely related, the use of some optimisations means that the results returned by the η_T are not the same as those returned by the Tate pairing. An exponentiation of the η_T result to a Tate pairing value can, however, be performed. A system that has been created to aid in the design and verification of the processors is also presented.

The Master's thesis of Maas [30] and the Ph.D. thesis of Lynn [64] provide good overviews of pairing-based cryptography. El Mrabet analyses efficient computation methods for pairing-based cryptography in [65].

The mathematical foundations of the Tate and η_T pairings are discussed in this chapter. Cryptography based on pairings is also discussed. The steps required to perform the Boneh Franklin identity-based encryption scheme [26] are described. The methodology and equipment that have been used for the hardware implementation of the Tate pairing is described.

3.2 The Tate Pairing

3.2.1 Definition

Let C be a hyperelliptic curve on $\mathbb{F}_q$ and let $J_C(\mathbb{F}_q)$ be the Jacobian of C. Let n be a large prime such that $n \mid \#J_C(\mathbb{F}_q)$. The property $gcd(n, q) = 1$ must hold to avoid the attack described by Rück in [66]. Also, n^2 must not divide $\#J_C(\mathbb{F}_q)$ so that the Pohlig-Hellman attack [53] cannot be used. Let k be the smallest integer that satisfies $n \mid q^k - 1$ (this is the embedding degree of the curve, as discussed in Section 2.6).

Consider a group containing the n-th multiples of all divisors in $J_C(\mathbb{F}_{q^k})$. This group is written as $nJ_C(\mathbb{F}_{q^k}) = \{[n]D \mid D \in J_C(\mathbb{F}_{q^k})\}$. The quotient group $J_C(\mathbb{F}_{q^k})/nJ_C(\mathbb{F}_{q^k})$ forms an equivalence class under an equivalence relation. Members D and D' are related under the equivalence $D \sim D'$ if $(D - D') \in nJ_C(\mathbb{F}_{q^k})$. Let $J_C(\mathbb{F}_{q^k})[n]$ be the n-torsion group of divisors in $J_C(\mathbb{F}_{q^k})$. Consider a divisor $D_1 \in J_C(\mathbb{F}_{q^k})[n]$. There exists a function f_{n,D_1} such that $div(f_{n,D_1}) = [n]D_1$. Let $D_2 \in J_C(\mathbb{F}_{q^k})/nJ_C(\mathbb{F}_{q^k})$.

Definition 3.2.1. ***(Tate pairing)*** *The Tate pairing is defined as*

$$\langle D_1, D_2\rangle_n = f_{n,D_1}(D_2) \tag{3.1}$$

and is a mapping

$$\langle ., .\rangle_n : J_C(\mathbb{F}_{q^k})[n] \times J_C(\mathbb{F}_{q^k})/nJ_C(\mathbb{F}_{q^k}) \rightarrow \mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^n \tag{3.2}$$

where $\mathbb{F}^*_{q^k}$ is an abelian multiplicative group and $\mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^n$ is an equivalence class group. Consider group members a and b, where $a, b \in \mathbb{F}^*_{q^k}$. The element a is equivalent to b only if $a/b \in (\mathbb{F}^*_{q^k})^n$. This means that $a \sim b$ if $a = bc^n$ for some element $c \in F^*_{q^k}$.

Note that the divisors D_1 and D_2 must be linearly independent so that the pairing result is non-trivial, i.e. D_1 and D_2 must have no common points.

The result of the pairing does not depend on the choice of D_2 since the only impact of its value is in the definition of the representation for the equivalence class. For simplicity, this means that D_2 can be a member of $J_C(\mathbb{F}_{q^k})$. With this information, the Tate pairing is now a mapping

$$\langle ., . \rangle_n : J_C(\mathbb{F}_{q^k})[n] \times J_C(\mathbb{F}_{q^k}) \rightarrow \mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^n \tag{3.3}$$

The Tate pairing does not return a unique value, which is required for cryptographic purposes. Members of the quotient group $\mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^n$ are defined up to a multiple of the n-th power of $c \in \mathbb{F}^*_{q^k}$. Since a unique result is required, the n-th powers must be eliminated. The relationship $c^{q^k-1} = 1$ holds due to the cyclical nature of the multiplicative group. The multiples can, therefore, be removed by exponentiating the pairing result to the power $(q^k - 1)/n$. The output of this exponentiation is a unique element of $\mathbb{F}^*_{q^k}$ of order n. The calculation of the Tate pairing followed by this exponentiation, known as the *final exponentiation*, is known as the *reduced Tate pairing*.

Definition 3.2.2. ***(reduced Tate pairing)*** *The reduced Tate pairing is defined by*

$$e(D_1, D_2) = \langle D_1, D_2 \rangle_n{}^{(q^k-1)/n} = f_{n,D_1}(D_2)^{(q^k-1)/n} \tag{3.4}$$

where the unique values returned by the pairing are members of $\mu_n = \{\mu^n = 1 \mid \mu \in \mathbb{F}^*_{q^k}\}$*, which is the group of* n*-th roots of unity of* $\mathbb{F}^*_{q^k}$*.*

The reduced Tate pairing satisfies the following useful properties:

- **Bilinearity:** For any integers s, t

$$e([s]D_1, [t]D_2) = e([t]D_1, [s]D_2) = e(D_1, D_2)^{st} \tag{3.5}$$

- **Non-degeneracy:** For each divisor $D_1 \in J_C(\mathbb{F}_{q^k})[n]$, where $D_1 \neq 0$, there exists a divisor $D_2 \in J_C(\mathbb{F}_{q^k})$ such that $e(D_1, D_2) \neq 1$.

- **Compatibility:** Let $N = sn$ for any integer s. Then

$$e(D_1, D_2) = \langle D_1, D_2 \rangle_n{}^{(q^k-1)/n} = \langle D_1, D_2 \rangle_N{}^{(q^k-1)/N} \tag{3.6}$$

3.2.2 Computation Using Miller's Algorithm

In the literature, the Tate pairing was first computed using Miller's algorithm [59, 67]. This algorithm does not return a reduced pairing value and must be followed by a final exponentiation in cryptographic applications that use it.

Consider the divisors $D_3 \in J_C(\mathbb{F}_{q^k})$ and $D_4 \in J_C(\mathbb{F}_{q^k})$. Let D_5' be the non-reduced divisor returned by adding D_3 and D_4 using the Jacobian group law (or point addition in the elliptic case) and let D_5 be the final reduced divisor. If addition is performed using Cantor's algorithm, for example, then D_5' is the output of the composition stage and D_5 is the output of the reduction stage. A function g_{D_3,D_4} always exists such that

$$div(g_{D_3,D_4}) = div(c_{D_3,D_4}) - div(d_{D_3,D_4}) = D_5' - D_5$$

where c and d are rational functions on the curve.

From Equations (3.1) and (3.3) the Tate pairing is calculated according to $\langle D_1, D_2\rangle_n = f_{n,D_1}(D_2)$, where $D_1 \in J_C(\mathbb{F}_{q^k})[n]$, $D_2 \in J_C(\mathbb{F}_{q^k})$ and f_{n,D_1} is a function such that $div(f_{n,D_1}) = [n]D_1$. For pairing computation it is not necessary to define the function f_{n,D_1}. The only result that is of importance is the evaluation of f_{n,D_1} at D_2. This can be calculated by defining a series of simpler, intermediate functions arising from the addition of D_1 to multiples of itself and accumulating the evaluations of these functions at D_2. The following relationship is used to define the intermediate functions and forms the basis for Miller's algorithm

$$f_{i+j,D_1} = f_{i,D_1} . f_{j,D_1} . g_{[i]D_1,[j]D_1} \tag{3.7}$$

where g is a function that equals c/d and c and d are the rational functions arising from the addition process. Note that $div(g) = div(c) - div(d) = div(c/d)$.

Miller's algorithm takes the form of an iterative loop. A binary *double-and-add* method can be used to add multiples of D_1 to itself on each iteration. The c and d functions related to these additions are then defined and evaluated at D_2. The accumulating function $f \in \mathbb{F}_{q^k}$

is updated according to Equation (3.7). This process is described by Algorithm 3 in the general hyperelliptic case.

Algorithm 3 Tate pairing computation using Miller's algorithm

INPUT: $D_1 \in J_C(\mathbb{F}_{q^k})[n], D_2 \in J_C(\mathbb{F}_{q^k})$

OUTPUT: $\langle D_1, D_2 \rangle_n$

Initialise: $f \leftarrow 1$, $D \leftarrow D_1$

Let $n = \sum_{i=0}^{s} n_i 2^i$, where s is the number of bits in n, $n_i \in \{0, 1\}$ and $n_s = 1$.

for $(i \leftarrow s-1,\ i \geq 0,\ i \leftarrow i-1)$ **do**

    $D \leftarrow [2]D$ and define $g_{D,D}$ during the doubling

    $f \leftarrow f^2.g_{D,D}(D_2)$

    if $n_i = 1$ **then**

        $D \leftarrow D + D_1$ and define g_{D,D_1} during the addition

        $f \leftarrow f.g_{D,D_1}(D_2)$

    end if

end for

RETURN: f

In the elliptic case addition is performed on the group of points on $E(\mathbb{F}_{q^k})$. Consider the addition of P and Q, where $P, Q \in E(\mathbb{F}_{q^k})$. The intermediate function $g_{P,Q}$ is related to the lines used in the addition process. The function $c_{P,Q}$ describes the straight line through P and Q and $d_{P,Q}$ describes the vertical line through $P + Q$. In the genus 2 case, either Cantor's algorithm or explicit formulae can be used to perform the required Jacobian addition. The rational functions are defined in terms of intermediate divisors, which are represented by Mumford polynomials. The functions are evaluated at points in the elliptic case. They are evaluated at divisors in the general genus 2 case. They can, however, be evaluated at points in certain circumstances (as discussed in the next subsection). These evaluations can also be incorporated into the addition process to reduce computation time [68], [69].

Miller's algorithm can be computationally expensive when working with large groups. Many additions and doublings are necessary. The multiplication of the accumulating $\mathbb{F}_{q^k}$ element by (c/d) on each iteration requires multiplication and inversion on $\mathbb{F}_{q^k}$, which are expensive operations. An implementation using the algorithm described in this subsection may not be suited to many applications as a sufficiently fast pairing computation may not be possible. There are, however, several methods that can be used to reduce pairing computation time.

3.2.3 Optimisations to Tate Pairing Computation

This section describes various optimisations for Tate pairing computation that are relevant to this research. A more detailed overview of many of the optimisations is available in [68].

The first input to the pairing can be defined on $J_C(\mathbb{F}_q)[n]$ rather than $J_C(\mathbb{F}_{q^k})[n]$, without a loss in security [70]. However, if the second input $D_2 \in J_C(\mathbb{F}_{q^k})$ is also a member of $J_C(\mathbb{F}_q)$ then the non-degeneracy property of the Tate pairing will not be satisfied. To ensure that this does not happen a non-$\mathbb{F}_q$-rational endomorphism can be used to generate the second input. This endomorphism, also known as a *distortion map*, ψ, exists on all supersingular curves [71] and proves very beneficial for pairing computation. Using this function a divisor on $J_C(\mathbb{F}_q)$ can be mapped to another on $J_C(\mathbb{F}_{q^k})$ such that the resultant divisor cannot be a member of the original field. A pairing that uses a distortion map is described as *modified*.

Definition 3.2.3. ***(modified Tate pairing)*** *Let $D_1 \in J_C(\mathbb{F}_q)[n]$ and $D_2 \in J_C(\mathbb{F}_q)$. The modified Tate pairing is given by*

$$\langle D_1, \psi(D_2)\rangle_n = f_{n,D_1}(\psi(D_2)) \tag{3.8}$$

The modified pairing, followed by a final exponentiation, is known as the *reduced modified Tate pairing.*

Definition 3.2.4. ***(reduced modified Tate pairing)*** *Let $D_1 \in J_C(\mathbb{F}_q)[n]$ and $D_2 \in$*

$J_C(\mathbb{F}_q)$. *The reduced modified Tate pairing is given by*

$$\hat{e}(D_1, D_2) = \langle D_1, \psi(D_2)\rangle_n^{(q^k-1)/n} = f_{n,D_1}(\psi(D_2))^{(q^k-1)/n} \tag{3.9}$$

In 2002, Barreto et al. [60] and Galbraith et al. [61] described several optimisations for Tate pairing computation in the elliptic case. Many of these improvements are based on similar observations and are discussed in this section.

A division on $\mathbb{F}_{q^k}$ is required on every iteration of the loop in Miller's algorithm. Each division consists of an extension field inversion followed by a multiplication. Extension field inversions require many expensive operations and should be avoided when possible. The divisions can, however, be replaced by multiplications on each loop iteration. The accumulating function f is first written as a quotient of two functions, $f = f_1/f_2$. The new functions can be independently updated on each iteration of the loop and each accumulated in the same manner as in Algorithm 3. The function f can then be computed on loop exit by dividing the final value of f_1 by the final value of f_2.

Consider a reduced modified Tate pairing $\langle P, \psi(Q)\rangle_n^{(q^k-1)/n}$ where $P \in E(\mathbb{F}_q)[n]$ and $Q \in E(\mathbb{F}_q)$. If a distortion map is available such that the x-coordinate of $\psi(Q)$ is defined on a sub-field of $\mathbb{F}_{q^k}$, then the operations required to compute the denominators of the intermediate functions of Miller's algorithm are not required. The reason for this is that they are eliminated by the final exponentiation. This means that the vertical line functions do not need to be evaluated on each loop iteration.

There is a very efficient point tripling operation on curves of characteristic 3. Tripling can be performed in $O(m)$ steps, whereas a doubling requires $O(m^2)$ steps in characteristic 3. The accumulating function can be calculated using a triple and add algorithm to take advantage of this.

The Hamming weight of n has a direct effect on the number of required operations. If n can be chosen so that it has low Hamming weight then the number of required point additions can be significantly reduced. Note that in this work, standard methods are used to return sparse Hamming weights. Investigations into possible mathematical optimisations for the reduction of Hamming weight are beyond the scope of this research.

In 2003, Duursma and Lee [62] published a paper that describes several improvements to Tate pairing calculation. These new optimisations can be applied to hyperelliptic curves of form $C(\mathbb{F}_q) : y^2 = x^p - x + d$, where $q = p^m$ and $d = \pm 1$. The conditions $p \geq 3$, $p \equiv 3 \mod 4$ and $gcd(m, 2p) = 1$ must be met. These types of curves have embedding degree $k = 2p$. This relationship enables some of the optimisations. Many of the methods described in the paper hold for $p > 3$, but the authors concentrate much of their effort on the characteristic 3 case. Their observations result in a very efficient pairing computation on these curves. The hardware implementations described in this thesis apply to only characteristic 2 and 3 curves (partly for this reason) and so higher characteristic cases will not be considered here. Duursma and Lee show that the Tate pairing can be implemented on groups of degenerate divisors on genus 2 curves. This means that the pairing can be computed using point arithmetic, which results in a significant reduction in complexity. They also suggest that pairing calculation can be performed efficiently if the order $p^{mp}+1$ is used. This new order has a Hamming weight of 2 in base p, which reduces the number of operations required within the iterative loop. It is also demonstrated that point addition is not required on each iteration. Instead, multiplication by the field characteristic p can be used to construct the pairing. A very efficient function for the multiplication of a point by p is provided. Frobenius actions can also be employed within the loop, which results in further savings since these operations are relatively trivial. One would expect that the larger order would increase the size of the loop from m to mp iterations. The authors show that this is not the case: a computation using m iterations is possible. The final exponentiation is also easier to calculate since $(q^k - 1)/n = (p^{2mp} - 1)/(p^{mp} + 1) = p^{mp-1}$. This is performed using Frobenius actions and a division.

The techniques described in this section result in an efficient Tate pairing computation on a restricted set of curves. These methods can be generalised and further improved, as seen in the next section.

3.3 The η_T Pairing

Barreto et al. [63] generalise the techniques of Duursma and Lee to characteristic 2 elliptic and genus 2 hyperelliptic curves and to a larger number of characteristic 3 elliptic curves. They define a generalised notation that can be used to describe pairing computation for each of the cases. They also show that further computational improvements are possible and express the optimised computations using the same notation. The techniques of Barreto et al. provide a very efficient means for pairing calculation. The methods do, however, return a different value to that returned by a more traditional computation of the Tate pairing. The pairing that is calculated using the new approach is called the *truncated eta pairing*, denoted η_T. It is bilinear and non-degenerate and can be used to form the basis for cryptographic applications in the same manner as the Tate pairing. The η_T pairing is closely related to the Tate pairing. An η_T result can be converted to a Tate result using an exponentiation.

Let C be a supersingular elliptic or hyperelliptic curve on $\mathbb{F}_q$, $q = p^m$, with an even embedding degree $k > 1$. The curve should be chosen so that it contains a distortion map ψ that enables the denominator elimination technique. Let N be the order used to compute the Tate pairing. The value of N can be the order of the Jacobian, a multiple of the order of the Jacobian, or the order of a prime subgroup of the Jacobian. Let $D_1, D_2 \in J_C(\mathbb{F}_q)$ be divisors of order dividing N and let f_{N,D_1} be a function such that $div(f_{N,D_1}) = [N]D_1$. The modified Tate pairing in this case is given by $\langle D_1, \psi(D_2)\rangle_N = f_{N,D}(\psi(D_2))$. The η_T pairing is defined as follows:

Definition 3.3.1. ***(η_T pairing)*** *For some $T \in \mathbb{Z}$ the η_T pairing is given by*

$$\eta_T(D_1, D_2) = f_{T,D_1}(\psi(D_2)) \tag{3.10}$$

The value of T can be chosen so that $T < N$, i.e. $[T]D_1$ does not have to be equivalent to zero. This means that a pairing computation can be achieved using a smaller number of iterations than would normally be required. A very small set of values of T will return a non-degenerate, bilinear pairing. However, a well defined method for the selection of T can be used. Multiplication of a finite field element by a power of p can be achieved using an automorphism on the curve. The η_T pairing is constructed so that this automorphism can be used to reduce computation time.

Barreto et al. provide a theorem that introduces several variables to define a fixed relationship between the η_T and Tate pairings.

Theorem 3.3.1. *Let $N \in \mathbb{N}$ be the order used to compute the Tate pairing. Let $M = (q^k - 1)/N$. Let each of $T \in \mathbb{Z}$, $a \in \mathbb{N}$ and $L \in \mathbb{Z}$ be co-prime to N and let $c \in \mathbb{Z}$. The η_T pairing is related to the Tate pairing by*

$$\left(\langle D_1, \psi(D_2)\rangle_N^M\right)^L = \left(\eta_T(D_1, D_2)^M\right)^{aT^{a-1}} \tag{3.11}$$

where:

1. *$[T]D_1 \equiv \gamma(D_1)$, where γ is some automorphism of C that is defined over $\mathbb{F}_q$.*
2. *γ and ψ satisfy the condition*

$$\gamma\psi^q(D_2) = \psi(D_2) \tag{3.12}$$

3. *$T^a + 1 = LN$*
4. *$T = q + cN$*

The following general equations are useful for the computation of the η_T pairing.

$$f_{T,D_1}(\psi(D_2))^{TM} = f_{T,TD_1}(\psi(D_2))^M \tag{3.13}$$

and

$$div(f_{T^a,D_1}) = div(f_{T,D}^{T^a-1}.f_{T,TD}^{T^a-2} \dots f_{T,T^{a-2}D}^{T}.f_{T,T^{a-1}D}) \tag{3.14}$$

The computation and hardware implementation of the Tate pairing using the η_T method in the characteristic 2 elliptic, the characteristic 2 genus 2 hyperelliptic and in the characteristic 3 elliptic cases are discussed in detail in Chapters 4, 5 and 6 respectively. The specific relationships between the Tate pairing and the η_T pairing in each of these cases will be described then.

3.4 Pairings in Cryptography

3.4.1 Security

The pairing-based cryptographic schemes that are relevant to this work rely on the *Bilinear Diffie-Hellman problem* (BDH).

Definition 3.4.1. ***(Bilinear Diffie-Hellman Problem (BDH))*** *Consider the groups G_1 and G_2 of prime order n. Let P be a generator of G_1. Let $e : G_1 \times G_1 \to G_2$ be a bilinear, non-degenerate map. Let $a, b, c \in \mathbb{Z}$. Given (P, aP, bP, cP), the BDH is the problem of finding the value of $e(P, P)^{abc}$.*

Systems that rely on the BDH are only as secure as the CDH (previously discussed in Section 2.6) in the groups used by the pairing. In the schemes relevant to this work G_1 is a group on an elliptic or a genus 2 hyperelliptic curve. This means that security is reliant on either the ECDLP or the HCDLP in this group. The second group, G_2, is the multiplicative finite field $\mathbb{F}^*_{q^k}$. The DLP in G_2 should be sufficiently difficult to protect against feasible attacks. All of the attacks on the DLP, the ECDLP and the HCDLP that were described in Subsection 2.6.1 can be used. This means that, for current security requirements, q^k should have an order of approximately 1024 bits and q should be large enough so that attacks on the ECDLP or the HCDLP are infeasible in practice.

3.4.2 Protocol Example: The Boneh-Franklin IBE Scheme

The concept of identity-based encryption and the advantages it provides over conventional public key schemes has been discussed in Subection 2.2.3. Having introduced the operations necessary for pairing computation, an example implementation of a Boneh-Franklin IBE scheme [26] is detailed here in order to illustrate a practical application of pairings in cryptography. Boneh and Franklin discuss two schemes called *BasicIdent* and *FullIdent*. *FullIdent* is the more complicated of the two but, unlike *BasicIdent*, protects against chosen ciphertext attacks. The *BasicIdent* scheme, when implemented using the η_T pairing, is discussed here. The Tate pairing can be used in the same manner. An elliptic curve is used to describe the scheme but a genus 2 curve could be used just as easily. In the following example, a user Alice wishes to send a message M to Bob. The scheme consists of four steps.

1. **Setup**
   - Choose an elliptic curve $E(\mathbb{F}_q)$ with embedding degree k along with a point P that generates a cyclic group $\langle P \rangle$ of order N.
   - Let ê be a bilinear, non-degenerate, reduced modified pairing that maps two elements $A, B \in \langle P \rangle$ to an element of $\mathbb{F}^*_{q^k}$ such that
   $$\hat{e}(A, B) = \eta_T(A, B)^{(q^k-1)/N} \tag{3.15}$$
   - Select a secret master key $s \in \mathbb{Z}$, where $s < N$. Compute $P_{pub} = [s]P$.
   - Choose a cryptographic hash function H_1 to map a binary string of arbitrary bit length to an element of $\langle P \rangle$. Choose another hash function H_2 that maps an element of $\mathbb{F}^*_{q^k}$ to a binary string of length t, where t is the maximum bit length of the message to be sent.
   - The system parameters are $(q, N, t, \langle P \rangle, \mathbb{F}^*_{q^k}, \hat{e}, P, P_{pub}, H_1, H_2)$ and are available to all users.
2. **Extract**

- Let ID be Bob's identifier. The PKG computes $Q_{ID} = H_1(ID)$, where $Q_{ID} \in \langle P \rangle$.
- The PKG computes $B_{priv} = [s]Q_{ID}$ using its master key. This is Bob's private key and is transferred to him over a secure channel.

3. **Encrypt**

   - Alice calculates $Q_{ID} = H_1(ID)$.
   - She then computes $g_1 = \hat{e}(Q_{ID}, P_{pub})$.
   - A random positive integer $r < N$ is generated.
   - The message M is incorporated into the ciphertext, C. The ciphertext is computed according to

$$C = (U, V) = ([r]P, M \oplus H_2(g_1^r))$$

4. **Decrypt**

   - Bob computes $g_2 = \hat{e}(U, B_{priv})$ on receipt of the ciphertext $C = (U, V)$.
   - The message M can now be retrieved according to

$$M = V \oplus H_2(g_2)$$

The bilinearity of the pairing enables Bob to retrieve the message. Bob computes $V \oplus H_2(g_2) = M \oplus H_2(g_1^r) \oplus H_2(g_2)$. This means that M is returned if g_1^r and g_2 are equal. This is easily demonstrated:

$$\begin{aligned}
g_1^r &= \hat{e}(Q_{ID}, P_{pub})^r \\
&= \hat{e}(Q_{ID}, [s]P)^r \\
&= \hat{e}([s]Q_{ID}, [r]P) \\
&= \hat{e}(B_{priv}, U) \\
&= g_2
\end{aligned}$$

In order to compromise the system an eavesdropper would have to recover the value of s from $P_{pub} = [s]P$ or the value of g_1 from g_1^r. The former requires a solution to the ECDLP while the latter requires a solution to the DLP in $\mathbb{F}_{q^k}^*$.

3.5 Methodology and Design System

A justification for the methodology of this work is provided in this section. The equipment used during the research is also described. The design cycle is discussed. A software system for the efficient generation and analysis of the pairing processors is also presented. This system reduces the design time required during the architectural definition, implementation and verification stages.

3.5.1 Justification

The use of dedicated hardware architectures for pairing computation provides several benefits. Many of the extension field arithmetic operations can be implemented using subfield units that operate in parallel. Pairing algorithms can also be expressed in terms of distinct computational stages. Custom hardware units can be designed for the fast implementation of these stages. The algorithms can be sequenced so that these units have the ability to operate in parallel.

The goal of this work is to create custom hardware processors that return a Tate pairing result as quickly as possible while also ensuring that resources are used efficiently. Processors are designed in the characteristic 2 and 3 elliptic cases and in the characteristic 2 genus 2 hyperelliptic curve case. The efficient generation of these hardware processors is vital. It is also important that the implementation platform be inexpensive so that various architectural solutions can be explored at will. Timing results must also be available as quickly as possible after each design stage so that modifications can be made, if necessary, and the updated architectures quickly re-instantiated. A robust verification procedure is also necessary to ensure the reliability of the processors.

3.5.2 Equipment

The hardware architectures of this work are implemented on a *Field Programmable Gate Array* (FPGA). These devices are relatively inexpensive. Their major advantage is the ease with which they can be reconfigured. The FPGA used for this work is a Xilinx Virtex-II Pro FPGA (xc2vp100-6ff1696) [72]. The high level architectural details of this family of FPGAs is described in [72]. The Virtex-II Pro consists of an array of *Configurable Logic Blocks* (CLBs), dedicated *Block RAM* (BRAM) and programmable interconnect for routing. Each CLB contains four modules, known as slices. Each slice contains two 4-input function generators, tri-state buffers, carry and arithmetic logic gates and two multiplexers. The function generators can be configured as two 4-input Lookup Tables (LUTs), RAM or registers. A switch matrix is used to connect the CLBs with the routing system of the FPGA. The area metric for a hardware architecture on an FPGA is the number of slices it uses. The Virtex-II has a capacity of 44,096 slices. The product of the area required by an implementation and the cycles required for computation is used as an indication of its efficiency. This is measured in *cycles.s*. A desired hardware architecture is described using VHDL, defined at the RTL level. The *Xilinx ISE* tool (version 8.1) is then used on a PC to synthesise the VHDL. ISE also determines how the design should be placed and routed, and creates a *bit file* that can be used to program the FPGA.

An interface is required between the host PC and the FPGA. The FPGA is mounted on an Alpha Data PCI Mezzanine Card (ADM-XRC-2 PMC) [73]. This card handles input/output for the FPGA and provides a user programmable clock and high local bus speeds. The Alpha Data card is connected to a Celoxica RC2000 PMI-PCI carrier card [74]. The PC is connected to the RC2000 with a standard PCI connector, which, in turn, communicates with the Alpha Data card containing the FPGA. A *C* code Application Programming Interface (API) is provided by the vendors and handles communication between the PC and the FPGA.

A typical design and implementation flow for Xilinx FPGAs using ISE can be summarised as follows. This flow is discussed in more detail in [75].

1. **Design Specification:** Architectures that describe the operations to be performed are designed. These are composed of interconnected units, which should each perform a specific, well-defined function.
2. **VHDL Definition:** Each unit is defined by writing VHDL at the RTL level. All storage, logic and control systems must be described. Test benches are also defined.
3. **Synthesis:** The Xilinx ISE tool checks syntax and also analyses the hierarchy of the design. If the hierarchy is not optimum, new interconnections are created that provide the same functionality. A netlist, formally describing the usage of specific components within the CLBs and slices is created. A file can also be added by the designer at this point that provides timing, implementation and hierarchical constraints if desired.
4. **Implementation:** Implementation involves the following steps: translation (merges the synthesis netlist and constraints into one design file), map (fits design into available resources on the target device), place and route (places and routes design on the device, selecting the most desirable interconnect structure) and program file generation (the creation of a bitstream file).
5. **Bitstream Download and FPGA Configuration:** The generated bitstream is downloaded onto the FPGA and it is configured according to the description provided.

Verification can be performed at various steps during the process. Behavioural verification can be performed after Step 2. The Modelsim XE analysis tool [76] can be used to rapidly test for functionality. This step provides an indication of any errors in the RTL code.

It does not, however, take into account any errors that may be introduced by timing delays. Post-synthesis and post-mapping simulation can also be useful, although for the architectures used in this research, they can be relatively time consuming due to the large areas required. A possible solution is to perform initial verification iterations using small field sizes.

Post synthesis, mapping and on-FPGA failure may be caused by several issues. These include incorrect binding, which may be caused by errors introduced by the attempted optimisations of the synthesis step. This can be solved by modifying the original hierarchy and interconnectivity of the design, changing the scheduling of operations, or changing the constraints. Fan-out, caused by loading the output of logical gates with too many inputs to other digital logic, can also be problematic and may require architectural and constraint design changes. The ISE tool itself may introduce failure while endeavouring to map the architecture to the smallest possible number of slices. This type of failure is common in a target device that that is approaching full occupancy. The Xilinx tools struggle to efficiently implement the desired architecture and the propagation of signals through the device may cause issues. A possible solution is to loosen area constraints, to add constraints that ensure the problem units are mapped so that they are placed near to each other on the target device, or to reduce the frequency used to clock the device.

3.5.3 Automated Design and Verification Environment

This work necessitates the creation of processors computing three different Tate pairings. Many iterations of the design cycle are required during the exploration and implementation stages. The results of each of the iterations must be verified and the processors benchmarked. Defining hardware architectures at the RTL level can be very time consuming. There is a significant probability of error as the work is very involved. The implementation stages usually require a lot of user intervention as various tools must be used. Verification may require significant effort due to the numerous issues that may cause failure. A software design program for the pairing processors of this work has been created for these reasons.

The program is denoted *design_sys* for convenience. The software is written in *C++* and includes Shell scripts that provides a significant level of automation. The object-oriented nature of this programming language is used extensively so that a designer can use various features of the system independently or in combination with each other. The final outcome is a highly automated design and verification environment, which significantly reduces the time and effort required to prototype a large range of architectures.

The design system contains a class for the software computation of the Tate pairing in each of the three cases. This is called *soft_sub_sys* and contains base classes that define the required curve, extension fields and the subfield operations. Pairing algorithms are written in terms of operations that call functions within these classes. *The Multiprecision Integer and Rational Arithmetic C Library* (MIRACL) is used to perform some of the curve and field arithmetic [77].

A class, called *vhdl_sub_sys*, has been created to aid in the generation of low level RTL descriptions of the hardware architectures. Arrays, conditional statements and iterative loops are used to reduce the repetitive nature of RTL level design. Variables are used to define field sizes, irreducible polynomials and other properties. This means that VHDL can easily be regenerated if changes in field and curve definition are required. As will be seen in later chapters, the architectural parameters of the arithmetic units can also be manipulated using variables.

The system contains an *analysis_sub_sys* in which architectural efficiency can be explored. The number of hardware clock cycles required for the implementation of operations through the desired hardware units can be computed in software. Control schemes and operation sequencing can also be investigated. The redundancy of hardware units can also be explored and designs altered if they are deemed to be inefficient.

The *imp_sub_sys* class is used to oversee the vendor tools, communicate with the FPGA and to handle verification and benchmarking. The Xilinx synthesis and Modelsim testing programs are invoked from here. Constraints are provided, mapping tools are called, and a bit file is retrieved on completion. The communication API library is incorporated. This class requests test vectors from *soft_sub_sys* and sends them to the FPGA. Results are retrieved from the processor and compared to those returned by the software computation. More than one processor can be implemented and tested without user intervention by using arrays that describe the desired hardware architectural parameters. The corresponding set of processors are then implemented, verified and benchmarked by the system and metrics automatically stored for later analysis. This capability is particularly useful for prototyping.

Finally, the system contains a flexible subclass, called *flex_sub_sys*, that provides some extra functionality for the design and implementation of the flexible Tate pairing processors. These processors are discussed in Chapter 6 and the *flex_sub_sys* class is described in Subsection 6.4.4.

An example of an automated flow, used to design and verify a Tate pairing processor, is illustrated in Figure 3.1. It is assumed that the software that performs the pairing algorithm to be implemented has already been written and included in the *soft_sub_sys.* The hierarchy of the system, its connectivity at a high level of abstraction, and the scheduling of operations are defined by a designer using the *vhdl_sub_sys*. A file describing the implementation constraints to be used by the Xilinx tools can also be created. Variables defining mathematical parameters such as the curve, field size and irreducible polynomial are defined. Architectural parameters of the finite field arithmetic modules (such as multiplier digit sizes) are also set here. Test benches can also be written with ease. Variables defining whether verification is to be performed at various stages of the implementation process are also set by the user.

The *analysis_sub_sys* can be used at this point to explore various architectural and scheduling options. The estimated times required to perform arithmetic operations in hardware can be investigated. The redundancy of hardware units can be computed to ensure that area resources are being used efficiently.

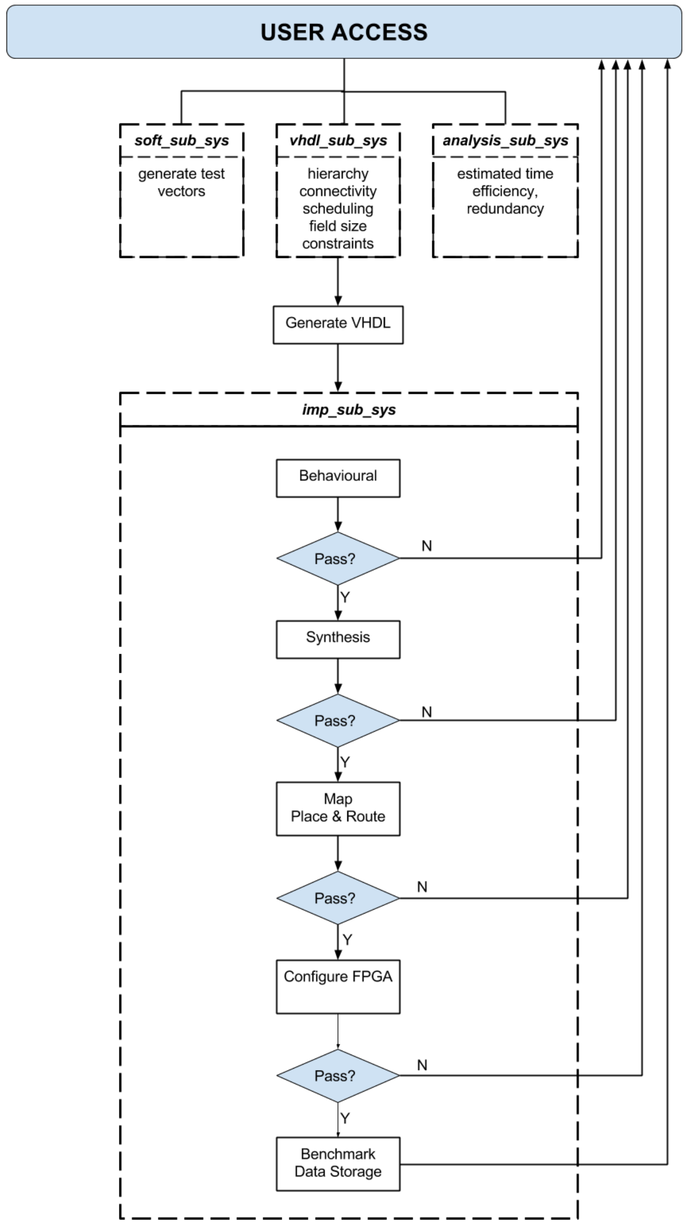

Figure 3.1: Automation of the design cycle using *design_sys*

Once all parameters have been chosen and the architectural connectivity and the scheduling of operations have been defined, VHDL describing the desired hardware processor is automatically generated by the *vhdl_sub_sys*. At this point, the flow enters the implementation subsystem *imp_sub_sys*. Extensive use is made of Shell scripting in order to automate tool usage, verification, benchmarking and data storage.

Behavioural verification is first performed. If failure occurs, control is handed back to the user of the system so that its cause can be investigated. Otherwise, synthesis commences. Post-synthesis verification is optional, as it can be time-consuming. User access is again provided if failure occurs. Once the post-synthesis netlist has been generated, the mapping and place & route steps are performed. On completion, a bitstream is returned by the Xilinx ISE tool. This file is transferred through the PCI card and programs the FPGA.

On-FPGA verification can be time consuming as, if failure occurs, the full implementation process must be repeated. To mitigate this, a system user can access the *imp_sub_sys* to change the I/O ports of the FPGA with ease so that intermediate values can be read. The *soft_sub_sys* can be used to provide its corresponding values so that comparisons can quickly be made and debugging performed efficiently. Verification of newly created processors is performed by computing 10,000 pairings on the processor using randomly generated input points. The results returned are compared to the results of software computations on the same points. Protection against the computation of a result at infinity does not need to be provided as this is not possible in these pairing-based systems. It has been noted since the research was performed that, due to the complexity of the processors created during this work, the successful computation of one pairing is usually sufficient proof of correct operation as full coverage is provided by the propagation of bits through the hardware.

Once a design and verification cycle has been completed successfully, the environment can be used to perform automated implementation and verification of various versions of the hardware pairing processors. Variables describing the mathematical and architectural parameters of the processors are stored in arrays within the *imp_sub_sys*. The design flow iterates until all desired hardware processors have been created, verified and benchmarked. The data outputs of each iteration can also be stored, if desired.

This environment reduces the difficulties associated with the creation of hardware architectures that must implement complex operations. It enables a high level of control over processor generation and verification while significantly reducing the effort required. A designer is able to spend the majority of available time on hardware specification and analysis without distraction. The rest of the process is aided by the software system: it generates the VHDL files, enables rapid analysis, calls the synthesis and programming tools, initiates hardware computation and handles verification and benchmarking. Similar design environments were later described in [78], [79] and [80].

This system is an important contribution of this work. Hundreds of thousands of lines of C++ and Shell code were written to design it. It was carefully developed to be easy to use, robust, scalable and to significantly reduce the effort required for FPGA design and verification. The design and verification environment enabled the author of this thesis to rapidly generate hardware processors for pairing-based cryptography. The environment, the hardware processors it produced and the results returned were described by the author of this thesis in [11], [12] and [13].

3.6 Conclusions

The Tate pairing and its efficient computation have been introduced in this chapter. The η_T pairing has also been discussed in detail. This pairing is closely related to the Tate pairing but can be computed more quickly in many cases. The η_T pairing is bilinear and non-degenerate and can be used in the same manner as the Tate in pairing-based applications. A conversion between pairing results is enabled by an exponentiation. The computation of the η_T pairing on characteristic 2 and 3 supersingular elliptic curves and on characteristic 2 genus 2 supersingular curves is particularly efficient. These curves have embedding degrees of 4, 6 and 12, respectively.

Security is reliant on both the DLP in the abelian additive group on the curve and on the DLP in the multiplicative extension field $\mathbb{F}_{q^k}^*$. This means that curves with larger embedding degree can provide the necessary level of security using a finite field $\mathbb{F}_q$ of smaller order, when compared to curves with smaller embedding degree. Note that the order of $\mathbb{F}_q$ must still be large enough so that attacks on the additive group on the curve (i.e. on the ECDLP or HCDLP) are infeasible. The hyperelliptic case offers the highest embedding degree but the benefit of the smaller underlying finite field size is offset by a more complex pairing computation. Similarly, the characteristic 3 elliptic curve case appears to be more attractive than the characteristic 2 case. Implementation of arithmetic in characteristic 3 is, however, more complicated than in characteristic 2 in both software and hardware.

This work investigates the hardware implementation of the Tate pairing in all three cases. The computation of the η_T pairing forms the basis for these implementations. The creation of dedicated architectures for pairing computation offers many benefits. The complexity of the required finite and extension field arithmetic and the difficulty of efficient operation scheduling does, however, present significant design challenges. At the time of this work, it was not clear if the η_T methods offered a worthwhile advantage over other algorithms in terms of the hardware computation of the Tate pairing.

A robust design environment, created to reduce the effort required to create and verify pairing processors and to automate the design cycle, has been discussed in this chapter. The advantages of its use will be apparent throughout the thesis.

A dedicated hardware architecture for characteristic 2 elliptic curve Tate pairing computation is presented in Chapter 4. The design of a custom architecture for characteristic 2 genus 2 pairing computation is described in detail in Chapter 5. A flexible processor, which can implement the Tate pairing on both characteristic 2 and 3 elliptic curves, is presented in Chapter 6.

Chapter 4

A Hardware Processor for Tate Pairing Computation on a Characteristic 2 Elliptic Curve

4.1 Introduction

The efficient computation of the characteristic 2 Tate pairing using the η_T method in hardware is discussed in this chapter. Hardware systems enable parallel computation. In this chapter it is shown that, through slight modification of the pairing algorithms, many of the required operations can be performed at the same time. Some of the required extension field arithmetic can also be implemented in terms of parallel $\mathbb{F}_{2^m}$ operations. Much of the arithmetic on $\mathbb{F}_{2^m}$ can, in turn, be performed in terms of operations on $\mathbb{F}_2$, all of which can be performed at the same time. A characteristic 2 elliptic curve pairing processor that takes advantage of this parallelism and other properties of the Tate pairing is discussed in this chapter.

The equations and algorithms required for Tate pairing computation using the η_T methods are provided in Section 4.2. The hardware implementation of arithmetic on $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^{4m}}$ is discussed in Sections 4.3 and 4.4, respectively. Dedicated hardware architectures, designed for the parallel computation of various pairing computation stages, are described in Section 4.5. Finally, the elliptic curve characteristic 2 Tate pairing processor is presented in Section 4.6 and the properties of several implementations are discussed.

4.2 The Characteristic 2 Elliptic Curve η_T and Tate Pairings

The characteristic 2 elliptic curve η_T pairing and its exponentiation to the Tate pairing is discussed in this section. The Duursma and Lee techniques [62] are first expressed in terms of the η_T notation presented by Barreto et al. [63]. Computation of the Tate pairing using an η_T pairing of reduced order, as discussed by Barreto et al., is then discussed in detail. More detail on much of the subject matter of this section is provided by Barreto et al. in [63].

Let $E(\mathbb{F}_q) : y^2 + y = x^3 + x + b$ be an elliptic curve defined on $\mathbb{F}_q = \mathbb{F}_{2^m}$, where $b = 0$ or 1 and m is odd. This curve has embedding degree $k = 4$. The order of the curve is given by $\#E(\mathbb{F}_{2^m}) = 2^m + 1 + (-1)^b 2^{(m+1)/2}$ if $m \equiv (1, 7) \mod 8$ or $\#E(\mathbb{F}_{2^m}) = 2^m + 1 - (-1)^b 2^{(m+1)/2}$ if $m \equiv (3, 5) \mod 8$.

The basis $\{1, \delta, \epsilon, \delta\epsilon\}$ is used to represent members of $\mathbb{F}_{2^{4m}}$, where $\delta, \epsilon \in \mathbb{F}_{2^{4m}}$ such that $\delta^2 = \delta + 1$ and $\epsilon^2 = \epsilon + \delta$. A distortion map providing denominator elimination is given by $\psi(x, y) = (x + \delta^2, y + \delta x + \epsilon)$.

Given a point $P = (x, y)$ and a function $\phi(x, y) = (x + 1, y + x)$ then it can be shown that

$$[2^i]P = \phi^i({x_p}^{(2i)}, {y_p}^{(2i)}) \tag{4.1}$$

This means that the expression $[q]P = \phi^m(P)$ can be used.

Barreto et al. use Theorem 3.3.1 to relate the Duursma and Lee techniques to computation of the Tate pairing using the η_T method. In this case, $T = q = 2^m$ and $N = 2^{2m} + 1$. This means that $c = 0$. The equation $T^a + 1 = LN$ must be satisfied, and this is achieved by setting $a = 2$ and $L = 1$. The value of M is given by $M = (q^k - 1)/N = (2^{4m} - 1)/(2^{2m} + 1) = 2^{2m} - 1$. The relationship between the η_T pairing and the Tate pairing is given by

$$\left(\eta_T(P, Q)^M\right)^{2q} = \langle P, \psi(Q)\rangle_N^M \tag{4.2}$$

Barreto et al. show that the value of T can again be reduced. Using $N = \#E(\mathbb{F}_{2^m}) = 2^m + 1 \pm 2^{(m+1)/2}$ then $[N]P = \infty$ for a point P on $E(\mathbb{F}_{2^m})$. The automorphism $\gamma(P) = [q]P$ can be rewritten as $\gamma(P) = [q - N]P = [\mp 2^{(m+1)/2} - 1]P$. This means that $T = \mp 2^{(m+1)/2} - 1$ can be used. From Theorem 3.3.1 the expression $c = -1$ must hold. Setting $a = 2$ yields $L = 2$. The exponent is given by $M = (2^{4m} - 1)/(2^m + 1 \pm 2^{(m+1)/2}) = (2^m \mp 2^{(m+1)/2} + 1)(2^{2m} - 1)$. The relationship between the pairings in the characteristic 2 elliptic case is now given by

$$\left(\eta_T(P, Q)^M\right)^{T} = \langle P, \psi(Q)\rangle_N^M \tag{4.3}$$

4.2.1 Computation of the η_T Pairing

The η_T pairing is returned according to $\eta_T(P, Q) = f_{T,P}(\psi(Q))$, where $f_{T,P}$ is the Miller function. This function is evaluated and accumulated using the fast point doubling operations available on the curve. From the value of T it is clear that computation requires $(m + 1)/2$ point doublings and a point addition. A point is doubled according to straight line and vertical line functions. As discussed previously, the vertical line function is not required for pairing calculation due to denominator elimination. Consider a point $A = (x_A, y_A)$. The straight line function governing the calculation of $[2]A$ is

$$g_A(x, y) = (x_A^2 + 1)(x_A + x) + (y_A + y) \tag{4.4}$$

Equations (3.13) and (3.14) can be used to show that the Miller function can be evaluated at $\psi(Q)$ according to

$$f_{T,P}(\psi(Q)) = \left(\prod_{i=0}^{(m-1)/2} \Big(g_{[2^i]P}(\psi(Q)) \Big)^{2^{(m-1)/2-i}} \right) .l(\psi(Q) \tag{4.5}$$

where $l(\psi(Q))$ is the line function arising from the final addition. Exponentiating to $2^{(m-1)/2-i}$ on each iteration requires $(m-1)/2$ $\mathbb{F}_{2^{4m}}$ squarings. These operations are, however, not necessary if the Miller function is accumulated differently. Let $P' = [2^{(m-1)/2}]P$. The value of P' can be calculated trivially according to $P' = \phi^{(m-1)/2}(\sqrt{x_P}, \sqrt{y_P})$, where $\phi(x, y) = (x + 1, y + x)$. The Miller function is now constructed using a point halving operation, which has the same complexity as doubling. Let $j = 2^{(m-1)/2} - i$. The evaluation of the Miller function at $\psi(Q)$ is now given by

$$f_{T,P}(\psi(Q)) = l(\psi(Q)). \left(\prod_{j=0}^{(m-1)/2} g_{[2^{-j}]P'}(\psi(Q))^{2^j} \right) \tag{4.6}$$

The definitions of the functions of Equation (4.6) are dependent on the value of b describing the elliptic curve and on the value of $m \mod 8$. The results presented at the end of this chapter are returned by a hardware implementation on a field with $m = 313$, which means that $m \mod 8 \equiv 1$. The irreducible trinomial defining $\mathbb{F}_{2^{313}}$ is $x^{313} + x^{79} + 1$. The middle coefficient of the trinomial is the smallest of those that can be used to generate a field of this size. This provides the best performance as the choice minimises the number of $\mathbb{F}_2$ operations required to perform $\mathbb{F}_{2^m}$ arithmetic. In this work, members of $\mathbb{F}_{2^{4m}}$ are represented by degree 3 polynomials generated by the irreducible polynomial $x^4 + x + 1$. These fields and irreducible polynomials are the same as those used by Barreto et al. in [63]. It should be noted, however, that the design environment supports the implementation of processors on any finite field generated by an irreducible polynomial.

The functions required for pairing computation can now be defined. Let $t = x_P$. The line function describing the final addition is given by

$$\begin{aligned} l(\psi(Q)) = & \Big(t.\big(x_P + x_Q + 1\big) + y_P + y_Q + b\Big) + \\ & \Big(t + x_Q + 1\Big)x + \Big(t + x_Q\Big)x^2 + \Big(0\Big)x^3 \end{aligned} \tag{4.7}$$

Now let $t = x_P^{(-j)} + 1$ (the brackets around the exponent are used for clarity). The intermediate functions are computed according to

$$\begin{aligned} g_{[2^{-j}]P'}(\psi(Q))^{2^j} =& \Big(t.\big(x_P^{(-1-j)} + x_Q^{(j)} + 1\big) + y_P^{(-1-j)} + y_Q^{(j)}\Big) + \\ & \Big(t + x_Q^{(j)} + 1\Big)x + \Big(t + x_Q^{(j)}\Big)x^2 + \Big(0\Big)x^3 \end{aligned} \tag{4.8}$$

The operations required for η_T pairing computation using Equations (4.6), (4.7) and (4.8) are presented in Algorithm 4. The most computationally expensive operations are required on Steps 2, 6, 8 and 10. Steps 2 and 6 require an $\mathbb{F}_{2^m}$ multiplication each, while Steps 8 and 10 require expensive $\mathbb{F}_{2^{4m}}$ multiplications

Algorithm 4 Computation of $\eta_T(P,Q)$ on $E(\mathbb{F}_{2^m}) : y^2 + y = x^3 + x + b$, $m \mod 8 \equiv 1$

INPUT: $P = (x_P, y_P), Q = (x_Q, y_Q)$, where $P, Q \in E(\mathbb{F}_{2^m})$

OUTPUT: $f = \eta_T(P,Q)$, where $f \in \mathbb{F}_{2^{4m}}$

1: Initialise: $t \leftarrow x_P$, $mill \leftarrow 1$
2: $f \leftarrow t.(x_P + x_Q + 1) + y_P + y_Q + b + (t + x_Q + 1)x + (t + x_Q)x^2$
3: $mill \leftarrow 1$
4: **for** $(i \leftarrow 0,\ i < (m+1)/2,\ i \leftarrow i + 1)$ **do**
5: $\quad t \leftarrow x_P + 1, x_P \leftarrow \sqrt{x_p}, y_P \leftarrow \sqrt{y_P}$
6: $\quad u \leftarrow t.(x_P + x_Q + 1) + y_P + y_Q + (t + x_Q + 1)x + (t + x_Q)x^2$
7: $\quad x_Q \leftarrow x_Q^2,\ y_Q \leftarrow y_Q^2$
8: $\quad mill \leftarrow mill.u$
9: **end for**
10: $f \leftarrow mill.f$

RETURN: f

Some of the computations required within the *for* loop can be performed in parallel if pairs of iterations are combined and some additional polynomials are introduced. This takes advantage of the sparse nature of the polynomials and is known as *unrolling* the loop. The operations required to calculate the η_T pairing using this technique are detailed in Algorithm 5. Note that i is incremented by 2 on each iteration of the new loop. Steps 2, 5, 8 and 14 require an $\mathbb{F}_{2^m}$ multiplication each. The polynomials u_0 and u_1 can be calculated in parallel in hardware. A custom multiplication routine, *smul*, is used to multiply u_0 by u_1

on Step 10. This routine is constructed with a hardware implementation in mind. Many of the $\mathbb{F}_{2^m}$ operations that are required during loop iteration can also be performed in parallel if they are scheduled carefully. Steps 11, 15 and 16 require an $\mathbb{F}_{2^{4m}}$ multiplication each. The hardware architectures that have been created to efficiently implement the operations required by Algorithm 5 are described later in this chapter.

Algorithm 5 Unrolled computation of $\eta_T(P,Q)$ on $E(\mathbb{F}_{2^m}) : y^2 + y = x^3 + x + b$, $m \bmod 8 \equiv 1$

INPUT: $P = (x_P, y_P), Q = (x_Q, y_Q)$, where $P, Q \in E(\mathbb{F}_{2^m})$

OUTPUT: $f = \eta_T(P,Q)$, where $f \in \mathbb{F}_{2^{4m}}$

1: Initialise: $t \leftarrow x_P$, $mill \leftarrow 1$

2: $f \leftarrow t.(x_P + x_Q + 1) + y_P + y_Q + b + (t + x_Q + 1)x + (t + x_Q)x^2$

3: **for** $(i \leftarrow 0,\ i < (m-1)/2,\ i \leftarrow i+2)$ **do**

4: $\quad t \leftarrow x_P + 1$, $x_P \leftarrow \sqrt{x_P}$, $y_P \leftarrow \sqrt{y_P}$

5: $\quad u_0 \leftarrow t.(x_P + x_Q + 1) + y_P + y_Q + (t + x_Q + 1)x$

6: $\quad x_Q \leftarrow {x_Q}^2$, $y_Q \leftarrow {y_Q}^2$

7: $\quad t \leftarrow x_P + 1$, $x_P \leftarrow \sqrt{x_P}$, $y_P \leftarrow \sqrt{y_P}$

8: $\quad u_1 \leftarrow t.(x_P + x_Q + 1) + y_P + y_Q + (t + x_Q + 1)x$

9: $\quad x_Q \leftarrow {x_Q}^2$, $y_Q \leftarrow {y_Q}^2$

10: $\quad u \leftarrow smul(u0, u1)$

11: $\quad mill \leftarrow mill.u$

12: **end for**

13: $t \leftarrow x_P + 1$, $x_P \leftarrow \sqrt{x_P}$, $y_P \leftarrow \sqrt{y_P}$

14: $u \leftarrow t.(x_P + x_Q + 1) + y_P + y_Q + (t + x_Q + 1)x + (t + x_Q)x^2$

15: $mill \leftarrow mill.u$

16: $f \leftarrow mill.f$

RETURN: f

4.2.2 Exponentiation to the Tate Pairing

The extension field element $f = \eta_T(P,Q)$ returned by Algorithm 5 must be exponentiated to $M = (2^m - 2^{(m+1)/2} + 1)(2^{2m} - 1)$ to provide a unique and useful pairing value. Computation of the η_T pairing followed by this exponentiation can be used to form the basis for pairing-based cryptographic applications. In this work, however, $\eta_T(P,Q)^M$ is further exponentiated to $T = 2^{(m+1)/2} \pm 1$ so that a Tate pairing value is returned. This means that the results returned by the processor can be directly compared to other Tate pairing implementations in the literature. The value of T is $2^{(m+1)/2}+1$ in the $m \mod 8 \equiv 1$ case. The operations required for the exponentiation of $\eta_T(P,Q)$ to $\eta_T(P,Q)^{MT} = \langle P, \psi(Q)\rangle_N^M$ are provided in Algorithm 6. The most expensive operations are the $\mathbb{F}_{2^{4m}}$ multiplications of Lines 6, 8, 10, 11 and 13 and the $\mathbb{F}_{2^{4m}}$ inversion of Line 12. Squaring and exponentiation to q are also required, but these operations are trivial in comparison (recall $q = 2^m$).

Algorithm 6 Exponentiation of $\eta_T(P,Q)$ to $\eta_T(P,Q)^{MT} = \langle P, \psi(Q)\rangle_N^M$, $m \mod 8 \equiv 1$

INPUT: $f = \eta_T(P,Q)$, where $f \in \mathbb{F}_{2^{4m}}$

OUTPUT: $c = \eta_T(P,Q)^{MT} = \langle P, \psi(Q)\rangle_N^M$, where $c \in \mathbb{F}_{2^{4m}}$

1: Initialise: $u \leftarrow f$, $v \leftarrow f$, $w \leftarrow f$
2: **for** $(i \leftarrow 0,\ i < (m+1)/2,\ i \leftarrow i+1)$ **do**
3:     $u \leftarrow u.u$
4: **end for**
5: $u \leftarrow u^q, v \leftarrow v^q$
6: $w \leftarrow u.v$
7: $v \leftarrow v^q$
8: $w \leftarrow w.v$
9: $u \leftarrow u^{2q}, v \leftarrow v^q$
10: $c \leftarrow u.v$
11: $c \leftarrow c.f$
12: $w \leftarrow w^{-1}$
13: $c \leftarrow c.w$

RETURN: c

The computation of $\eta_T(P,Q)$ and the exponentiation to MT require many operations on $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^{4m}}$. Addition, multiplication, squaring and inversion are required on both fields. Exponentiation of $\mathbb{F}_{2^{4m}}$ elements to q is also necessary. The hardware modules used to implement $\mathbb{F}_{2^m}$ field operations are described in the next section. The architectures that perform operations on $\mathbb{F}_{2^{4m}}$ are then described in Section 4.4.

4.3 Hardware Implementation of Arithmetic on $\mathbb{F}_{2^m}$

The finite field $\mathbb{F}_{2^m}$ is an m-degree extension of $\mathbb{F}_2$. The field $\mathbb{F}_2$ has elements 0 and 1 and all arithmetic is performed modulo 2. The additive and multiplicative operations of $\mathbb{F}_2$ are shown in Table 4.1. An element of $\mathbb{F}_2$ requires one hardware bit for storage. Addition is performed using a logical XOR gate. Multiplication is performed using an AND gate. Note that addition and subtraction yield the same result on $\mathbb{F}_2$.

Table 4.1: Addition and Multiplication on $\mathbb{F}_2$

a	**b**	**a+b**	**a.b**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Members of $\mathbb{F}_{2^m}$ are represented by polynomials with a maximum degree of $(m-1)$ with coefficients in $\mathbb{F}_2$. An element $a \in \mathbb{F}_{2^m}$ is given by

$$a(x) = \sum_{i=0}^{m-1} a_i x^i = a_0 + a_1 x + \ \ldots\ a_{m-1}x^{m-1} \tag{4.9}$$

where all $a_i \in \mathbb{F}_2$. Each member of $\mathbb{F}_{2^m}$ requires m hardware bits for storage. For simplicity, a polynomial representation should be assumed from this point and $a(x)$ will be written as a. As previously discussed in Subsection 2.3.2, the field $\mathbb{F}_{2^m}$ is generated by an irreducible polynomial, f, of degree m. Irreducible polynomials are of form $f = \sum_{i=0}^{m} f_i x^i$, where $f_m = f_0 = 1$ and all $f_i \in \mathbb{F}_2$. Low weight irreducible polynomials reduce the complexity of the field arithmetic. Trinomials are used in this work. The efficient computation of $\mathbb{F}_{2^m}$ addition, multiplication, squaring and inversion is vital for pairing-based systems. This section discusses the hardware architectures used to implement these arithmetic operations.

4.3.1 $\mathbb{F}_{2^m}$ Addition

Given $a, b, c \in \mathbb{F}_{2^m}$, addition of a and b is performed bitwise according to

$$c = a + b = \sum_{i=0}^{m-1} (a_i + b_i) x^i \tag{4.10}$$

All of the $\mathbb{F}_2$ coefficient additions can be performed in parallel. A total of m XOR gates are used to perform addition on $\mathbb{F}_{2^m}$ and a result is returned in one clock cycle.

4.3.2 $\mathbb{F}_{2^m}$ Multiplication

An introduction to multiplication on $\mathbb{F}_{p^m}$ was provided in Subsection 2.3.2. A multiplication $c = a.b$, where $a, b, c \in \mathbb{F}_{2^m}$, is performed in two stages: composition and reduction. Given $a = \sum_{i=0}^{m-1} a_i x^i$ and $b = \sum_{j=0}^{m-1} b_j x^j$ a polynomial z is first computed according to

$$z = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i.b_j x^{i+j} \tag{4.11}$$

where z is of degree $2m - 2$. This new polynomial is then reduced modulo the irreducible polynomial f in the second step.

$$c = z \mod f \tag{4.12}$$

There are several ways to perform $\mathbb{F}_{2^m}$ multiplication in hardware. Reduction is usually performed in hardware using the relationship $f = \sum_{i=0}^{m} f_i x^i = x^m + \sum_{i=0}^{m-1} f_i x^i$. An example of reduction using this method was provided in Subsection 2.3.2. This technique is implemented using a network of XOR and AND gates. In this work, the VHDL that describes the reduction networks is automatically generated in software according to desired field size and irreducible polynomial.

One general technique forms the basis for reduction and is described first. Consider the field $\mathbb{F}_{2^m}$ generated by the irreducible polynomial $f = \sum_{i=0}^{m} f_i x^i = x^m + \sum_{i=0}^{m-1} f_i x^i$. Since x is a root of the irreducible polynomial then $f = 0 \in \mathbb{F}_{2^m}$. This means that

$$x^m = \sum_{i=0}^{m-1} f_i x^i \tag{4.13}$$

This relationship can be used to recursively reduce a composition polynomial by replacing the terms of higher degree with terms of degree smaller than m. As an example, consider the field $\mathbb{F}_{2^4}$, generated by an irreducible polynomial $f = x^4 + x + 1$. The relationship $x^4 = x + 1$ holds on this field. Let the polynomial $a = x^3 + x^2 + 1$ represent the element $(1101)_2$ and let $b = x^3 + 1$ represent $(1001)_2$. The composition stage produces the polynomial $z = x^6 + x^5 + x^2 + 1$. This can be reduced by noting that $x^5 = x.x^4 = x(x+1) = x^2 + x$. Also, $x^6 = x.x^5 = x(x^2+x) = x^3 + x^2$. Now, $z = x^3 + x^2 + x^2 + x + x^2 + 1 = x^3 + x^2 + x + 1$.

Various strategies have been suggested for the efficient hardware implementation of field multiplication. In *bit-serial* architectures [81, 82] multiplication is performed according to

$$c = a.b = (\sum_{i=0}^{m-1} a_i x^i).(\sum_{j=0}^{m-1} b_j x^j) = \sum_{j=0}^{m-1} ((b_j.ax^j) \mod f) \tag{4.14}$$

This method requires an iterative implementation. In hardware, one bit of the b polynomial is multiplied by a left-shifted version of the a polynomial on each iteration. The resultant polynomial is reduced modulo f and accumulated with the results of previous iterations. The main advantage of an implementation of this type is a relatively small area footprint. A result is returned after m clock cycles. This cycle cost is, however, too high for the applications relevant to this research as the field sizes are large.

In *bit-parallel* architectures [83, 84, 85], parallel multiplications of the a polynomial by each bit of b are performed. These architectures return a result in one clock cycle. The area required, however, is prohibitively large for the relevant field sizes.

Digit-serial architectures were proposed by Song and Parhi in [86]. These multipliers offer a balance between resource usage and the number of clock cycles required to return a result. Computations are performed in parallel on D bits of one of the input polynomials. D is known as the *digit size* and $1 \leq D \leq m$. A result is returned in $n = (m/D + \theta)$ clock cycles, where θ is a small number of cycles required for combinatorial logic and storage. As D increases, so too does the area required by the architecture. Digit-serial multipliers are used in this research as they can be used to satisfy different application requirements by varying the digit size.

Consider the multiplication $c = a.b$, where $a, b, c \in \mathbb{F}_{2^m}$. The process begins by separating b into bit strings of length D. There are n such strings, where n is the value given by rounding m/D upward to the nearest integer. Let $b = \sum_{i=0}^{m-1} b_i x^i$. The polynomial is first rewritten as

$$b = \sum_{j=0}^{n-1} \tilde{b}_j x^{jD} = \tilde{b}_0 + \tilde{b}_1 x^D + \tilde{b}_2 x^{2D} + \ldots \tilde{b}_{n-1} x^{(n-1)D} \tag{4.15}$$

where all $\tilde{b}_j$ are digits such that

$$\tilde{b}_j = \sum_{l=0}^{D-1} b_{jD+l} x^l \tag{4.16}$$

and all $b_j \in \mathbb{F}_2$.

Multiplication of a by b using the digit-serial method is performed by multiplying the a polynomial by the digits of b, accumulating the results and reducing modulo f:

$$c = a.b = \big(a.(\sum_{j=0}^{n-1} \tilde{b}_j x^{jD})\big) \mod f \tag{4.17}$$

$$= \big(\sum_{j=0}^{n-1} a.\tilde{b}_j x^{jD}\big) \mod f \tag{4.18}$$

A multiplication $a.\tilde{b}_j$ is known as a digit-word multiplication (the polynomial a of length m is known as a word). These multiplications can be performed in one clock cycle in hardware by multiplying a digit of b by each bit of a simultaneously. These multiplications are reduced modulo f before accumulation to minimise operational complexity and storage requirements. Digit-word multiplication can be performed efficiently using matrix-vector methods. Consider the multiplication of $a = \sum_{i=0}^{m-1} a_i x^i$ by a digit of b. A composition polynomial is first computed:

$$z = \sum_{i=0}^{m-1} \sum_{l=0}^{D-1} a_i . b_{jD+l} x^{i+l} \tag{4.19}$$

The polynomial z has degree $m + D - 2$ and its coefficients are members of $\mathbb{F}_2$. Let $\mathbf{z}$ be a vector of length $m + D - 1$ containing the value of the coefficients of z, i.e. $\mathbf{z}^T = [z_0, z_1, z_2, \ldots z_{m+D-2}]$. The polynomial a can also be represented by a vector, $\mathbf{a}$, of length m. An $(m+D-1)\times m$ product matrix $\mathbf{B}$ can be generated that represents the $\mathbb{F}_2$ operations that are required during the composition stage. The $\mathbf{z}$ vector is returned by the matrix-vector multiplication of $\mathbf{B}$ by $\mathbf{a}$. All of the operations are on $\mathbb{F}_2$. The product matrix is used to generate a network of AND and XOR gates that implement the composition stage in hardware.

The result returned by the composition stage must be reduced modulo f if its degree is greater than $m - 1$. This reduction can also be performed using matrix-vector methods. Let $z = \sum_{s=0}^{u-1} z_s x^s$, where $u > m - 1$ and let $\mathbf{z}$ be a vector of length u containing the coefficients of z. A reduction matrix $\mathbf{R}$, of size $m \times u$, can be created that represents the $\mathbb{F}_2$ additions required for reduction. A network of XOR gates that returns the reduced value of the digit-word product is generated using this matrix. More information on the construction and use of these matrices is provided in [86].

The $\mathbb{F}_{2^m}$ multiplication of a by b can be implemented in hardware according to Equation (4.18). The degree of the accumulating polynomial increases by D after each digit-word multiplication. A straightforward implementation would require a relatively large area to store and reduce the resulting bit strings. If the results of the digit-word composition stages are, however, shifted left by D bits after computation and immediately reduced modulo f, then the degree of the accumulating polynomial will not exceed $m-1$. Another matrix can be used to generate the network of XOR gates required to reduce the polynomials of $(m+2D-1)$ bits returned by digit-word composition and shifting. The equation governing

$\mathbb{F}_{2^m}$ multiplication using this method is

$$c = a.b = \Big(\sum_{j=0}^{n-1} \big(x^{jD}(a \times \tilde{b}_j)\big) \mod f\Big) \mod f \tag{4.20}$$

where $a \times \tilde{b}_j$ represents the composition stage of a digit word multiplication $a.\tilde{b}_j$. The operations required for the hardware implementation of Equation (4.20) are presented as Algorithm 7.

Algorithm 7 $\mathbb{F}_{2^m}$ Digit-Serial Multiplication

INPUT: $a, b \in \mathbb{F}_{2^m}$, where $a = \sum_{i=0}^{m-1} a_i x^i, b = \sum_{j=0}^{n-1} \tilde{b}_j x^{jD}$ and $\tilde{b}_j = \sum_{l=0}^{D-1} b_{jD+l} x^l$

OUTPUT: $c = a.b$, where $c \in \mathbb{F}_{2^m}$

1: Initialise: $z^{(0)} \leftarrow 0$, $b^{(0)} \leftarrow b$

2: **for** $(i \leftarrow 1, i \leq n-1, i \leftarrow i+1)$ **do**

3: $\quad z^{(i)} \leftarrow x^D(z^{(i-1)} + \tilde{b}_{d-1}^{(i-1)} \times a) \mod f$

4: $\quad b^{(i)} \leftarrow x^D b^{(i-1)}$

5: **end for**

6: $c \leftarrow (z^{(d-1)} + \tilde{b}_{d-1} \times a) \mod f$

RETURN: c

The hardware architecture used for digit serial multiplication is illustrated in Figure 4.1. The polynomial a is stored in an m-bit register and b is stored in an m-bit shift register with a D-bit output. On each iteration, the a polynomial is sent to the multiplication block with a digit of b and a composition string of size $(m + D - 1)$ bits is calculated. The worst case propagation delay for this block is 1 AND gate and $[\log_2 D - 1]$ XOR gates. The lower m bits of the digit-word composition result are then added to the m-bit accumulation polynomial computed during the previous iteration. The result is shifted to the left by D bits and enters the reduction block. This unit consists of a network of XOR gates that is constructed according to the reduction matrix. The worst case propagation delay of this block is dependent on the field size and the irreducible polynomial. A low weight irreducible polynomial reduces the complexity and propagation delay of this network significantly. The output of the reduction block is stored in an m-bit register. The result $c = a.b$ is available at the output of this register on completion of the *for* loop. A *Finite State Machine* (FSM) provides all of the signals that are required to handle the operations. The digit-serial multiplication architecture produces a result in $n = m/D + 2$ clock cycles.

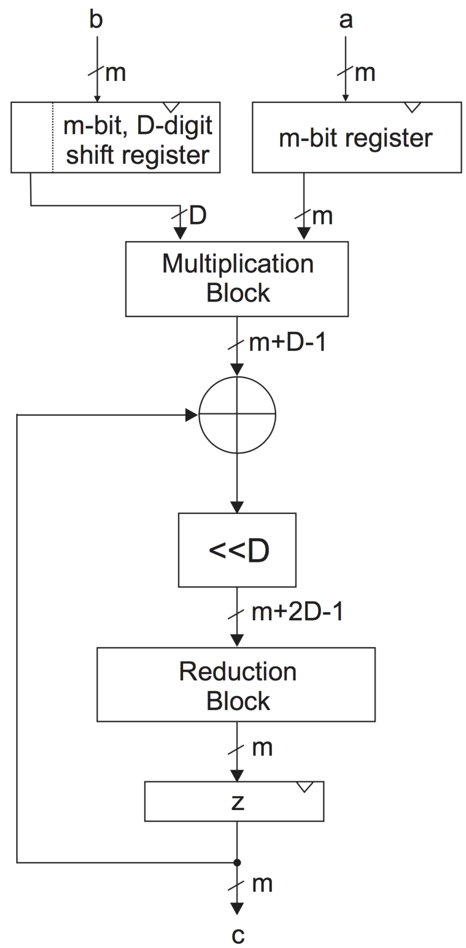

Figure 4.1: $\mathbb{F}_{2^m}$ Digit-Serial Multiplier Architecture

The VHDL for the multiplication architecture is generated in the design environment using *C++* code. Variables are used to represent the field size, irreducible polynomial and digit size. Multipliers can be created automatically on the input of these parameters. This means that pairing processors, each containing multipliers with different digit sizes, can be generated with ease. This enables detailed analysis of the capabilities and resource requirements of various architectural solutions to pairing computation.

4.3.3 $\mathbb{F}_{2^m}$ Squaring

Squaring of an element $a \in \mathbb{F}_{2^m}$ can be achieved using the digit-serial multiplication architecture. This operation can, however, be performed much more efficiently by employing the techniques described by Wu in [87]. The use of their methods means that squaring can be implemented with a purely combinatorial unit.

Consider $a = \sum_{i=0}^{m-1} a_i x^i$ with all $a_i \in \mathbb{F}_2$. Since $0.0 = 0$ and $1.1 = 1$ on $\mathbb{F}_{2^m}$, then $(a_i)^2 = a_i$. This means that a composition polynomial z can be calculated according to

$$z = (\sum_{i=0}^{m-1} a_i x^i)^2 = \sum_{i=0}^{m-1} a_i^2 x^{2i} = \sum_{i=0}^{m-1} a_i x^{2i} \tag{4.21}$$

This is performed in hardware by placing a zero between each bit of a, which results in a bit string of length $2m-1$. This string, which represents the composition polynomial, is reduced modulo the irreducible polynomial using a network of *XOR* gates defined by a reduction matrix. A low weight irreducible polynomial also reduces the complexity of this network. The VHDL for the squaring unit is generated in software according to the desired field size and irreducible polynomial. Since squaring requires combinatorial logic alone a result is returned in one clock cycle.

4.3.4 $\mathbb{F}_{2^m}$ Inversion

Inversion is typically more expensive than all other operations on $\mathbb{F}_{2^m}$. Fermat's Little Theorem can be used to perform inversion. An element $c = b^{-1}$, where $b, c \in \mathbb{F}_{2^m}$, can be computed using the relationship $b = b^{2m}$ for all $b \in \mathbb{F}_{2^m}$. This means that $b^{-1} = b^{2m-2}$. This exponentiation can be performed using a series of squaring and multiplications. A hardware architecture that uses a generalised version of the Itoh-Tsujii inversion method [88] is presented in [89]. The computation time required is, however, in the order of m

multiplications, which means that it is sometimes unsuitable for the relatively large field sizes used in pairing-based applications.

Inversion can also be performed using the *Extended Euclidean Algorithm* (EEA). This is the most efficient method to perform hardware inversion [90]. The EEA is used to compute the *Greatest Common Divisor* (GCD) of 2 polynomials $a, b \in \mathbb{F}_{2^m}$. The EEA also returns a pair of polynomials $w, u \in \mathbb{F}_{2^m}$ such that $GCD(a, b) = wa + ub$. The irreducible polynomial f generating the field is not divisible by any polynomial in $\mathbb{F}_{2^m}$, which means that

$$GCD(f, b) = wf + ub = 1 \tag{4.22}$$

Also, since $f = 0 \in \mathbb{F}_{2^m}$, then $ub = 1$ and therefore $b^{-1} = u$. This forms the basis for the EEA-based inversion algorithm of [91], presented in more detail in [90]. The calculation of the inverse of an element $b \in \mathbb{F}_{2^m}$ using the EEA is described by Algorithm 8. Operations are performed on the polynomial pairs (r, s) and (u, v), where $r, s, u, v \in \mathbb{F}_{2^m}$. On initialisation, (r, s) is set to (b, f) and (u, v) to $(1, 0)$. The *for* loop is iterated $2m$ times. An integer δ is used to track the differences between the degrees of r and s. The relationships between the polynomial pairs remains the same throughout the loop. By accumulating the (u, v) polynomials in this manner, the inverse of b is given by the value of the u polynomial after $2m$ iterations.

In hardware, the (u, v) polynomial pair can be updated in parallel to (r, s) since they are computed independently of each other. The inverter described in [90] uses this parallelism to perform $\mathbb{F}_{2^m}$ inversion and is used in this work. The architecture contains hardware units that are known as *bit-slices*. Each bit-slice is dedicated to updating a specific bit of a polynomial pair. A chain of RS slices, with each slice containing the same combinatorial logic, returns the final (r, s) polynomial pair in $2m$ clock cycles. A chain of UV slices is also used to return the final value of (u, v) in $2m$ cycles. An architecture containing a chain of $(m + 1)$ RS slices and a parallel chain of m UV slices performs an $\mathbb{F}_{2^m}$ inversion in $2m$ clock cycles.

It would be interesting to investigate whether the area cost of the inverter is justified by

Algorithm 8 $\mathbb{F}_{2^m}$ Inversion

INPUT: $b = \sum_{i=0}^{m-1} b_i x^i \in \mathbb{F}_{2^m}$

OUTPUT: $c = b^{-1}$, where $u \in \mathbb{F}_{2^m}$

Initialise: $r^{(0)} \leftarrow b, s^{(0)} \leftarrow f, u^{(0)} \leftarrow 1, v^{(0)} \leftarrow 0, \delta^{(0)} \leftarrow 0$

for $(i \leftarrow 1,\, i \leq 2m,\, i \leftarrow i+1)$ **do**

  if $r_m^{(i-1)} = 0$ **then**

    $r^{(i)} \leftarrow x(r^{(i-1)})$

    $s^{(i)} \leftarrow s^{(i-1)}$

    $u^{(i)} \leftarrow x(u^{(i-1)})$ mod f

    $v^{(i)} \leftarrow v^{(i-1)}$

    $\delta^{(i)} \leftarrow \delta^{(i-1)} + 1$

  else

    if $\delta^{(i-1)} = 0$ **then**

      $r^{(i)} \leftarrow x(s^{(i-1)} + s_m^{(i-1)} r^{(i-1)})$

      $s^{(i)} \leftarrow r^{(i-1)}$

      $u^{(i)} \leftarrow x(v^{(i-1)} + s_m^{(i-1)} u^{(i-1)})$ mod f

      $v^{(i)} \leftarrow u^{(i-1)}$

      $\delta^{(i)} \leftarrow \delta^{(i-1)} + 1$

    else

      $r^{(i)} \leftarrow r^{(i-1)}$

      $s^{(i)} \leftarrow x(s^{(i-1)} - s_m^{(i-1)} r^{(i-1)})$

      $u^{(i)} \leftarrow (u^{(i-1)})/x$ mod f

      $v^{(i)} \leftarrow v^{(i-1)} + s_m^{(i-1)} u^{(i-1)}$

      $\delta^{(i)} \leftarrow \delta^{(i-1)} - 1$

    end if

  end if

end for

$c \leftarrow u$

RETURN: c

the reduction in clock cycles required to perform the pairing. The inverter is utilised only during the exponentiation. Fermat's Little Theorem can be used to define how many times the field member to be inverted should be iteratively squared and multiplied with itself to return a result. The dedicated inversion architecture is also desirable in other curve-based cryptographic systems.

4.4 Hardware Implementation of Arithmetic on $\mathbb{F}_{2^{4m}}$

Extension field addition, multiplication, squaring, inversion and exponentiation to q are required to perform Tate pairing computation according to Algorithms 5 and 6. This section describes the hardware architectures that have been used to implement these operations.

An element $a \in \mathbb{F}_{2^{4m}}$ is represented by a degree 3 polynomial of form

$$a = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{4.23}$$

where all $a_i \in \mathbb{F}_{2^m}$. The $\mathbb{F}_{2^{4m}}$ field is generated by the degree 4 irreducible polynomial $p = x^4 + x + 1$.

4.4.1 $\mathbb{F}_{2^{4m}}$ Addition

The addition operation is performed bitwise on each coefficient of the polynomials. Let $a, b, c \in \mathbb{F}_{2^{4m}}$. Then

$$c = a + b = \sum_{i=0}^{3} a_i x^i + \sum_{i=0}^{3} b_i x^i = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + (a_3 + b_3)x^3 \tag{4.24}$$

Addition on $\mathbb{F}_{2^{4m}}$ is relatively trivial and can be computed in one clock cycle using a total of $4m$ *XOR* gates.

4.4.2 $\mathbb{F}_{2^{4m}}$ Multiplication

Multiplication on $\mathbb{F}_{2^{4m}}$ is required on each iteration of the *for* loop of Algorithm 5. Multiplication on the extension field is, however, relatively expensive and efforts must be made to ensure that it is performed as efficiently as possible.

Consider $a, b, c \in \mathbb{F}_{2^{4m}}$. The extension field multiplication $c = a.b$ can be performed by multiplying each coefficient of a by each coefficient of b and reducing the resulting composition polynomial. This method requires 16 $\mathbb{F}_{2^m}$ multiplications and nine $\mathbb{F}_{2^m}$ additions during the composition stage alone. *Karatsuba multiplication* [92] can instead be used to reduce the number of $\mathbb{F}_{2^m}$ multiplications at the expense of more trivial $\mathbb{F}_{2^m}$ additions. Nine $\mathbb{F}_{2^m}$ multiplications are first performed:

$$\begin{aligned}
mul_0 =& a_0.b_0 \\
mul_1 =& a_1.b_1 \\
mul_2 =& a_2.b_2 \\
mul_3 =& a_3.b_3 \\
mul_4 =& (a_0 + a_1).(b_0 + b_1) \\
mul_5 =& (a_0 + a_2).(b_0 + b_2) \\
mul_6 =& (a_1 + a_3).(b_1 + b_3) \\
mul_7 =& (a_2 + a_3).(b_2 + b_3) \\
mul_8 =& (a_0 + a_1 + a_2 + a_3).(b_0 + b_1 + b_2 + b_3)
\end{aligned}$$

Seven partial products are now calculated using these results. This stage requires no further $\mathbb{F}_{2^m}$ multiplications:

$$
\begin{aligned}
\gamma_0 =& mul_0 \\
\gamma_1 =& mul_0 + mul_1 + mul_4 \\
\gamma_2 =& mul_0 + mul_1 + mul_2 + mul_5 \\
\gamma_3 =& mul_0 + mul_1 + mul_2 + mul_3 + mul_4 + mul_5 + mul_6 + mul_7 + mul_8 \\
\gamma_4 =& mul_1 + mul_2 + mul_3 + mul_6 \\
\gamma_5 =& mul_2 + mul_3 + mul_7 \\
\gamma_6 =& mul_3
\end{aligned}
$$

These partial products are reduced modulo the irreducible polynomial $p = x^4 + x + 1$. This can be performed using the matrix reduction techniques described in the previous section. The irreducible polynomial is, however, fixed in this case. This means that the following additions can be used to reduce the partial products and return the final result:

$$
\begin{aligned}
c_0 =& mul_0 + mul_1 + mul_2 + mul_3 + mul_6 \\
c_1 =& mul_0 + mul_4 + mul_6 + mul_7 \\
c_2 =& mul_0 + mul_1 + mul_5 + mul_7 \\
c_3 =& mul_0 + mul_1 + mul_2 + mul_4 + mul_5 + mul_6 + mul_7 + mul_8
\end{aligned}
$$

A dedicated hardware architecture for extension field multiplication using the Karatsuba method has been created for this work and is illustrated in Figure 4.2. The unit takes as input the m-bit coefficients of the input operands. All nine $\mathbb{F}_{2^m}$ multiplications are performed in parallel in hardware. The digit-serial architectures described in Subsection 4.3.2 are used to perform $\mathbb{F}_{2^m}$ multiplication. The $\mathbb{F}_{2^{4m}}$ additions are nested, when possible, and a network of 22 adders is used. This architecture returns an $\mathbb{F}_{2^{4m}}$ multiplication result in $(m/D + 2)$ clock cycles.

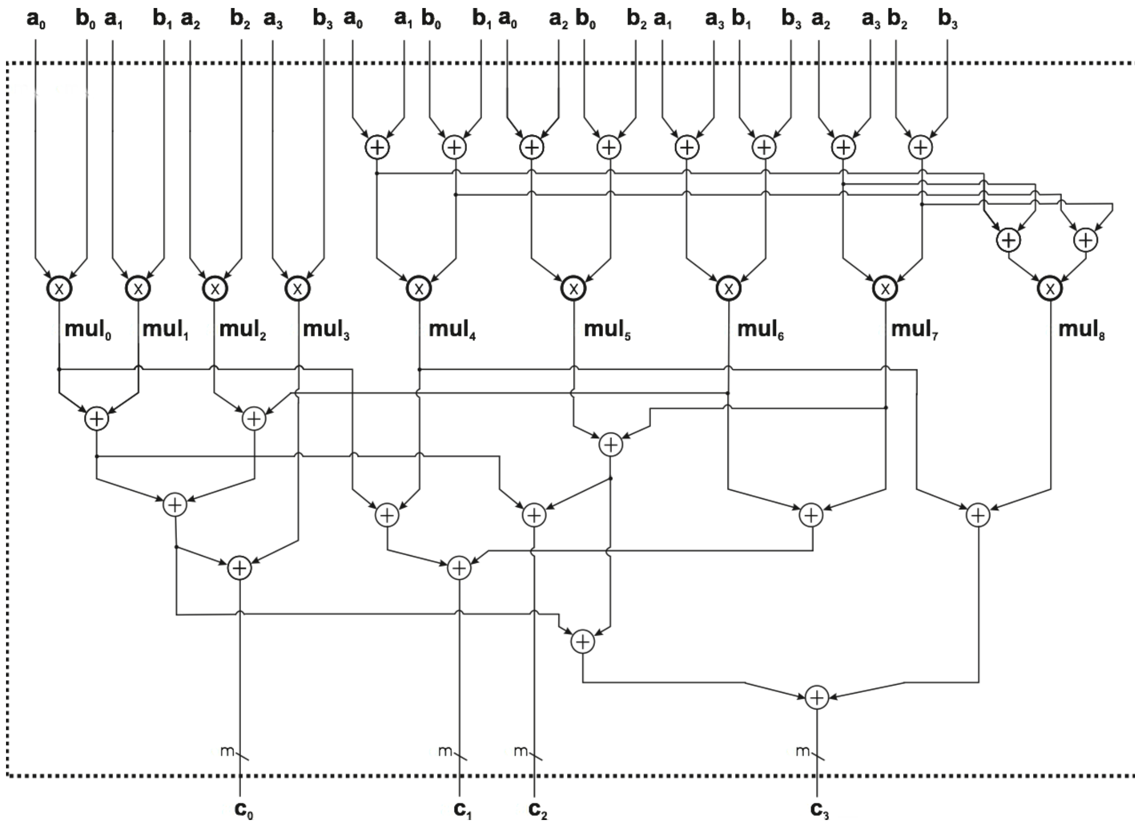

Figure 4.2: Architecture for $\mathbb{F}_{2^{4m}}$ Karatsuba multiplication

4.4.3 $\mathbb{F}_{2^{4m}}$ Squaring

Consider $a = \sum_{i=0}^{3} a_i x^i$ and $c = \sum_{i=0}^{3} c_i x^i$, where $a, c \in \mathbb{F}_{2^{4m}}$. The computation $c = a^2$ begins with the composition of an intermediate polynomial z:

$$z = \sum_{i=0}^{3} {a_i}^2 x^{2i} = {a_0}^2 + {a_1}^2 x^2 + {a_2}^2 x^4 + {a_3}^2 x^6 \tag{4.25}$$

This polynomial must now be reduced. The irreducible polynomial has value $p = 0 \in \mathbb{F}_{2^{4m}}$, which means that $x^4 = x + 1$, $x^5 = x^2 + x$ and $x^6 = x^3 + x^2$. Substituting into Equation

(4.25) gives:

$$
\begin{aligned}
c =& ({a_0}^2 + {a_1}^2x^2 + {a_2}^2x^4 + {a_3}^2x^6) \mod p && (4.26)\\
=& {a_0}^2 + {a_1}^2x^2 + {a_2}^2(x+1) + {a_3}^2(x^3+x^2) && (4.27)\\
=& ({a_0}^2 + {a_2}^2) + {a_2}^2x + ({a_1}^2 + {a_3}^2)x^2 + {a_3}^2x^3 && (4.28)
\end{aligned}
$$

Squaring on $\mathbb{F}_{2^{4m}}$ requires four $\mathbb{F}_{2^m}$ squaring units and two $\mathbb{F}_{2^m}$ adders. The architecture that has been designed for the computation of squaring on the extension field is displayed in Figure 4.3.

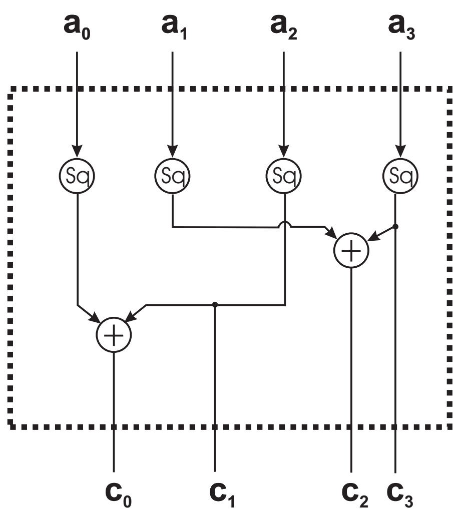

Figure 4.3: Architecture for Squaring on $\mathbb{F}_{2^{4m}}$

4.4.4 Exponentiation to q

Consider $a \in \mathbb{F}_{2^{4m}}$ where $a = \sum_{i=0}^{k-1} a_i x^i$ and all $a_i \in \mathbb{F}_{2^m}$. Exponentiation of a to $q = 2^m$ can be implemented with m extension field squaring operations. As a is squared, however, a pattern emerges that can be used to considerably reduce the number of required field operations. It can be easily shown that a^{2^5} is equal to a^{2^1}, a^{2^6} is equal to a^{2^2} and a^{2^7} is equal to a^{2^3}. This sequence continues every four steps. This means that a^{2^m} is dependent on the value of $m \mod 4$. The relationships $a^{2^0} = a$ for $m \mod 4 = 0$, $a^{2^1} = a^2$ for m

mod $4 = 1$, $a^{2^2} = a^2$ for $m \mod 4 = 2$ and $a^{2^3} = a^3$ for $m \mod 4 = 3$ hold. Recall also that $a_i^{2^m} = a_i$ for all $a_i \in \mathbb{F}_{2^m}$. The value of $c = a^q$, where $c = \sum_{i=0}^{k-1} c_i x^i$ is returned according to Table 4.2.

Table 4.2: Calculation of $c = a^q$, where $a, c \in \mathbb{F}_{2^{4m}}$ for different values of $m \mod 4$

Output	**m mod 4**			
	0	1	2	3
c_0	a_0	$a_0 + a_2$	$a_0 + a_1 + a_2 + a_3$	$a_0 + a_1$
c_1	a_1	a_2	$a_1 + a_3$	$a_2 + a_3$
c_2	a_2	$a_1 + a_3$	$a_2 + a_3$	a_1
c_3	a_3	a_3	a_3	a_3

Exponentiation to q can, therefore, be implemented in hardware using XOR gates and a result returned in one clock cycle.

4.4.5 $\mathbb{F}_{2^{4m}}$ Inversion

The final exponentiation, described by Algorithm 6, requires a multiplicative inversion on $\mathbb{F}_{2^{4m}}$. Inversion is the most complex field operation. This subsection describes the algorithmic and hardware optimisations that can be employed to implement extension field inversion.

Inversion on any extension field $\mathbb{F}_{n^k}$, where n and k are positive integers, can be performed using a variant of the EEA. An efficient method for extension field inversion based on the EEA was presented by Lim and Hwang in [93], in which several $\mathbb{F}_n$ inversions are replaced by less costly $\mathbb{F}_n$ multiplications. The field operations required to perform $\mathbb{F}_{n^k}$ inversion are presented in Algorithm 9. The algorithm takes as input $a \in \mathbb{F}_{n^k}$ and the irreducible polynomial p generating $\mathbb{F}_{n^k}$ and returns $b \in \mathbb{F}_{n^k}$ such that $b = a^{-1}$. In order to clarify the notation, consider line 9. The term $g_{deg(g)} \cdot p_{deg(p)-1}$ denotes the $\mathbb{F}_n$ multiplication of the uppermost coefficient of g by the second to uppermost coefficient of p.

Algorithm 9 $\mathbb{F}_{n^k}$ Inversion

INPUT: $a = \sum_{i=0}^{k-1} a_i x^i \in \mathbb{F}_{n^k}$, where $a_i \in \mathbb{F}_n$, and irreducible polynomial p generating $\mathbb{F}_{n^k}$

OUTPUT: $b = \sum_{i=0}^{k-1} b_i x^i = a^{-1} \in \mathbb{F}_{n^k}$, where $b_i \in \mathbb{F}_n$

1: Initialise: $b \leftarrow 0,\ c \leftarrow 1,\ g \leftarrow a$
2: **while** $deg(p) \neq 0$ **do**
3:     **if** $deg(p) < deg(g)$ **then**
4:         Swap p with g and b with c
5:     **end if**
6:     $j \leftarrow deg(p) - deg(g)$
7:     $\alpha \leftarrow {g_{deg(g)}}^2$
8:     $\beta \leftarrow p_{deg(p)} \cdot g_{deg(g)}$
9:     $\gamma \leftarrow g_{deg(g)} \cdot p_{deg(p)-1} - p_{deg(p)} \cdot g_{deg(g)-1}$
10:     $p \leftarrow \alpha.p - (\beta.x^j + \gamma.x^{j-1}).g$
11:     $b \leftarrow \alpha.b - (\beta.x^j + \gamma.x^{j-1}).c$
12:     **if** deg(p)=deg(g) **then**
13:         $p \leftarrow g_{deg(p)} . p - p_{deg(p)} . g$
14:         $b \leftarrow g_{deg(p)} . b - p_{deg(p)} . c$
15:     **end if**
16: **end while**
17: $p_0 \leftarrow {p_0}^{-1}$
18: $b \leftarrow p_0.b$

RETURN: b

The main calculation cost of Algorithm 9 lies in the operations required within the *while* loop that begins on Line 2. On each iteration, the variables α, β and γ must be computed and the values of p and b updated, requiring several $\mathbb{F}_n$ field operations, the most costly of which are the multiplications. If the degree of p is equal to the degree of g at the end of an iteration, further operations are required. On completion of the loop, $p_0 \in \mathbb{F}_n$ is inverted and multiplied by $b \in \mathbb{F}_{n^k}$. The *while* loop can be implemented in hardware using a finite state machine, with the degree of p and the degree of g acting as control lines. The control scheme would also have to provide a means for implementation of the required if statements and *swap* operations.

An efficient technique that is also more amenable to hardware implementation can be developed by fixing the extension field in question. The sparse nature of an irreducible polynomial can result in a dramatic reduction in the number of required operations. The conditional statements of Algorithm 9 can also be removed and a sequence of field operations explicitly stated. This simplifies the corresponding hardware control scheme significantly.

Table 4.3: Computational steps for $\mathbb{F}_{2^{4m}}$ Inversion

Var.	**Init.**	**Step 1**	**Step 2**	**Step 3**
α	-	${g_3}^2$	${p_2}^2$	${p_1}^2$
β	-	g_3	$g_3.p_2$	$g_2.p_1$
γ	-	g_2	$g_2.p_2 + g_3.p_1$	$g_1.p_1 + g_2.p_0$
p_4	1	0	0	0
p_3	0	0	0	0
p_2	0	$\beta.g_1 + {g_2}^2$	0	0
p_1	1	$\alpha + \beta.g_0 + \gamma.g_1$	$\alpha.g_1 + \beta.p_0 + \gamma.p_1$	0
p_0	1	$\alpha + \gamma.g_0$	$\alpha.g_0 + \gamma.p_0$	$\alpha.g_0 + \gamma.p_0$
b_3	0	0	0	$\beta.b_2$
b_2	0	0	$\beta.p_1$	$\beta.b_1 + \gamma.b_2$
b_1	0	β	$\beta.p_0 + \gamma.b_1$	$\alpha.c_1 + \beta.b_0 + \gamma.b_1$
b_0	0	γ	$\alpha + \gamma.b_0$	$\alpha.c_0 + \gamma.b_0$
g_3	a_3	a_3	0	-
g_2	a_2	a_2	p_2	-
g_1	a_1	a_1	p_1	-
g_0	a_0	a_0	p_0	-
c_1	0	0	b_1	-
c_0	1	1	b_0	-

In the case of $\mathbb{F}_{2^{4m}}$, the *while* loop is replaced by a sequence of operations that can be grouped into three computational steps. Table 4.3 demonstrates the $\mathbb{F}_{2^m}$ operations required during each step. The p, b, g and c polynomials are initialised before the first step. Note that the degree 2 and 3 coefficients of c are omitted since they are not operands of any operation and, therefore, do not need to be updated. On each step, the values of α, β and γ are first calculated. The p, b, g and c polynomials are then updated. On the first computational step, the variables β and γ are initialised and γ is calculated with an $\mathbb{F}_{2^m}$ squaring operation. Subsequently, p is calculated, requiring four multiplications and four additions on $\mathbb{F}_{2^m}$. The b polynomial is updated with a simple reassignment. The g and c polynomials remain unchanged. Note that the degree of p is reduced by 2 during this step, due to the sparse nature of the irreducible polynomial defining the field. On step 2, the α, β and γ variables are calculated, this time requiring three multiplications, one squaring and one addition on $\mathbb{F}_{2^m}$. The p polynomial is updated, which requires five $\mathbb{F}_{2^m}$ multiplications and three additions, and b is updated with four $\mathbb{F}_{2^m}$ multiplications and two additions. After this g and c are allocated the values of p and b at the end of Step 1, respectively. On the third step the α, β and γ variables are updated, requiring three $\mathbb{F}_{2^m}$ multiplications, one addition and one squaring. The p polynomial is updated with two multiplications and one addition on $\mathbb{F}_{2^m}$ and b is updated with eight multiplications and four additions. No further steps are necessary as the degree of p has reached 0. The three steps of Table 4.3 require a total of 29 multiplications, four squarings and 16 additions on $\mathbb{F}_{2^m}$. These operations satisfy the calculations required by the *while* loop of Algorithm 9. After this sequence, $p_0 \in \mathbb{F}_{2^m}$ is inverted and multiplied by $b \in \mathbb{F}_{2^{4m}}$. The latter operation requires four $\mathbb{F}_{2^m}$ multiplications.

The most costly operations are the multiplications. All other operations can be performed combinatorially in hardware. A hardware implementation provides an opportunity to perform many of the 33 $\mathbb{F}_{2^m}$ multiplications in parallel. The quantity of multipliers returning the most efficient computation was determined through analysis of clock cycle and area data. It was found that three multipliers, operating in parallel, provide an excellent solution for the computation of the multiplications.

The scheduling of the 33 multiplicative operations through three multipliers is shown in Table 4.4 (the data in the table itself points towards increasing redundancy within the multipliers as the number of units grows). The stage at which the $\mathbb{F}_{2^m}$ inversion begins is

Table 4.4: Scheduling of $\mathbb{F}_{2^m}$ multiplications through three multipliers for $\mathbb{F}_{2^{4m}}$ inversion

Stage	**Mult 0**	**Mult 1**	**Mult 2**	**Inv**
1	$\beta.g_1$	$\beta.g_0$	-	-
2	$\gamma.g_1$	$\gamma.g_0$	-	-
3	$g_3.p_2$	$g_2.p_2$	$g_3.p_1$	-
4	$\alpha.g_1$	$\beta.p_0$	$\gamma.p_1$	-
5	$\gamma.g_0$	$\gamma.p_0$	$\beta.b_1$	-
6	$\beta.p_0$	$\gamma.b_1$	$\gamma.b_0$	-
7	$p_1.g_1$	$g_2.p_0$	-	-
8	$\alpha.g_0$	$\gamma.p_0$	$g_2.p_1$	-
9	$\beta.b_2$	$\beta.b_1$	$\gamma.b_2$	$inv(p_0^{-1})$
10	$\alpha.c_1$	$\beta.b_0$	$\gamma.b_1$	
11	$\alpha.c_0$	$\gamma.b_0$	-	$ret(p_0^{-1})$
12	$p_0^{-1}.b_0$	$p_0^{-1}.b_1$	$p_0^{-1}.b_2$	-
13	$p_0^{-1}.b_3$	-	-	-

also noted. Extension field inversion is completed in a total of 13 stages. Stages 1-8 and 12-13 require $n = m/D$ clock cycles each, where D is the digit size of the $\mathbb{F}_{2^m}$ multipliers. The $\mathbb{F}_{2^m}$ inversion of p_0 begins at Stage 9. This operation can be performed in parallel with the multiplications of Stages 9-11. Stage 12 cannot, however, begin until the value of p_0^{-1} is returned by the inverter, which returns a result after $2m$ clock cycles. This means that Stages 9-11 require the larger of $3n$ or $2m$ clock cycles. All other field operations are combinatorial. An $\mathbb{F}_{2^m}$ inverter using three multipliers, therefore, returns a result in $13n$ clock cycles if $3n > 2m$ or in $10n + 2m$ clock cycles otherwise. The redundancy of the multipliers is very low. The first multiplier is in use 100% of the time, while the second and third multipliers are in use 92% and 62% of the time, respectively. All other options are considerably less efficient.

The techniques discussed in this subsection form the basis of a co-authored publication that appeared in [7]. The paper describes the FPGA implementation of the $\mathbb{F}_{2^{4m}}$ inverter. The architecture consists of an Arithmetic Logic Unit (ALU), dedicated RAM and an uncomplicated control system. The ALU contains an $\mathbb{F}_{2^m}$ inverter, two adders, a squaring unit and three multipliers. The control system sequences the arithmetic operations, accesses the RAM and sends the m-bit buses representing the $\mathbb{F}_{2^m}$ elements to the requisite logic units. The system also controls the logic units and stores their outputs in the correct place in memory, when available. The inverter was prototyped on the extension field $\mathbb{F}_{2^{4m}}$, where $m = 283$, and then used to optimise a Tate pairing processor that was previously described by Keller et al. in [94]. The inverter leads to a dramatic reduction in pairing computation time. The minimum Tate pairing computation time, without the use of the inverter, is $2ms$ assuming that there are no area constraints and a clock frequency of $40MHz$. The minimum computation time required with the inclusion of the extension field inverter is $0.69ms$ if the same parameters are used. The author of this thesis provided the VHDL and the files necessary to implement the inversion architecture. Help was also provided during the integration of the architecture with the Tate pairing processor.

Since extension field inversion is only required once during the computation of the Tate pairing (using the η_T method) the inclusion of three multipliers dedicated to inversion alone would not be prudent. Fortunately, other multipliers within the processor architecture can be accessed if care is taken with the routing of buses and the design of top level control systems. As will be seen in the next section, three parallel $\mathbb{F}_{2^m}$ multipliers are used to implement the special multiplication known as *smul* in Algorithm 5. These multipliers are not in use during exponentiation. The inverter is placed within an exponentiation unit that has access to these multipliers. This unit will be discussed in the next section and the hardware implementation of extension field inversion further clarified.

4.5 Dedicated Hardware Units for Pairing Computation

Computation of the η_T pairing followed by exponentiation to the Tate pairing, is performed according to Algorithms 5 and 6, respectively. The most expensive component of these algorithms is the *for* loop of the η_T computation. This loop iterates $(m+1)/4$ times and it is, therefore, paramount that its field operations be performed as efficiently as possible. The *for* loop of Algorithm 5 can be described by four computation stages:

1. The calculation of t and the roots and squares of x_P, y_P, x_Q, y_Q as calculated on Steps 4, 6, 7 and 9 of the algorithm, respectively.
2. The calculation of u_0 and u_1 according to Steps 5 and 8.
3. The computation of $u_0.u_1$ on Step 10. This multiplication is performed using a special function, called *smul*, that takes advantage of the structure of the polynomials.
4. The $\mathbb{F}_{2^{4m}}$ multiplication $mill.u$ on Step 11.

Dedicated hardware units have been designed to ensure a fast computation of these four stages. An exponentiation unit has also been created for the implementation of the computations required by Algorithm 6. It should also be noted that Steps 14-16 of Algorithm 5 require computations on $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^{4m}}$, but these operations need only be performed once and can be implemented by reusing other units. The custom hardware units that have been created for the characteristic 2 elliptic curve Tate pairing processor described in this chapter are as follows:

- ***Precomputation Unit:*** Many $\mathbb{F}_{2^m}$ squaring and rooting operations are required during the *for* loop. The inclusion of dedicated units for all of these operations would be wasteful and the control scheme necessary to supply the correct values to other hardware blocks, when required, would not be trivial. This unit computes all required squares and roots before iteration of the *for* loop. The desired $\mathbb{F}_{2^m}$ precomputation values are supplied to the other units, as required.

- ***Unit for the parallel calculation of*** u_0 ***and*** u_1**:** The use of the unrolled *for* loop means that u_0 and u_1 can be calculated at the same time. This unit performs this parallel calculation efficiently.

- ***Unit for the calculation of*** $smul(u_0, u_1)$***:*** The custom multiplication routine *smul* is implemented in this unit and the value of $u = smul(u_0, u_1)$ returned.

- $\mathbb{F}_{2^{4m}}$ ***Multiplication Unit:*** The *mill* variable must be updated by means of an $\mathbb{F}_{2^{4m}}$ multiplication by u on each iteration. Extension field multiplication is an expensive operation. A dedicated unit is included in the pairing processor that utilises the Karatsuba multiplication method. This unit was discussed in Subsection 4.4.2.

- ***Exponentiation Unit:*** This unit performs the exponentiation from the η_T to the Tate pairing according to Algorithm 6.

The most efficient interconnection of these units was carefully studied during the design phase of the Tate pairing processor. A scheme was created to handle control signals and data buses and to sequence the required operations. The functional units are considered individually in this section. Their hierarchy and connectivity at the top level of the processor will be described in Section 4.6.

4.5.1 Precomputation Unit

Squares of the values of $x_Q, y_Q \in \mathbb{F}_{2^m}$ must be computed on Lines 6 and 9 of Algorithm 5. Square roots of $x_P, y_P \in \mathbb{F}_{2^m}$ are required by lines 4, 7 and 13. As seen in Subsection 4.3.3, squaring is relatively trivial in hardware. Square rooting is, however, more costly. Fortunately, the latter operation is not required if precomputation methods are employed. In this processor, the values of $x_P^{2^i}$ and $y_P^{2^i}$, for all $0 < i < m$, are computed and stored in an indexed memory array before the *for* loop begins. Since the $\mathbb{F}_{2^m}$ field is cyclic, the roots required on each iteration of the loop can be obtained by accessing the squares from memory in reverse order. The values of $x_Q^{2^i}$ and $y_Q^{2^i}$ are also precomputed before the loop.

As mentioned previously, the u_0 and u_1 polynomials of Lines 5 and 8 of Algorithm 5 can be computed in parallel in hardware. This requires some rearranging of the operations performed on Lines 4-9 and the introduction of some extra variables. As an example, consider the second iteration of the loop. The operations and variables of Lines 4-9 can be related to the original pairing input points as follows:

$$\begin{aligned} t_0 \leftarrow & x_P^{\frac{1}{4}} + 1, x_{P_0} \leftarrow {x_P}^{\frac{1}{8}}, y_{P_0} \leftarrow {y_P}^{\frac{1}{8}}, x_{Q_0} \leftarrow x_Q^4, y_{Q_0} \leftarrow y_Q^4 \\ t_1 \leftarrow & {x_P}^{\frac{1}{8}} + 1, x_{P_1} \leftarrow {x_P}^{\frac{1}{16}}, y_{P_1} \leftarrow {y_P}^{\frac{1}{16}}, x_{Q_1} \leftarrow {x_Q}^8, y_{Q_1} \leftarrow {y_Q}^8 \\ u_0 \leftarrow & (t_0.(x_{P_0} + x_{Q_0} + 1) + y_{P_0} + y_{Q_0}) + (t_0 + x_{Q_0} + 1)x \\ u_1 \leftarrow & (t_1.(x_{P_1} + x_{Q_1} + 1) + y_{P_1} + y_{Q_1}) + (t_1 + x_{Q_1} + 1)x \end{aligned}$$

The values of $x_{P_0}, x_{P_1}, y_{P_0}, y_{P_1}, x_{Q_0}, x_{Q_1}, y_{Q_0}, y_{Q_1}$ are the powers of the pairing input coordinates that are required on each iteration of the loop. A modular architecture has been created for precomputation that supplies all of the values that are required during a parallel computation of u_0 and u_1. The x_P precomputation module is illustrated in Figure 4.4. This module contains $m \times m$ bit dual port block RAM along with an $\mathbb{F}_{2^m}$ squaring unit and an m-bit multiplexer. The multiplexer is used at the input of the RAM to allow storage of either the initial value of x_P or the result at the output of the squaring unit. A control system at the top level of the pairing processor employs a counter to write the squares to memory. The RAM address connections are controlled by the $ind0$ and $ind0 - 1$ buses, the latter of which represents the integer value of *ind0* minus the integer one. The calculation and storage of these squares requires $(m + 2)$ clock cycles. Once precomputation is complete the *for* loop can begin. Using appropriate indices, all of the powers of x_P that are required for the parallel computation of u_0 and u_1 can be supplied simultaneously on each iteration of the loop.

The top level precomputation unit contains modules for each of the input coordinates and is illustrated in Figure 4.5. It receives as input the m-bit values of x_P, y_P, x_Q and y_Q.

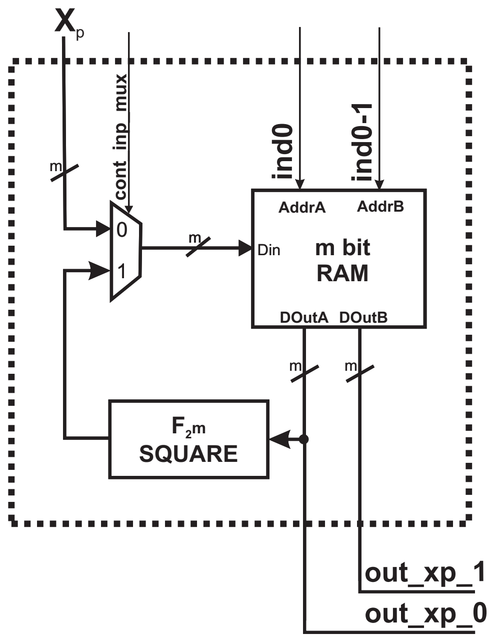

Figure 4.4: Precomputation module for x_P

It is controlled by a pair of index buses and a 1-bit signal. The first index (*ind*0) is used to access the RAM within the x_P and y_P modules and the other (*ind*1) to access the RAM within the x_Q and y_Q modules. The 1-bit signal *cont_inp_mux* is used to handle the multiplexers within each of the precomputation modules. The three control signals are used to handle both precomputation and the supply of powers of coordinates during loop iteration. The unit has two $4m$-bit output buses. The *Dout*0 bus supplies the powers required for the computation of u_0, whilst *Dout*1 supplies the powers for the computation of u_1. This architecture means that all powers are calculated in parallel and are available after a total of $(m + 2)$ clock cycles.

4.5.2 Unit for the Computation of u_0 and u_1

On completion of the precomputation stage, the *for* loop begins and u_0 and u_1 must be computed. On inspection of Algorithm 4, it can be seen that the u polynomial has a special form before loop unrolling. It can be written as $u = c_0 + c_1x + (c_1 + 1)x^2$, where

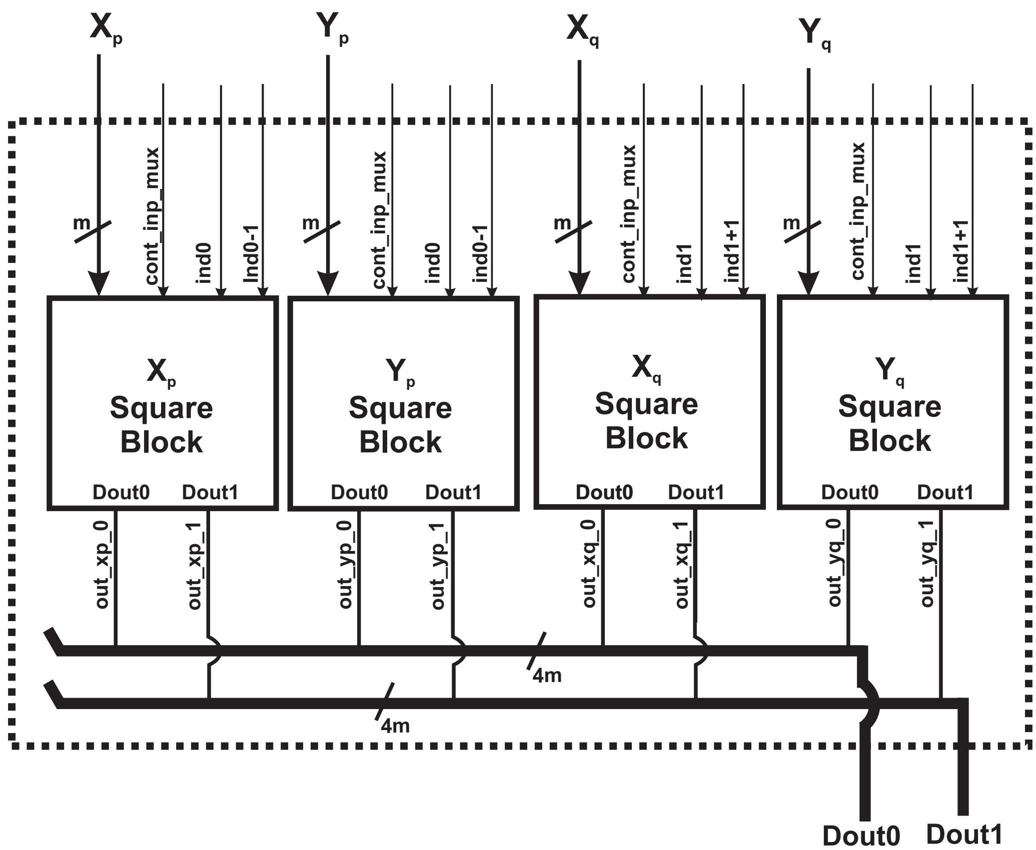

Figure 4.5: Precomputation unit for the characteristic 2 elliptic curve Tate pairing processor

all $c_i \in \mathbb{F}_{2^m}$. After the loop has been unrolled u_0 and u_1 are of the same structure. Let $u_0 = a_0 + a_1x + (a_1 + 1)x^2$ and $u_1 = b_0 + b_1x + (b_1 + 1)x^2$, where all $a_i, b_i \in \mathbb{F}_{2^m}$. The coefficients of x^2 in u_0 and u_1 do not need to be computed since they can be trivially retrieved from a_1 and b_1 when $smul(u_0, u_1)$ must be performed. This results in a saving of $2m$-bits of storage and reduces the output bus sizes of the u_0 and u_1 computation unit by a total of $2m$ bits. This is reflected in Lines 5 and 8 of Algorithm 5, in which the x^2 coefficients of u_0 and u_1 are not computed.

The hardware unit that was designed for the parallel computation of u_0 and u_1 is illustrated in Figure 4.6. The required powers of the input coordinates are supplied by the precomputation unit on a pair of $4m$-bit input buses. Two separate but similar branches compute u_0 and u_1 simultaneously. Each branch contains one $\mathbb{F}_{2^m}$ multiplier and four $\mathbb{F}_{2^m}$ adders. An extra m-bit register is required in the upper branch since t_0 is dependent on the value of x_{P_1} from the previous iteration. The operations required to compute u_0 and u_1 are very similar to those required to compute the f and u polynomials of Algorithm 5 (Lines 2 and 14). This means that this unit can also be used to calculate f and u. These polynomials are computed in the upper branch. Some extra 1-bit adders and 1-bit multiplexers are required since the least significant bits of these polynomials differ to those of u_0. The $Dout0$ output bus supplies either the lower $2m$ bits of the u_0 polynomial or the $3m$-bit f and u polynomials that must be computed outside of the loop. The lower $2m$ bits of u_1 are produced on $Dout1$. The hardware unit contains a total of two $\mathbb{F}_{2^m}$ multipliers, eight $\mathbb{F}_{2^m}$ adders, an m-bit register, seven 1-bit adders and two 1-bit multiplexers. The u_0 and u_1 polynomials are computed in a total of $(m/D + 2)$ clock cycles, where m is the field size and D is the digit size of the multipliers.

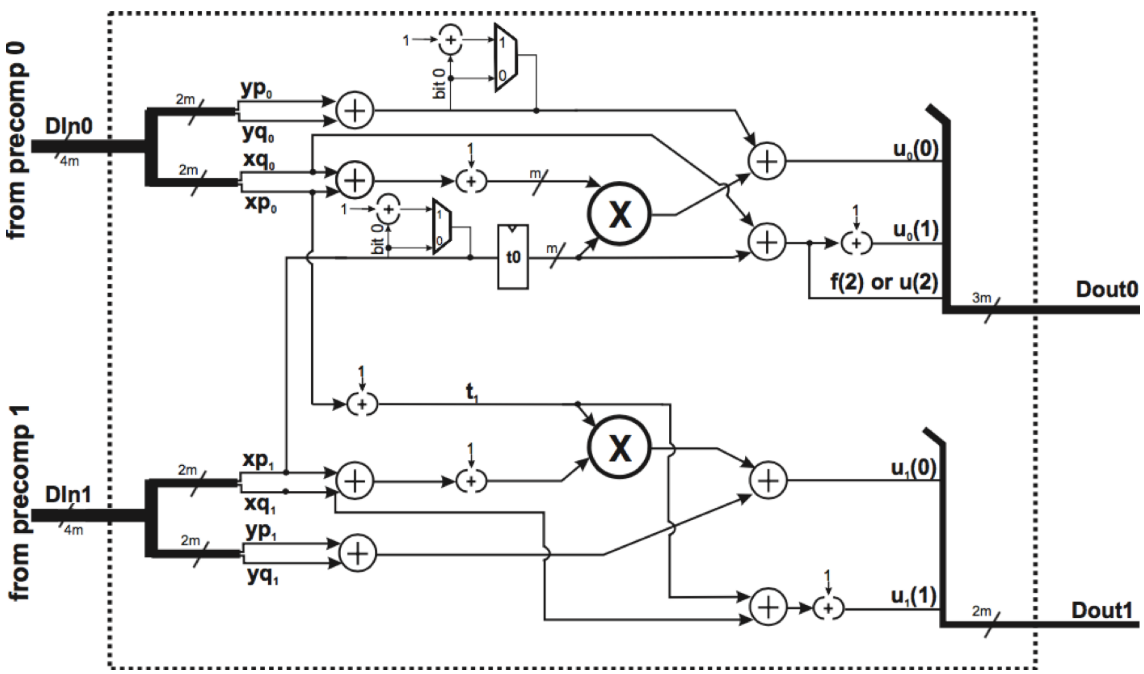

Figure 4.6: Unit for the computation of u_0 and u_1

4.5.3 Unit for the Computation of $smul(u_0, u_1)$

Once computed, the u_0 and u_1 polynomials must be multiplied together. This can be performed using degree 2 Karatsuba multiplication, which would require six $\mathbb{F}_{2^m}$ multiplications and a number of additions. A specialised multiplication routine can, however, be defined that requires fewer multiplications. This routine, called $smul(u_0, u_1)$, takes advantage of the special form of u_0 and u_1.

Consider $u_0 = a_0 + a_1x + (a_1 + 1)x^2$ and $u_1 = b_0 + b_1x + (b_1 + 1)x^2$, where all $a_i, b_i \in \mathbb{F}_{2^m}$. The multiplication $u_0.u_1$ proceeds in the usual fashion. A composition polynomial is first computed:

$$\begin{aligned}
z &= (a_0 + a_1x + (a_1 + 1)x^2).(b_0 + b_1x + (b_1 + 1)x^2)\\
&= a_0.b_0 + a_0.b_1x + a_0.(b_1 + 1)x^2\\
&\quad + a_1.b_0x + a_1.b_1x^2 + a_1.(b_1 + 1)x^3\\
&\quad + b_0.(a_1 + 1)x^2 + b_1.(a_1 + 1)x^3 + (a_1 + 1).(b_1 + 1)x^4\\
&= a_0.b_0 + (a_0.b_1 + a_1.b_0)x + (a_0.b_1 + a_0 + a_1.b_1 + a_1.b_0 + b_0)x^2\\
&\quad + (a_1 + b_1)x^3 + (a_1.b_1 + a_1 + b_1 + 1)x^4
\end{aligned}$$

The z polynomial is now reduced modulo $x^4 + x + 1$, the irreducible polynomial defining the extension field. The reduced result is given by:

$$\begin{aligned}
u_0.u_1 = {} & (a_0.b_0 + a_1.b_1 + a_1 + b_1 + 1) + (a_0.b_1 + a_1.b_0 + a_1.b_1 + a_1 + b_1 + 1)x\\
& + (a_0.b_1 + a_0 + a_1.b_1 + a_1.b_0 + b_0)x^2 + (a_1 + b_1)x^3
\end{aligned}$$

The *smul* routine can now be constructed. Let $mul_0 = (a_0.b_0)$, $mul_1 = (a_1.b_1)$ and $mul_2 = (a_0 + a_1).(b_0 + b_1) = a_0.b_0 + a_0.b_1 + a_1.b_0 + a_1.b_1$. Substituting into the previous equation and cancelling terms when possible gives the $\mathbb{F}_{2^m}$ operations required to compute

$u_0.u_1 = smul(u_0, u_1)$:

$$\begin{aligned} u_0.u_1 = &(mul_0 + mul_1 + a_1 + b_1 + 1) \\ &+ (mul_0 + mul_2 + a_1 + b_1 + 1)x \\ &+ (mul_0 + mul_2 + a_0 + b_0)x^2 \\ &+ (a_1 + b_1)x^3 \end{aligned}$$

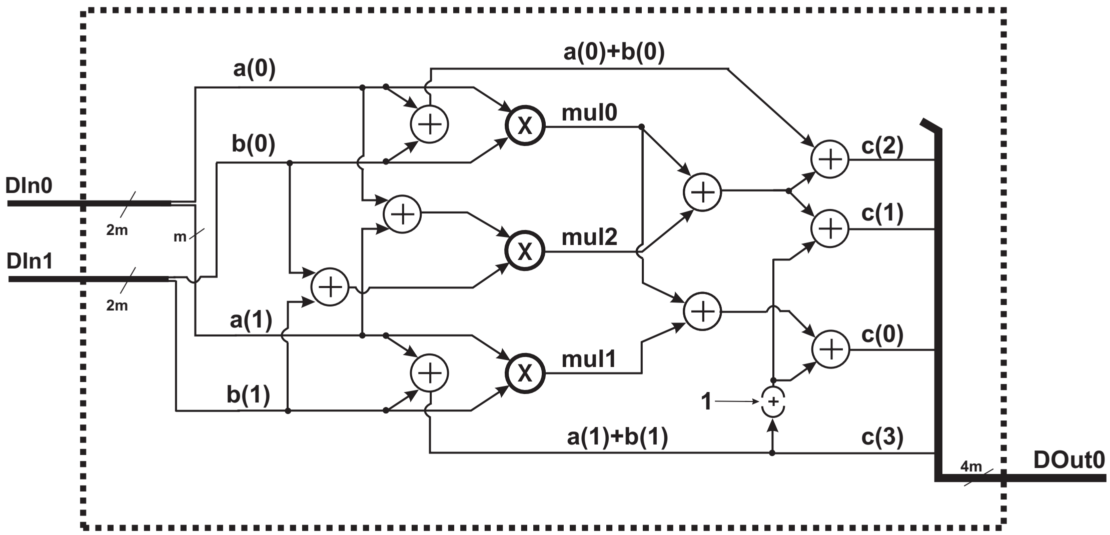

Figure 4.7: Unit for the computation of $smul(u_0, u_1)$

The hardware architecture that has been designed for the computation of $smul(u_0, u_1)$ is presented in Figure 4.7. The $2m$-bit inputs $DIn0$ and $DIn1$ are supplied by the (u_0, u_1) computation unit and contain the values of a_0, a_1 and b_0, b_1 respectively. The $\mathbb{F}_{2^m}$ multiplications mul_0, mul_1 and mul_2 are performed in parallel. The unit contains a total of three $\mathbb{F}_{2^m}$ multipliers, nine $\mathbb{F}_{2^m}$ adders and a 1-bit adder. The $4m$-bit output bus $DOut0$ contains the value of $smul(u_0, u_1)$ after computation. A result is returned in $(m/D + 2)$ clock cycles, where m is the field size and D is the digit size of the multipliers.

4.5.4 Exponentiation Unit

The η_T result of Algorithm 5 is followed by an exponentiation to a unique Tate pairing value. The relationship between the pairings on characteristic 2 elliptic curves was already defined in Section 4.2 and is given by

$$\eta_T(P,Q)^{MT} = \langle P, \psi(Q)\rangle_N^M \tag{4.29}$$

where $M = (2^{2m}-1)(2^m - 2^{(m+1)/2}+1)$ and $T = 2^{(m+1)/2}-1$ in the $m \mod 8 \equiv 1$ case.

Exponentiation to MT is performed according to Algorithm 6. A total of $(m+1)/2$ $\mathbb{F}_{2^{4m}}$ squarings, five $\mathbb{F}_{2^{4m}}$ multiplications, six $\mathbb{F}_{2^{4m}}$ exponentiations to $q = 2^m$ and one $\mathbb{F}_{2^{4m}}$ inversion is required. Most of the extension field operations require operands that are not available until the previous algorithmic step has completed. This means that the algorithm provides little scope for parallelism. The design of an exponentiation unit containing large dedicated arithmetic modules would not be resource efficient as far fewer operations are required in comparison to the η_T computation. Fortunately, the exponentiation is computed after iteration of the *for* loop of Algorithm 5. This means that the hardware units that have been created for the implementation of that loop can be reused. The $\mathbb{F}_{2^{4m}}$ multiplications can be performed by sending the input operands to the extension field multiplication architecture described in Subsection 4.4.2. As seen in Subsection 4.4.5, the most efficient implementation of extension field inversion arises from the use of three parallel $\mathbb{F}_{2^m}$ multipliers. The *smul* unit contains three multipliers that operate in parallel and are used to perform the $\mathbb{F}_{2^m}$ multiplications required for $\mathbb{F}_{2^{4m}}$ inversion.

The hardware unit that has been designed for the implementation of exponentiation is presented in Figure 4.8. The unit has two input data buses. The first is $Din0$, a $3m$-bit bus that is connected to the outputs of the three $\mathbb{F}_{2^m}$ multipliers of the *smul* unit. The second, $Din1$, is $4m$-bits in length and is connected to the output of the $\mathbb{F}_{2^{4m}}$ multiplication unit. Arithmetic modules for extension field squaring and powering to q are included and require only combinatorial logic. These modules are combinatorial in nature and are inexpensive in terms of area. The unit also contains an $\mathbb{F}_{2^{4m}}$ inversion module. This module does not perform every operation that is required for extension field inversion. It contains an $\mathbb{F}_{2^m}$ adder, an $\mathbb{F}_{2^m}$ squaring unit and an $\mathbb{F}_{2^m}$ inverter. It also handles $Dout2$, the $6m$-bit output bus that is used to send operands to the external *smul* multipliers.

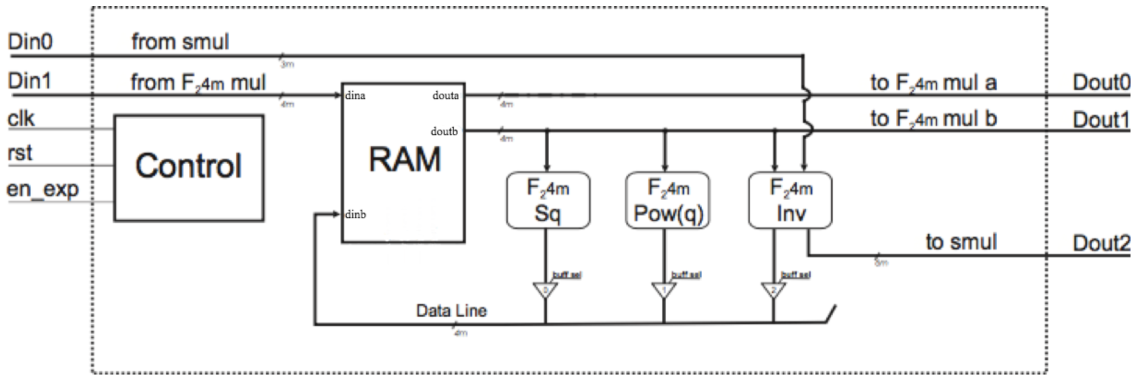

Figure 4.8: Characteristic 2 Elliptic Curve Exponentiation Unit

The extension field variables of Algorithm 6 are stored in dual port $4m$-bit RAM. The first input port is directly connected to the output of the external $\mathbb{F}_{2^{4m}}$ multiplier. The arithmetic modules receive their inputs from the second $4m$-bit RAM output. The outputs of the arithmetic modules are each connected to a tri-state buffer. A control signal is used to select the $4m$-bit result to be stored in RAM. The RAM output ports are also connected to the $DOut0$ and $DOut1$ outputs. These $4m$-bit buses are sent to the inputs of the external $\mathbb{F}_{2^{4m}}$ multiplier. A control system takes as input *clk*, *rst* and *en_exp* signals from the top level processor. This system contains a finite state machine that outputs the signals that are required to enable reads and writes from and to the RAM and to control the logic units.

Exponentiation begins when the *en_exp* control signal is set. The η_T pairing result is located at the output of the $\mathbb{F}_{2^{4m}}$ multiplier at this time and is immediately stored in RAM. The $(m+1)/2$ $\mathbb{F}_{2^{4m}}$ squarings, required by Line 3 of Algorithm 6, are first performed and the result stored in RAM. The u and v elements are then powered to q using the $\mathbb{F}_{2^{4m}}$ $pow(q)$ module. Their values are then sent to the external $\mathbb{F}_{2^{4m}}$ multiplier and the control system waits the correct number of clock cycles for a result. Subsequently v is again sent through the $pow(q)$ module and w and v are multiplied according to Lines 7 and 8. The

$\mathbb{F}_{2^{4m}}$ inversion of w can be performed in parallel with the operations of Lines 9-11. This means that the extension field inversion can be performed in parallel with the powerings to q and the two extension field multiplications of Lines 9-11. An $\mathbb{F}_{2^{4m}}$ multiplication of c by the result of the inversion is required on Line 13. On completion of this operation the Tate pairing value is available at the output of the extension field multiplier.

This type of architecture provides a very efficient way to implement exponentiation as use is made of resource intensive units that would otherwise be redundant.

4.6 The Characteristic 2 Elliptic Curve Pairing Processor

The hardware processor that has been created for accelerated pairing computation in the characteristic 2 elliptic curve case is presented in Figure 4.9. An η_T computation is followed by an exponentiation to return a Tate pairing value. The processor receives the input points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ on the $2m$-bit buses *DIn0* and *DIn1*. The clock and the *rst* and *en* control signals are supplied by an external source. The Tate result $\langle P, \psi(Q)\rangle_N^M$ is returned on the $4m$-bit *DOut0* output bus on completion of pairing computation.

4.6.1 Operation Scheduling

Field operations are performed in the $\mathbb{F}_{2^{4m}}$ multiplication unit and the four custom hardware units. These units, discussed as separate systems in Subsection 4.4.2 and in the previous section, each perform a specific function. The more general motivation behind the design of the units can now be explained in the context of the top level of the architecture. The *for* loop of Algorithm 5 iterates $(\frac{m-1}{4} - 1)$ times. Every iteration consists of three main computational stages: the calculation of u_0 and u_1, the computation of $u = smul(u_0, u_1)$, and the $\mathbb{F}_{2^{4m}}$ multiplication $mill = mill.u$. The hardware units each

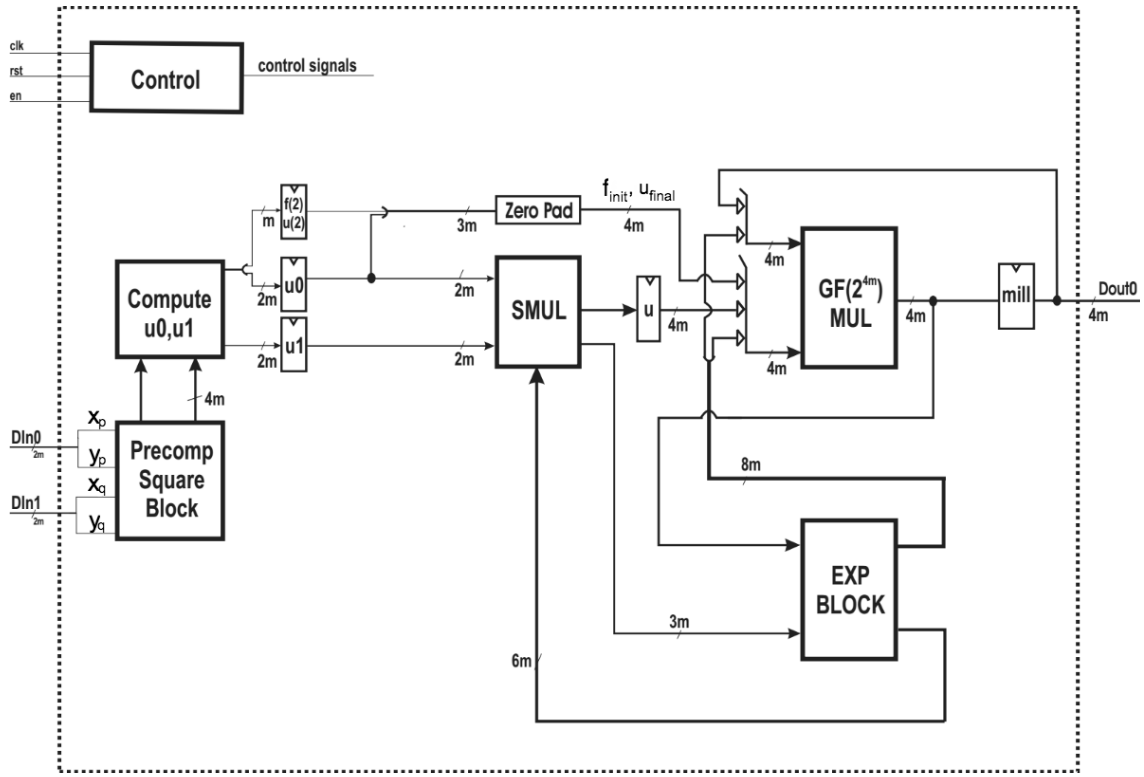

Figure 4.9: The Characteristic 2 Elliptic Curve Tate Pairing Processor

complete their assigned computation in $(m/D+2)$ cycles. A small number of clock cycles, denoted ζ, is also required for register writes and for the propagation of signals. Completion of the *for* loop would, therefore, require $(\frac{m-1}{4} - 1)(3m/D + 2 + \zeta)$ clock cycles if each iteration were considered in isolation. The processor and its units have, however, been designed so that the computational stages of different iterations of the loop can be performed in parallel.

The $(u_0,\, u_1)$ computation unit receives its inputs from the precomputation block and does not rely on results from any previous iteration. Let $(u_{0,1})_{i=0}$ be the values of u_0 and u_1 that are calculated on the first iteration. Once $(u_{0,1})_{i=0}$ have been stored in the registers at its output, the $(u_0,\, u_1)$ computation unit can be reset and the computation of $(u_{0,1})_{i=2}$ can begin almost immediately. When $(u_{0,1})_{i=0}$ are available, the *smul* unit computes $u_{i=0} = smul\big((u_{0,1})_{i=0}\big)$ and the result is stored at its output. The *smul* unit is now reset and the computation of $u_{i=2} = smul\big((u_{0,1})_{i=2}\big)$ can begin immediately. This is possible since the $(u_{0,1})_{i=2}$ polynomials have been calculated in parallel with the $u_{i=0}$ computation. The value of $mill_{i=0} = 1.u_{i=0}$ is now computed in the $\mathbb{F}_{2^{4m}}$ multiplication unit and stored at its output. The multiplication $mill_{i=2} = mill_{i=0}.u_{i=2}$ can begin immediately after this since $u_{i=2}$ has been calculated in parallel with $mill_{i=0}$. Computation proceeds in this manner until the value of the *mill* variable at the end of the *for* loop has been returned. The calculation of f and u, according to Lines 2 and 14 of Algorithm 5, respectively can also be performed in parallel with other operations near the end of the *for* loop. The scheduling of operations through the hardware units is detailed in Table 4.5.

Table 4.5: Scheduling of the *for* loop of Algorithm 5 through the Tate pairing processor

$(u0,\, u1)$ **Unit**	***smul*** **Unit**	$\mathbb{F}_{2^{4m}}$ **Mult. Unit**
$(u_{0,1})_{i=0}$	———	———
$(u_{0,1})_{i=2}$	$u_{i=0} = smul\big((u_{0,1})_{i=0}\big)$	———
$(u_{0,1})_{i=4}$	$u_{i=2} = smul\big((u_{0,1})_{i=2}\big)$	$mill_{i=0} = 1.u_{i=0}$
$(u_{0,1})_{i=6}$	$u_{i=4} = smul\big((u_{0,1})_{i=4}\big)$	$mill_{i=2} = mill_{i=0}.u_{i=2}$
$(u_{0,1})_{i=8}$	$u_{i=6} = smul\big((u_{0,1})_{i=6}\big)$	$mill_{i=4} = mill_{i=2}.u_{i=4}$
$(u_{0,1})_{i=10}$	$u_{i=8} = smul\big((u_{0,1})_{i=8}\big)$	$mill_{i=6} = mill_{i=4}.u_{i=6}$
.	.	.
.	.	.
.	.	.
$(u_{0,1})_{i=\frac{m-1}{2}-3}$	$u_{i=\frac{m-1}{2}-5} = smul\big((u_{0,1})_{i=\frac{m-1}{2}-5}\big)$	$mill_{i=\frac{m-1}{2}-7} = mill_{i=\frac{m-1}{2}-9}.u_{i=\frac{m-1}{2}-7}$
$(u_{0,1})_{i=\frac{m-1}{2}-1}$	$u_{i=\frac{m-1}{2}-3} = smul\big((u_{0,1})_{i=\frac{m-1}{2}-3}\big)$	$mill_{i=\frac{m-1}{2}-5} = mill_{i=\frac{m-1}{2}-7}.u_{i=\frac{m-1}{2}-5}$
f	$u_{i=\frac{m-1}{2}-1} = smul\big((u_{0,1})_{i=\frac{m-1}{2}-1}\big)$	$mill_{i=\frac{m-1}{2}-1} = mill_{i=\frac{m-1}{2}-5}.u_{i=\frac{m-1}{2}-3}$
u	–	$mill_{i=\frac{m-1}{2}-1} = mill_{i=\frac{m-1}{2}-3}.u_{i=\frac{m-1}{2}-1}$

A total of $(\frac{m-1}{4} + 1)$ steps are required. Each step is completed in $(m/D + \theta)$ clock cycles, where θ is a small number of cycles required for control and for the propagation of signals through all of the registers and combinatorial logic in the data chain. A further two $\mathbb{F}_{2^{4m}}$ multiplications are required by Lines 15 and 16 of Algorithm 5. Precomputation is performed in $(m + 2)$ clock cycles. This means that an η_T result can be returned in $\big(m + 2 + (\frac{m-1}{4} + 3)(m/D + \theta)\big)$ clock cycles.

4.6.2 Architectural Overview

Tate pairing computation begins when *rst* is toggled and *en* is set. A control system containing an FSM outputs all of the signals that are required to manage the arithmetic and storage units and to handle bus selection.

The $2m$-bit pairing input points P and Q are stored in the precomputation block on initialisation. All squares and square roots are then computed and stored. The control unit ensures that the required precomputation results are available at the output of the unit during the *for* loop.

The u_0 and u_1 polynomials are computed and stored in two $2m$-bit registers at the output of the (u_0, u_1) computation unit. The u_0 branch of this unit is also used to calculate the f and u polynomials outside of the loop and an extra m-bit register is required to store the degree 2 coefficients. The outputs of the $u0$ and $u1$ registers are sent to the *smul* unit. The result in the m-bit register is combined with the output of the $u0$ register, padded and sent to the $\mathbb{F}_{2^{4m}}$ multiplication unit since f and u must be multiplied by the *mill* variable at the end of the algorithm.

The *smul* unit is used to compute u during the *for* loop and the result is stored in a $4m$-bit register at its output. It also receives a $6m$-bit input bus from the exponentiation unit as its $\mathbb{F}_{2^m}$ multipliers are used during extension field inversion. The $\mathbb{F}_{2^m}$ multiplication results are sent to the exponentiation unit on a $3m$-bit bus.

The $\mathbb{F}_{2^{4m}}$ multiplication unit returns the product of the two $4m$-bit buses at its input. The result is stored in the $4m$-bit *mill* register at its output and is also sent to the exponentiation unit. Various $\mathbb{F}_{2^{4m}}$ multiplications must be performed during Algorithms 5 and 6. Tri-state buffers are used for input selection as the buses are relatively wide at this location and a multiplexer system would introduce unacceptably large combinatorial delays. These buffers are also used to minimise resource usage as they are available in every FPGA slice. The first multiplication input is selected from either the output of the *mill* register or from the upper half of an $8m$-bit bus that is sent from the exponentiation unit. The second input is selected from the (f, u) bus, the u register or from the lower $4m$-bits of the exponentiation bus. This routing and selection system enables the computation of all extension field multiplications while minimising the required area.

The η_T pairing result is located at the output of the *mill* unit on completion of all of the computations required by Algorithm 5. The exponentiation unit is used to perform the field operations of Algorithm 6. It has access to the three $\mathbb{F}_{2^m}$ multipliers of the *smul* unit and to the $\mathbb{F}_{2^{4m}}$ multiplier. An internal control system is used to manage the field operations performed by its own logic modules and by the external multipliers. A $4m$-bit Tate pairing result is produced at the output of the *mill* register on completion of the exponentiation.

4.6.3 Results and Comparisons

The characteristic 2 elliptic curve Tate pairing processor was captured in VHDL and implemented on a Virtex-II Pro FPGA (xc2vp100-6ff1696). This FPGA contains 44,096 slices. The VHDL is generated by the *C++* design system and receives input variables such as field size, irreducible polynomial and $\mathbb{F}_{2^m}$ multiplier digit size from the user. This means that the processor can be modified with ease if security, clock cycle or area requirements change.

Implementations of the processor on an elliptic curve $E(\mathbb{F}_{2^m})$, where $m = 313$, were used to gather speed and area data. The irreducible polynomial defining the field is $x^{313} + x^{79} + 1$. The extension field is $k \times m = 1252$ bits in size. The security level of an application that uses this implementation is roughly equivalent to 1024-bit RSA. Results for implementations using multiplier digit sizes of 1, 4, 8 and 12 are presented in Table 4.6. The *Area* column lists the number of slices required by each version whilst the *Utilisation* column lists the percentage of the FPGA that is occupied in each case. The

Clock Cycles column lists the number of clock cycles required by the processor to compute $\eta_T(P, Q)^{MT} = \langle P, \psi(Q) \rangle_N^M$ in each case. The *AC Product* column contains products of clock cycles and area usage. The AC products are measured in *slice.cycles* and can be used as an indication of efficiency.

Table 4.6: Results returned by the characteristic 2 Tate pairing processor for $m = 313$

D (Bits)	**Area** (Slices)	**Utilisation**	**Clock Cycles**	**AC Product** (Mslice.cycles)
1	28435	64%	17340	493
4	34675	79%	11165	387
8	41078	93%	6200	255
12	44060	99%	4818	212

The $D = 12$ case returns a Tate pairing result in 4,818 clock cycles. The AC product in this case is 212 Mslice.cycles. This is a 17% reduction on the $D = 8$ case. However, the $D = 8$ case provides a much larger reduction of 34% over the $D = 4$ case. This indicates that the $D = 8$ case may be the better option as the extra area required by the $D = 12$ case may incur a drop in clock frequency that does not compensate for the relatively small reduction in clock cycles over the $D = 8$ case. The processors occupy a large percentage of the FPGA. As utilisation increases, unrelated logic must be packed into slices and some slices must be used by the Xilinx tool to enable an efficient place and route process. This can result in a decreased in the maximum clock frequency. It would be interesting to investigate whether slice usage would decrease on larger FPGAs on which the routing matrix could be used more efficiently to place and route the design.

Barreto et al. perform a software computation of the η and η_T pairings in [63]. Pairings are computed on a Pentium IV processor operating at 3 GHz. Results are returned (without exponentiation to the Tate pairing) in 5.83 ms and 3 ms in the η and η_T cases respectively. The custom processor described in this chapter provides a significant reduction in pairing computation time, even for relatively low clock frequencies.

Results returned by other hardware implementations of characteristic 2 elliptic curve pairing processors are listed in Table 4.7. As discussed previously, the security of pairing-based systems can be measured in terms of the cost of the index calculus attack in the extension field. This means that the size of the extension field can be used to compare the security level returned by different implementations. The implementations of [8] and [95] both use an extension field of 1132 bits. It should be kept in mind while analysing the results that this is smaller than the 1252 bits of our implementations.

Table 4.7: Results returned by characteristic 2 elliptic curve pairing implementations in the literature

Ref.	**m**	**Alg.**	**Device**	**Arch.**	**D** (Bits)	**Area** (Slices)	**Cycles**	**AC** (MSl.cycles)
[8]	283	BKLS	Virtex-II	Macro	4	27411	68250	1871
					6	29421	55110	1621
				Mi. 1	6	4273	240000	1026
				Mi. 9	6	15065	120000	1808
[95]	283	η	Virtex-II P (PPR)	Macro	16	22726	7308	166
					32	37803	4392	166
		η_T			16	33252	4368	145

In [8], Keller et al. compute the Tate pairing using the BKLS algorithm described in [60]. Two types of processor, which they call *macro* and *micro*, are presented. The macro processor contains dedicated, hard-wired, logic units for extension field operations. An implementation on a Virtex-II FPGA with $m = 283$ and $D = 6$ returns a Tate pairing in 55,110 clock cycles. The custom processor provides an 11.4x speed up on this. A total of 29,421 slices are required, resulting in an AC product of 1621 Mslice.cycles. The AC product of the processor discussed in this chapter is only 13.1% of this value. The *micro* processor does not contain any extension field logic units. A number of $\mathbb{F}_{2^m}$ logic units, including digit-serial multipliers, are used. This processor can be implemented with various quantities of parallel $\mathbb{F}_{2^m}$ multipliers. An implementation with nine multipliers, each with a digit size of 6, requires 120,000 clock cycles for computation. The custom processor is 24.9x faster. The implementation uses 15,065 slices and has an AC product of 1,808. The custom processor's AC product is only 11.7% of this value.

Shu et al. [95] discuss a Tate pairing implementation using the η and η_T methods. This was published contemporaneously with much of the subject matter of this chapter. They use a more traditional architecture with a main controller, central memory, interconnection networks and a central ALU. The ALU contains one extension field multiplier and a number of subfield arithmetic units. Their results are generated from information returned by the place and route tools of the Xilinx software, rather than from actual FPGA implementations. They also use a subfield size of 283 bits in comparison to the 313 bits used here. These factors make it difficult to perform a direct comparison but a brief analysis follows. Their processor returns a Tate pairing in 4,368 clock cycles using $\mathbb{F}_{2^m}$ multipliers of digit size 16, a speed up of 1.1x in comparison. A total of 33,252 slices are required, which results in an AC product of 145 Mslice.cycles. This is a 30% reduction in comparison to 212 Mslice.cycles. It will be seen in Chapter 6 that the ALU-based processors that have been designed during this work can return a smaller AC product at these field sizes.

4.7 Conclusions

The computation of the characteristic 2 elliptic curve Tate pairing using the η_T methods has been discussed in this chapter. Calculation of the η_T pairing, followed by a suitable exponentiation, returns a Tate result in a very efficient manner.

Hardware architectures for $\mathbb{F}_{2^m}$ arithmetic have been described. Digit-serial multipliers are used to perform $\mathbb{F}_{2^m}$ multiplication since, through modification of their digit size, they enable a useful speed/area trade-off. Field arithmetic is ideally suited to hardware computation as many of the operations can be performed in parallel. Hardware architectures and design methods for the computation of $\mathbb{F}_{2^{4m}}$ arithmetic have also been presented. Many of the required operations can be performed in terms of parallel $\mathbb{F}_{2^m}$ operations. This provides further motivation for hardware implementation when large finite fields are in use. All efforts have been made to design arithmetic modules that return results quickly while endeavouring to minimise resource usage.

Dedicated hardware units for the computation of the major stages of the pairing algorithms have been presented. A precomputation unit stores all of the squares and square roots required by the η_T computation in $(m + 2)$ cycles. Another three units compute the operations required by the *for* loop of Algorithm 5. An exponentiation unit controls the operations required by Algorithm 6. Although implementing distinct stages of the pairing algorithms, the units have been created with their use and connectivity at the top level of the processor in mind. An efficient scheduling system has been created that allows simultaneous use of the units during the *for* loop of Algorithm 5. This strategy results in a very fast pairing computation.

An overview of the top level architecture of the pairing processor has also been provided. The processor utilises a relatively simple control scheme to schedule operations and to store intermediate variables. Results returned by the processor, when implemented on a Virtex-II FPGA with various multiplier digit sizes, have been presented. The processor returns its fastest Tate pairing result in 4,818 clock cycles. The results show that a hardware implementation of the elliptic curve characteristic 2 Tate pairing can return a result very quickly and in an efficient manner.

The topics and architectures discussed in this chapter have been published in [6], [7], and [8].

Chapter 5

A Processor for the Tate Pairing on a Genus 2 Hyperelliptic Curve

5.1 Introduction

Algorithms for Tate pairing computation on genus 2 curves are more complex than on elliptic cures as group addition, doubling and other curve arithmetic are generally more intricate. Supersingular genus 2 curves also have an embedding degree of $k = 12$, which is larger than the $k = 4$ and $k = 6$ characteristic 2 and characteristic 3 elliptic cases, respectively. The degree 12 extension field computations require many more base field arithmetic operations. These issues provide a barrier against the computation of the Tate pairing in the genus 2 case. However, Barreto et al. show that the Tate pairing can be computed efficiently on genus 2 curves using the η_T methods [63]. On another positive note, the security level of systems that rely on pairings are largely dependent on the product $k \times m$. This means that m can be 3 and 2 times smaller than in equivalent characteristic 2 and 3 elliptic curve systems. Implementations on genus 2 curves scale better as the value of m increases at a smaller rate as the security level is increased. This means that a system based on a genus 2 Tate pairing can provide an attractive alternative to more traditional elliptic curve based systems.

During this work hardware units, dedicated to the fast computation of the various operations required by the genus 2 Tate pairing, have been created. The more complex extension field computations are reduced to sub-field operations that are calculated in parallel. An efficient processor has been created that performs a fast genus 2 Tate pairing computation. This processor is more hybrid in nature than the characteristic 2 elliptic curve processor as the units share some operations.

Section 5.2 provides an introduction to Tate pairing computation, using the η_T method, on genus 2 curves. The hardware units that have been designed for the accelerated implementation of the main stages of pairing computation are discussed in Section 5.3. The genus 2 Tate pairing processor is then presented in Section 5.4, along with some implementation results.

5.2 The Genus 2 η_T and Tate Pairings

An introduction to Tate pairing computation using the η_T method in the genus 2 case is provided in this section. More detailed treatments of some of this subject matter are available in [62], [68] and [63].

Consider the supersingular genus 2 hyperelliptic curve $C(\mathbb{F}_q) : y^2 + y = x^5 + x^3 + d$, where $q = 2^m$, $d = 0$ or 1 and m is coprime to 6. This curve has embedding degree $k = 12$. Duursma and Lee [62] showed that degenerate divisors of the form $D = (P) - (\infty)$ can be used to compute the Tate pairing on curves of genus greater than 1. Katagi et al. show that this does not result in a loss of security [96]. The order of the Jacobian, $\#J_C(\mathbb{F}_q)$, is given by

$$2^{2m} + (-1)^d 2^{(3m+1)/2} + 2^m + (-1)^d 2^{(m+1)/2} + 1$$

if $m \equiv (1, 7, 17, 23) \mod 24$, or by

$$2^{2m} - (-1)^d 2^{(3m+1)/2} + 2^m - (-1)^d 2^{(m+1)/2} + 1$$

if $m \equiv (5, 11, 13, 19) \mod 24$.

The $\mathbb{F}_{2^{12m}}$ field can be built as a degree 12 extension of $\mathbb{F}_{2^m}$ using an irreducible polynomial of degree 12 with coefficients in $\mathbb{F}_2$. The extension field can also be constructed as a degree 2 extension of $\mathbb{F}_{2^{6m}}$. The $\mathbb{F}_{2^{6m}}$ field is first generated using an irreducible polynomial of degree 6. The $\mathbb{F}_{2^{6m}}$ field is then extended to $\mathbb{F}_{2^{12m}}$ using an irreducible polynomial of degree 2. The latter method allows arithmetic on $\mathbb{F}_{2^{12m}}$ to be expressed in terms of field operations on $\mathbb{F}_{2^{6m}}$. Operations on $\mathbb{F}_{2^{6m}}$ can then be written in terms of arithmetic on $\mathbb{F}_{2^m}$. This is advantageous from a hardware standpoint as some of the $\mathbb{F}_{2^{12m}}$ operations can be implemented using $\mathbb{F}_{2^{6m}}$ logic units that operate in parallel. The $\mathbb{F}_{2^{6m}}$ units can be implemented so that the required $\mathbb{F}_{2^m}$ arithmetic operations are also performed in parallel. This greatly simplifies the design process without reducing the attainable level of parallelism.

Extension fields are commonly generated by irreducible polynomials with coefficients in $\mathbb{F}_2$. There is, however, a well known distortion map on the curve that moves points from $C(\mathbb{F}_{2^m})$ to $C(\mathbb{F}_{2^{12m}})$. This mapping can be performed relatively trivially if the degree 2 extension of $\mathbb{F}_{2^{6m}}$ to $\mathbb{F}_{2^{12m}}$ is performed using an irreducible polynomial with coefficients in $\mathbb{F}_{2^{6m}}$.

The extension fields are constructed as follows. Let $\mathbb{F}_{2^m}$ be an m-degree extension of $\mathbb{F}_2$ generated by an irreducible polynomial f. Consider the field $\mathbb{F}_{2^6}$ generated by the irreducible polynomial

$$g = x^6 + x^5 + x^3 + x^2 + 1 \tag{5.1}$$

Let $w \in \mathbb{F}_{2^6}$ be a root of g. Members of $\mathbb{F}_{2^{6m}}$ can be represented using the polynomial basis:

$$\{1, w, w^2, w^3, w^4, w^5\} \tag{5.2}$$

For example, an element $a \in \mathbb{F}_{2^{6m}}$ can be written in polynomial form as

$$a = \sum_{i=0}^{5} a_i w^i = a_0 + a_1 w + a_2 w^2 + a_3 w^3 + a_4 w^4 + a_5 w^5 \tag{5.3}$$

where all $a_i \in \mathbb{F}_{2^m}$. Note that since $g = 0 \in \mathbb{F}_{2^6}$, then $w^6 = w^5 + w^3 + w^2 + 1$, $w^7 = w^5 + w^4 + w^2 + w + 1$, $w^8 = w + 1$ etc. These relationships can be used for the reduction of polynomials to $\mathbb{F}_{2^{6m}}$ when necessary.

Now, consider the degree 2 extension. Let h be a generator polynomial such that

$$h = y^2 + y + w^5 + w^3 \tag{5.4}$$

and let $s_0 \in \mathbb{F}_{12}$ be a root of h. Members of the field $\mathbb{F}_{2^{12m}}$ can be represented using the basis

$$\{1, w, w^2, w^3, w^4, w^5, s_0, ws_0, w^2 s_0, w^3 s_0, w^4 s_0, w^5 s_0\} \tag{5.5}$$

A member $b \in \mathbb{F}_{2^{12m}}$ can be written as

$$\begin{aligned} b =& b_0 + b_1 w + b_2 w^2 + b_3 w^3 + b_4 w^4 + b_5 w^5 + \\ & b_6 s_0 + b_7 w s_0 + b_8 w^2 s_0 + b_9 w^3 s_0 + b_{10} w^4 s_0 + b_{11} w^5 s_0 \end{aligned} \tag{5.6}$$

for all $b_i \in \mathbb{F}_{2^m}$, or as

$$b = u + s_0 v \tag{5.7}$$

where $u, v \in \mathbb{F}_{2^{6m}}$ such that $u = \sum_{i=0}^{5} u_i w^i$, $v = \sum_{i=0}^{5} v_i w^i$ and all $u_i, v_i \in \mathbb{F}_{2^m}$.

Let $s_1 = w^2 + w^4$ and $s_2 = w^4 + 1$. The distortion map moving a point $P = (x, y)$ from $C(\mathbb{F}_{2^m})$ to $C(\mathbb{F}_{2^{12m}})$ is given by

$$\psi(x, y) = (x + w, y + s_2 x^2 + s_1 x + s_0) \tag{5.8}$$

This is an inexpensive operation due to the manner in which $\mathbb{F}_{2^{12m}}$ was constructed. Substituting for s_1 and s_2 means that $\psi(x, y)$ can be computed according to

$$\begin{aligned} x &\to x + w \\ y &\to (y + x^2) + xw^2 + (x + x^2)w^4 + s_0 \end{aligned}$$

This mapping can also be expressed using the basis representation:

$$
\begin{aligned}
x \to & \{x, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\} \\
y \to & \{y + x^2, 0, x, 0, x + x^2, 0, 1, 0, 0, 0, 0, 0\}
\end{aligned}
$$

This requires one squaring and two additions on $\mathbb{F}_{2^m}$. Note also that the x-coordinate is a member of the sub-field $\mathbb{F}_{2^{6m}}$, which shows that the denominator elimination technique used to calculate the η_T pairing applies.

There is a very efficient octupling operation on the curve. Given a degenerate divisor $D = (P) - \infty$, a multiplication by 8 returns another degenerate divisor $[8]D = ([8]P) - (\infty)$, where

$$[8]P = (x^{2^6} + 1, y^{2^6} + x^{2^7} + 1) \tag{5.9}$$

This operation is inexpensive as it relies on squaring elements of $\mathbb{F}_{2^m}$, which is a relatively trivial operation. Let π be the 2-power Frobenius map and $\phi(x, y) = (x + 1, y + x^2 + 1)$. Equation (5.9) can be written as $[8]P = \phi\pi^6(P)$.

Barreto et al. [63] show how the Duursma and Lee computation techniques can be expressed in the generalised notation of Theorem 3.3.1. The value of T is given by $T = q = 2^{3m}$. This value means that the octupling formula can be used with ease. Note that pairing computation requires at most m iterations as octupling and not doubling forms the basis for the calculation of the Miller function. The value of N is given by $N = 2^{6m} + 1$. From Theorem 3.3.1, this means that the values of the other variables can be set as $c = 0$, $a = 2$ and $L = 1$. The exponent M is given by $M = (q^k - 1)/N = 2^{6m} - 1$. Let $D_1, D_2 \in J_C(\mathbb{F}_q)$. The relationship between the η_T and Tate pairings is given by

$$\left(\eta_T(D_1, D_2)^M\right)^{2q} = \langle D_1, \psi(D_2)\rangle_N^M \tag{5.10}$$

The value of T can be reduced. Let $q = 2^{3m}$ and $N = \#J_C(F_q) = 2^{2m} \pm 2^{(3m+1)/2} + 2^m \pm 2^{(m+1)/2} + 1$. Condition 1 of Theorem 3.3.1 holds if $T = \mp 2^{(3m+1)/2} - 1$. Computation of the pairing using this value of T requires approximately $m/2$ iterations using the octupling formula. The resultant value of c is $-(2^m \mp 2^{(m+1)/2} + 1)$. If $a = 2$ is selected, then $L = 2^{m+1} \mp$

$2^{(m+3)/2}+2$. The exponent M is given by $(2^{12m}-1)/(2^{2m}\pm 2^{(3m+1)/2}+2^m\pm 2^{(m+1)/2}+1)$. The relationship between the η_T and Tate pairings is now given by

$$\left(\eta_T(D_1,D_2)^M\right)^{2T} = \left(\langle D_1,\psi(D_2)\rangle_N^M\right)^L \tag{5.11}$$

5.2.1 Computation of the η_T Pairing

The η_T pairing is returned by $\eta_T(D_1,D_2) = f_{T,D_1}(\psi(D_2))$. If $T<0$ then T should be set to $-T$ and D_1 to $-D_1$. This means that $T = 2^{(3m+1)/2}\pm 1$. The Miller function is computed according to Equations (3.13) and (3.14). The fast octupling operation is used during the accumulation of this function. Since the multiplication of a degenerate divisor by 8 returns another degenerate divisor, computation of the intermediate functions can be expressed in terms of operations that are performed on the points in the support of the input divisors. Let $D_1 = (P)-(\infty)$ and $D_2 = (Q)-(\infty)$, where $P=(x_P,y_P)$ and $Q=(x_Q,y_Q)$. An intermediate function $f_{8,P}$ can be evaluated according to

$$f_{8,P}(x,y) = (y+b_4(x))^2(y+b_8(x)) \tag{5.12}$$

where

$$\begin{aligned} b_4(x) =& y_P^4 + (x_P^4)x + (x_P^8+x_P^4)x^2 + x^3 \\ b_8(x) =& (y_P^{16}+x_P^{16}+x_P^{48}+1) + (x_P^{32}+x_P^{16})x + (x_P^{32}+1)x^2 \end{aligned}$$

The value of T can be rewritten as $T = 2^{3(m-1)/2+2}\pm 1$ to accommodate the octupling operation. This means that the Miller function can be computed with $(m-1)/2$ octuplings, two point doublings and a final point addition or subtraction. The value of $\eta_T(P,Q)$ can now be computed according to

$$f_{T,P}(\psi(Q)) = \left(\prod_{i=0}^{(m-3)/2} f_{8,[8^i]P}(\psi(Q))^{2^{(3m-5)/2-3i}}\right).l_1(\psi(Q))^2.l_2(\psi(Q)).l_3(\psi(Q)) \tag{5.13}$$

where l_1 and l_2 are functions arising from final point doublings and l_3 corresponds to the final point addition or subtraction, as appropriate.

Barreto et al. described several efficient methods for the computation of Equation (5.13) in [63]. The $f_{8,[8^i]P}(\psi(Q))$ evaluations can be computed without the requirement that $[8^i]P$ be explicitly calculated. The distortion map can also be incorporated into the functions, which further simplifies the calculations. The exponentiations of the intermediate functions to $2^{(3m-5)/2-3i}$ can be performed using 2-power Frobenius operations on the coordinates of P and Q. Using these techniques, the relationship

$$\prod_{i=0}^{(m-3)/2} f_{8,[8^i]P}(\psi(Q))^{2^{(3m-5)/2-3i)}} = \prod_{i=0}^{(m-3)/2} \alpha.\beta \tag{5.14}$$

holds, where α and β are some members of $\mathbb{F}_{2^{12m}}$, defined as follows.

The efficient calculation of α and β requires that powers of squares of the input coordinates be precomputed. This can be achieved trivially using the 2-power Frobenius map π. The notation $x_P^{(i)} = x_P^{2^i}$ is used from this point. On initiation of pairing computation, the values of $x_P^{(i)}, x_Q^{(i)}, y_P^{(i)}, y_Q^{(i)}$ for all $0 \leq i \leq m-1$ are calculated and stored.

The value of $\alpha \in \mathbb{F}_{2^{12m}}$ is given by

$$\alpha = a + bw + cw^2 + dw^4 + s_0 \tag{5.15}$$

where

$$\begin{aligned}
a =\ & y_Q^{((3m-7-6i)/2)} + (x_P^{((3m-1+6i)/2)} + x_P^{((3m-3+6i)/2)}).x_Q^{((3m-5-6i)/2)} \\
& + (x_P^{((3m-3+6i)/2)} + 1 + x_Q^{((3m-5-6i)/2)}).x_Q^{((3m-7-6i)/2)} + y_P^{((3m-3+6i)/2)} + \gamma \\
b =\ & x_Q^{((3m-5-6i)/2)} + x_Q^{((3m-7-6i)/2)} \\
c =\ & x_Q^{((3m-5-6i)/2)} + x_P^{((3m-3+6i)/2)} + 1 \\
d =\ & x_P^{((3m-1+6i)/2)} + x_P^{((3m-3+6i)/2)}
\end{aligned}$$

The value of $\beta \in \mathbb{F}_{2^{12m}}$ is given by

$$\beta = e + f_2 w + gw^2 + hw^4 + s_0 \tag{5.16}$$

where

$$\begin{aligned}
e = \ & (x_P^{((3m+1+6i)/2)} + x_P^{((3m-1+6i)/2)}).x_Q^{((3m-9-6i)/2)} + (x_P^{((3m-1+6i)/2)} + x_Q^{((3m-7-6i)/2)}) \\
& .x_P^{((3m+1+6i)/2)} + y_P^{((3m-1+6i)/2)} + x_P^{((3m-1+6i)/2)} + y_Q^{((3m-9-6i)/2)} + \gamma \\
f_2 = \ & x_P^{((3m+1+6i)/2)} + x_P^{((3m-1+6i)/2)} \\
g = \ & x_P^{((3m+1+6i)/2)} + x_Q^{((3m-9-6i)/2)} + 1 \\
h = \ & x_Q^{((3m-7-6i)/2)} + x_Q^{((3m-9-6i)/2)}
\end{aligned}$$

Indices that are larger than m are reduced modulo m. The γ variable is set to 1 if $i \equiv 1 \mod 4$ and 0 otherwise.

Let $f'(\psi(Q)) = \prod_{i=0}^{(m-3)/2} f_{8,[8^i]P}(\psi(Q))^{2^{(3m-5)/2-3i}}$. The η_T pairing of Equation (5.13) is returned by multiplication of $f'(\psi(Q))$ by the evaluations of the l_1, l_2 and l_3 functions at $\psi(Q)$. This can be performed efficiently by analysing the divisors of the functions and the effect they have on the overall computation. Let D' be the divisor of f', where $D' = (P') - (\infty)$ and $P' = (x_{P'}, y_{P'})$. Let $l(x,y)$ be a function such that

$$l(x,y) = y + x^3 + (x_{P'}^8 + x_{P'}^4)x^2 + (x_{P'}^4)x + y_{P'}^4 \tag{5.17}$$

The value of $f'(\psi(Q)).l_1(\psi(Q))^2.l_2(\psi(Q)).l_3(\psi(Q))$ can be computed by squaring $f'(\psi(Q))$ twice and multiplying the result by $l(\psi(Q))$.

To summarise, the computation of the η_T pairing on the divisors $D_1 = (P) - (\infty)$ and $D_2 = (Q) - (\infty)$ can be performed according to

$$\eta_T(P,Q) = \left(\prod_{i=0}^{(m-3)/2} \alpha.\beta \right)^4 .l(\psi(Q)) \tag{5.18}$$

where α and β are calculated according to Equations (5.15) and (5.16), respectively, and $l(\psi(Q))$ is calculated according to Equation (5.17).

The operations that are required for the computation of $\eta_T(P,Q)$ using a field size of $m = 103$ are listed in Algorithm 10. This is the same field size and algorithm as that used

Algorithm 10 Computation of $\eta_T(P,Q)$ in the genus 2 case for $m = 103$

INPUT: $P = (x_P, y_P), Q = (x_Q, y_Q)$, where $P, Q \in J_C(F_{2^m})$

OUTPUT: $f = \eta_T(P,Q)$, where $f \in \mathbb{F}_{2^{12m}}$

Initialise: $f \leftarrow 1$, $\gamma \leftarrow 0$
for $(i \leftarrow 0,\ i \leq m-1,\ i \leftarrow i+1)$ **do**
  $x_1[i] \leftarrow x_P^{2^i},\ y_1[i] \leftarrow y_P^{2^i},\ x_2[i] \leftarrow x_Q^{2^i},\ y_2[i] \leftarrow y_Q^{2^i}$
end for
for $(i \leftarrow 0,\ i \leq (m-3)/2,\ i \leftarrow i+1)$ **do**
  ▷ *The following indices should be reduced modulo* m
  $k_1 \leftarrow (3m-9-6i)/2,\ k_2 \leftarrow (k_1+1),\ k_3 \leftarrow (k_2+1)$
  $k_4 \leftarrow (3m-3+6i)/2,\ k_5 \leftarrow (k_4+1),\ k_6 \leftarrow (k_5+1)$
  ▷ *Calculate* $\alpha \leftarrow a + bw + cw^2 + dw^4 + s_0$
  $b \leftarrow x_2[k_3] + x_2[k_2]$
  $c \leftarrow x_2[k_3] + x_1[k_4] + 1$
  $d \leftarrow x_1[k_4] + x_1[k_5]$
  $a \leftarrow d.x_2[k_3] + c.x_2[k_2] + y_2[k_2] + y_1[k_4] + \gamma$
  ▷ *Calculate* $\beta \leftarrow e + f_2 w + gw^2 + hw^4 + s_0$
  $f_2 \leftarrow x_1[k_5] + x_1[k_6]$
  $g \leftarrow x_2[k_1] + x_1[k_6] + 1$
  $h \leftarrow x_2[k_2] + x_2[k_1]$
  $e \leftarrow f_2.x_2[k_1] + (x_1[k_5] + x_2[k_2]).x_1[k_6] + y_2[k_1] + y_1[k_5] + x_1[k_5] + \gamma$
  $f_{current} \leftarrow \mathrm{cmul}(\alpha, \beta)$
  $f \leftarrow f.f_{current}$
end for
▷ *Perform the final operations*
$x_3 \leftarrow x_1[m-3] + 1$
$y_3 \leftarrow y_1[m-3] + x_1[m-2]$
$t \leftarrow (y_2[0] + x_2[1].(1 + x_2[0] + x_3^8 + x_3^4) + x_3^4.x_2[0] + y_3^4)$
$f \leftarrow f^4.(t + (x_2[1] + x_3^4)w + (x_3^8 + x_3^4)w^2 + w^3 + (x_2[1] + x_2[0])w^4 + s_0)$

RETURN: f

by Barreto et al. in [63]. The resultant extension field size is $km = 1,236$. Note that very minor changes are required to generalise the algorithm to any given field size.

The required powers of squares of the input coordinates are first computed using the 2-power Frobenius map. Once precomputation is complete, the main *for* loop of the algorithm begins to iterate. Six indices are used to access the precomputed values. The value of $\alpha \in \mathbb{F}_{2^{12m}}$ is first calculated. This requires six additions and two multiplications on $\mathbb{F}_{2^m}$ and two 1-bit additions. The value of $\beta \in \mathbb{F}_{2^{12m}}$ is also calculated, which requires eight additions and two multiplications on $\mathbb{F}_{2^m}$ and two 1-bit additions. Subsequently, $f_{current} \in \mathbb{F}_{2^{12m}}$ is computed by multiplying α by β. Due to the sparse nature of the operands, computing their product requires less computations than a regular $\mathbb{F}_{2^{12m}}$ multiplication. A customised routine, denoted *cmul*, has been created to perform this multiplication. Finally, the accumulating polynomial $f \in \mathbb{F}_{2^{12m}}$ is multiplied by $f_{current}$, which requires a regular multiplication on $\mathbb{F}_{2^{12m}}$. The final operations are performed after loop completion and require two 1-bit additions, eight $\mathbb{F}_{2^m}$ additions, two $\mathbb{F}_{2^m}$ multiplications and a multiplication on $\mathbb{F}_{2^{12m}}$.

5.2.2 Exponentiation to the Tate Pairing

The $\eta_T(P, Q)$ pairing result of Algorithm 10 must be exponentiated to a Tate pairing value. From Equation (5.11), the relationship between the pairings is given by $\left(\eta_T(D_1, D_2)^M\right)^{2T} = \left(\langle D_1, \psi(D_2)\rangle_N^M\right)^L$. This means that an exponentiation of the η_T pairing result to $\frac{2TM}{L}$ returns $\langle D_1, \psi(D_2)\rangle_N^M$. The exponentiation can be performed efficiently by factoring and cancelling terms when possible. The M term can be factored to give

$$M = (2^{6m} - 1)(2^m \mp 2^{(m+1)/2} + 1)(2^{3m} \mp 2^{(3m+1)/2} + 1) \tag{5.19}$$

The L term can be written as

$$L = 2(2^m \mp 2^{(m+1)/2} + 1) \tag{5.20}$$

After further cancellation, the exponent is given by

$$\frac{2TM}{L} = (2^{6m} - 1)(2^{3m} \mp 2^{4m} 2^{(m+1)/2} - 1) \tag{5.21}$$

The operations required to perform the conversion to the Tate pairing are detailed in Algorithm 11 for $m = 103$. The exponent is $(2^{6m} - 1)(2^{3m} - 2^{4m} 2^{(m+1)/2} - 1)$ in this case. Exponentiation to the first factor is computed on Lines 2-4. Powering to 2^{6m} is performed using a simple conjugation. This is followed by a multiplication and an inversion on $\mathbb{F}_{2^{12m}}$. The 2^{3m} and $-2^{4m} 2^{(m+1)/2}$ terms of the second factor are computed on Lines 5-9, requiring four applications of the Frobenius map and $(m+1)/2$ squarings. These terms are immediately multiplied by z_3, the first factor of the exponent. The (-1) term of the second factor is included last as the inverse of an element that has been powered to any multiple of $(2^{6m} - 1)$ is returned by the conjugate of that element. Exponentiation requires a total of four Frobenius actions, $(m+1)/2$ squarings, two conjugations, three multiplications and an inversion on $\mathbb{F}_{2^{12m}}$.

Algorithm 11 Exponentiation of $\eta_T(P, Q)$ to $\langle P, \psi(Q) \rangle_N^M$ in genus 2, $m = 103$ case

INPUT: $f = \eta_T(P, Q)$, where $f \in \mathbb{F}_{2^{12m}}$

OUTPUT: $z = \langle P, \psi(Q) \rangle_N^M$, where $z \in \mathbb{F}_{2^{12m}}$

1: $z_1 \leftarrow f$
2: $z_2 \leftarrow conj(z_1)$
3: $z_1 \leftarrow z_1^{-1}$
4: $z_3 \leftarrow z_2.z_1$
5: $z_4 \leftarrow (z_3)^{2^{3m}}$
6: $z_5 \leftarrow (z_4)^{2^m}$
7: **for** $(i \leftarrow 0,\ i < (m+1)/2,\ i \leftarrow i+1)$ **do**
8:     $z_5 \leftarrow z_5.z_5$
9: **end for**
10: $z_5 \leftarrow z_5.z_3$
11: $z_5 \leftarrow conj(z_5)$
12: $z \leftarrow z_4.z_5$

RETURN: z

The hardware implementation of all operations required to compute a Tate pairing using Algorithms 10 and 11 are described in the next section.

5.3 Hardware Units for Pairing Computation

Various strategies for fast Tate pairing computation were explored since no hardware implementations of any genus 2 pairings existed in the literature at the time of design. An embedding degree of $k = 12$ means that the extension field arithmetic operations require many computations on $\mathbb{F}_{2^m}$. A direct implementation may, therefore, result in a prohibitively large area footprint. The $\mathbb{F}_{2^{12m}}/\mathbb{F}_{2^{6m}}/\mathbb{F}_{2^m}$ tower of extensions is again advantageous as operations on $\mathbb{F}_{2^{12m}}$ can be expressed in terms of operations on $\mathbb{F}_{2^{6m}}$, thereby simplifying the design process. Arithmetic on $\mathbb{F}_{2^m}$ is implemented using the $\mathbb{F}_{2^m}$ hardware architectures previously discussed in Section 4.3.

The principal operations required for η_T pairing computation are:

1. The precomputation of the powers of the squares of the input coordinates before the *for* loop of Algorithm 10.
2. Calculation of the α and β terms on each iteration of the loop.
3. Computation of $f_{current} = cmul(\alpha, \beta)$, where *cmul* is a special routine that performs the multiplication of α by β in an efficient manner.
4. The $\mathbb{F}_{2^{12m}}$ multiplication of f by $f_{current}$.
5. The final operations that are required to return an η_T pairing value.
6. Exponentiation to the Tate pairing according to Algorithm 11.

The genus 2 Tate pairing processor contains a precomputation unit, a unit for the calculation of α and β, a dual mode multiplication unit that can perform either $cmul(\alpha, \beta)$ or $\mathbb{F}_{2^{12m}}$ multiplication, and an exponentiation unit. These units are discussed in this section. The interconnectivity of these units at the upper architectural levels of the processor will be discussed in Section 5.4.

5.3.1 The Precomputation Unit

The precomputation of $(x_1[i], y_1[i]) = ({x_P}^{2^i}, {y_P}^{2^i})$ and $(x_2[i], y_2[i]) = ({x_Q}^{2^i}, {y_Q}^{2^i})$ is required for all $0 \leq i \leq m-1$. Precomputation modules have been created that calculate and store the required powers of squares for each of the input coordinates. Each module contains $m \times m$ bit block RAM for storage, an $\mathbb{F}_{2^m}$ squarer and a control unit to handle reads and writes. The x_P module is illustrated in Figure 5.1. On initiation of the pairing computation, *write_en* is set, a counter within the control unit iterates from 0 to $m-1$ and the required powers of x_P are stored.

On completion of the precomputation stage, the main *for* loop of Algorithm 10 can proceed. On each iteration, powers of squares must be read according to the indices defined on Lines 7 and 8. These indices must be calculated *modulo m*. Hardware implementation of modular reduction can be avoided in this case as the indices rely solely on the field size m and the iterator i. The VHDL for the genus 2 processor is generated by *C++* software. This means that once the variable defining the field size has been defined, the indices that are required for all values of i can be generated in software and stored as constants in hardware. The indices are stored in arrays within the control unit. The control system uses the indices to generate the desired RAM address signals on each iteration and ensures that the correct powers are supplied.

The full precomputation unit is illustrated in Figure 5.2. It contains a precomputation module for each of the m-bit input coordinates. The calculations of α and β each require the availability of 2 $x_1[i]$ values, 2 $x_2[i]$ values, 1 $y_1[i]$ value and 1 $y_2[i]$ value on each

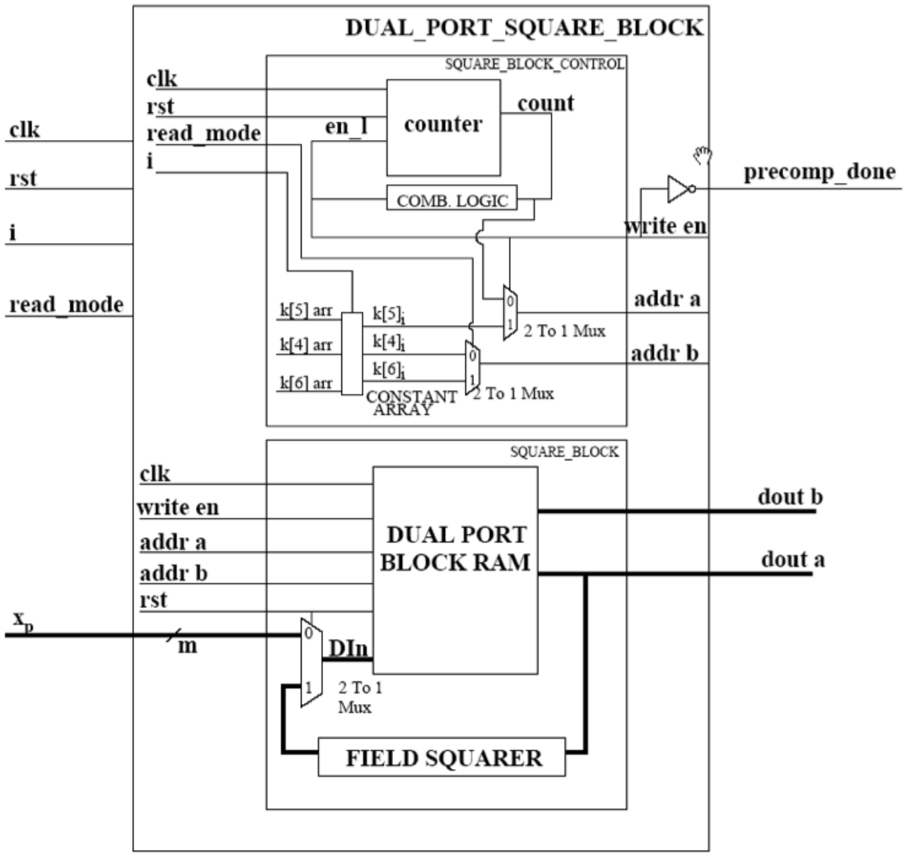

Figure 5.1: Precomputation Module for x_P

iteration. For this reason, the x_P and x_Q modules contain dual port RAM and the y_P and y_Q modules contain single port RAM. The precomputation unit has α and β modes, the selection of which is controlled by the *read_mode* input signal. In the first mode the m-bit powers required for α are provided while, in the second mode, the m-bit β powers are provided. The precomputation unit computes and stores all powers of the input coordinates in $(m+2)$ clock cycles and ensures the timely provision of powers during loop iteration.

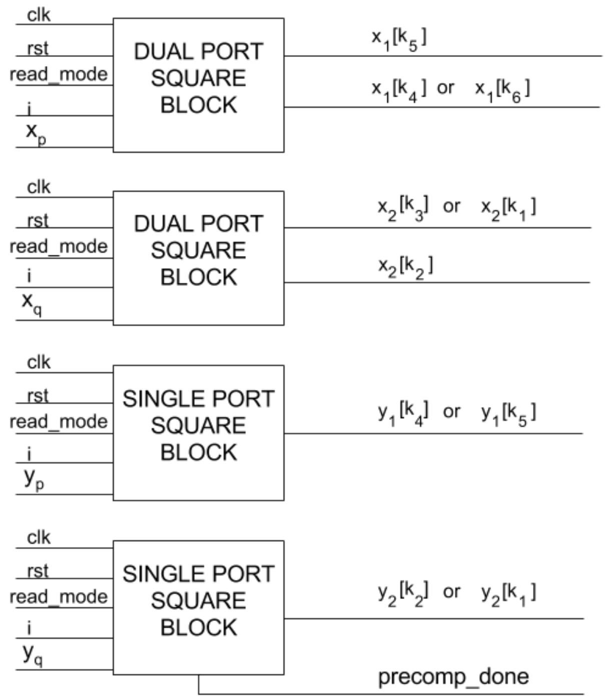

Figure 5.2: Precomputation Unit

5.3.2 Unit for the Calculation of α and β

The values of $\alpha, \beta \in \mathbb{F}_{2^{12m}}$ are calculated on Lines 9-18 of Algorithm 10. Similarities between the terms of these variables can be exploited so that resources can be shared. The unit that has been designed for the calculation of α and β is illustrated in Figure 5.3. It receives six m-bit inputs directly from the precomputation block and a 1-bit input corresponding to the value of γ. The input values are dependent on the *read_mode* control signal that is used to set the precomputation block to either α or β mode. The first $\mathbb{F}_{2^m}$ multiplier calculates $d.x_2[k_3]$ in α mode and $f_2.x_2[k_1]$ in β mode. Two multiplexers are used to select the inputs to the second multiplier as its inputs will be on different precomputation buses in the different modes. A *mode* signal, with the same value as *read_mode*, is used on the select line of these multiplexers. The second multiplier calculates $c.x_2[k_2]$ when in α mode and $(x_1[k_5]+x_2[k_2]).x_1[k_6]$ when in β mode, the results of which are added to a third term to produce either a or e, respectively. The b, c, d terms of α and f_2, g, h terms of β require additions alone and are collected from m-bit XOR gates. The unit contains a total of two 1-bit XOR gates, eight m-bit XOR gates and two m-bit digit serial multipliers of digit size D. An α or β result is returned in $(m/D+2)$ clock cycles.

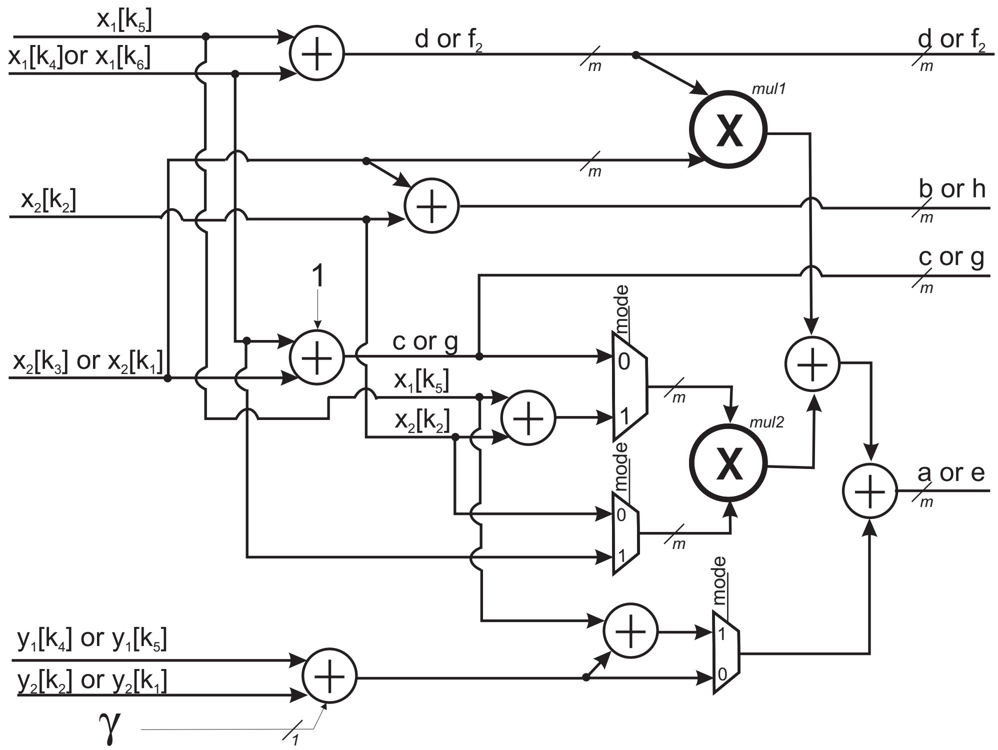

Figure 5.3: Unit for the calculation of α and β

5.3.3 Unit for Computation of $cmul(\alpha, \beta)$ and $\mathbb{F}_{2^{12m}}$ Multiplication

The values of $\alpha, \beta \in \mathbb{F}_{2^{12m}}$ must be multiplied together on Line 19 of Algorithm 10. This is performed by building a dedicated routine, $cmul(\alpha, \beta)$, that takes advantage of the sparse input polynomials. The resulting element, $f_{current}$, must then be multiplied by the accumulating polynomial f, which requires a multiplication on $\mathbb{F}_{2^{12m}}$.

As will be seen in this section, multiplication on $\mathbb{F}_{2^{12m}}$ requires three multiplications on $\mathbb{F}_{2^{6m}}$ and a number of other, less expensive, arithmetic operations. The computation of $cmul(\alpha, \beta)$ requires one $\mathbb{F}_{2^{6m}}$ multiplication and some other, trivial, operations. A dedicated extension field multiplication unit containing three $\mathbb{F}_{2^{6m}}$ multiplication modules would require a prohibitively large area, even with the use of $\mathbb{F}_{2^m}$ multipliers of low digit size. Due to the similarities between the arithmetic operations required by *cmul* and full $\mathbb{F}_{2^{12m}}$ multiplication, the design of a unit for the shared computation of both operations is attractive from an efficiency perspective. Such a unit has been designed, the main component of which is one $\mathbb{F}_{2^{6m}}$ multiplier. Other hardware modules are included in this unit so that both $cmul(\alpha, \beta)$ and $\mathbb{F}_{2^{12m}}$ multiplication can be computed using only one such sub-field multiplier.

In this section, multiplication on $\mathbb{F}_{2^{12m}}$ is first expressed in terms of arithmetic on $\mathbb{F}_{2^{6m}}$. The operations required by the *cmul* routine are also discussed. An architecture for fast $\mathbb{F}_{2^{6m}}$ multiplication is provided. A dual mode multiplication unit that can perform both $cmul(\alpha, \beta)$ and multiplication on $\mathbb{F}_{2^{12m}}$ efficiently is then presented.

Multiplication on $\mathbb{F}_{2^{12m}}$

An efficient technique for multiplication on fields of type $\mathbb{F}_{(q^n)^2}$ is described in [97]. Using this method, the composition and reduction stages of $\mathbb{F}_{2^{12m}}$ multiplication can be performed in one step. Recall from Section 5.2 that the irreducible polynomial defining $\mathbb{F}_{2^6}$ is $x^6+x^5+x^3+x^2+1$. Let $w \in \mathbb{F}_{2^6}$ be a root of this polynomial. The irreducible polynomial defining $\mathbb{F}_{2^{12}}$ is $y^2 + y + w^5 + w^3$. Let $s_0 \in \mathbb{F}_{12}$ be a root. Then ${s_0}^2 + s_0 = w^5 + w^3$. Let $a = u_1 + s_0 v_1$ and $b = u_2 + s_0 v_2$, where $a, b \in \mathbb{F}_{2^{12m}}$ and $u_1, u_2, v_1, v_2 \in \mathbb{F}_{2^{6m}}$. Then $c = a.b$, where $c \in \mathbb{F}_{2^{12m}}$, is given by

$$c = u_3 + s_0 v_3 \tag{5.22}$$

and

$$u_3 = u_1.u_2 + (w^5 + w^3)(v_1.v_2)$$
$$v_3 = (u_1 + v_1).(u_2 + v_2) + u_1.u_2$$

The multiplication of $(w^5 + w^3)$ by $(v_1.v_2)$ can be performed by creating a composition polynomial and reducing by g. This process does not require multiplication and can be achieved using $\mathbb{F}_{2^m}$ addition alone. This means that multiplication on $\mathbb{F}_{2^{12m}}$ requires three $\mathbb{F}_{2^{6m}}$ multiplications, four $\mathbb{F}_{2^{6m}}$ additions and 13 $\mathbb{F}_{2^m}$ additions that are required for the multiplication of $(w^5 + w^3)$ by $(v_1.v_2)$.

The Multiplication of α by β

The sparse nature of the α and β polynomials means that computing their product using a full $\mathbb{F}_{2^{12m}}$ multiplication would be inefficient. From Algorithm 10, α is given by

$$\alpha = a + bw + cw^2 + dw^4 + s_0 \tag{5.23}$$

where $a, b, c, d \in \mathbb{F}_{2^m}$. This can be rewritten as $\alpha = u_1 + s_0$, where $u_1 \in \mathbb{F}_{2^{6m}}$. Similarly, $\beta = u_2 + s_0$.

The routine $cmul(\alpha, \beta) = \alpha.\beta$ can now be defined. It is computed according to

$$\begin{aligned} cmul(\alpha, \beta) &= (u_1 + s_0).(u_2 + s_0) \\ &= u_1.u_2 + s_0(u_1 + u_2) + s_0^2 \end{aligned} \tag{5.24}$$

Recall that $s_0^2 + s_0 = (w^5 + w^3)$. Substituting for s_0^2 gives

$$\begin{aligned} cmul(\alpha, \beta) &= u_1.u_2 + s_0(u_1 + u_2) + s_0 + (w^5 + w^3) \\ &= (u_1.u_2 + w^5 + w^3) + s_0(u_1 + u_2 + 1) \end{aligned} \tag{5.25}$$

The calculation of the u component, therefore, requires one $\mathbb{F}_{2^{6m}}$ multiplication and two 1-bit additions. The calculation of the v component requires one $\mathbb{F}_{2^{6m}}$ addition and one 1-bit addition.

$\mathbb{F}_{2^{6m}}$ **Multiplication**

The most expensive operation required by both $cmul(\alpha, \beta)$ and by $\mathbb{F}_{2^{12m}}$ multiplication is multiplication on $\mathbb{F}_{2^{6m}}$. The pairing processor contains a dedicated module for fast $\mathbb{F}_{2^{6m}}$ multiplication for this reason. The Karatsuba algorithm [92] for polynomial multiplication can be used to perform multiplication on $\mathbb{F}_{2^{6m}}$. This algorithm is advantageous as it reduces the quantity of required $\mathbb{F}_{2^m}$ multiplications at the expense of more additions, which are less expensive.

Consider the multiplication $c = a.b$, where $a, b, c \in \mathbb{F}_{2^{6m}}$. Let $a = \sum_{i=0}^{i=5} a_i w^i$, $b = \sum_{i=0}^{i=5} b_i w^i$ and $c = a.b = \sum_{i=0}^{i=5} c_i w^i$, where all $a_i, b_i, c_i \in \mathbb{F}_{2^m}$. The calculation of c begins with a composition stage, in which a degree 10 polynomial, c', is computed. The a and b polynomials are first separated into lower and upper half polynomials of degree 2:

$$\begin{aligned} a &= (a_0 + a_1 w + a_2 w^2) + w^3(a_3 + a_4 w + a_5 w^2) \\ &= A_l + w^3 A_h \end{aligned} \tag{5.26}$$

$$\begin{aligned} b &= (b_0 + b_1 w + b_2 w^2) + w^3(b_3 + b_4 w + b_5 w^2) \\ &= B_l + w^3 B_h \end{aligned} \tag{5.27}$$

Three polynomials are then computed according to

$$\begin{aligned} P &= A_l.B_l \\ Q &= (A_l + A_h).(B_l + B_h) \\ R &= A_h.B_h \end{aligned} \tag{5.28}$$

where $P = \sum_{i=0}^{4} p_i w^i, Q = \sum_{i=0}^{4} q_i w^i, R = \sum_{i=0}^{4} r_i w^i$ and all $p_i, q_i, r_i \in \mathbb{F}_{2^m}$.

These degree 2 multiplications can be performed by again separating the inputs into lower and upper halves. Let $a' = \sum_{i=0}^{2} a'_i w^i$, $b' = \sum_{i=0}^{2} b'_i w^i$, and $d' = \sum_{i=0}^{4} d'_i w^i$ where all $a'_i, b'_i, d'_i \in \mathbb{F}_{2^m}$. The degree 4 polynomial $d' = a'.b'$ can be computed by first performing the following six $\mathbb{F}_{2^m}$ multiplications:

$$\begin{array}{lll} mul_0 = a'_0.b'_0 & mul_1 = a'_1.b'_1 & mul_2 = a'_2.b'_2 \\ mul_3 = (a'_0 + a'_1)(b'_0 + b'_1) & mul_4 = (a'_1 + a'_2)(b'_1 + b'_2) & mul_5 = (a'_0 + a'_2)(b'_0 + b'_2) \end{array}$$

The coefficients of the degree 4 polynomial d' is given in vector form by

$$\begin{bmatrix} d'_0 \\ d'_1 \\ d'_2 \\ d'_3 \\ d'_4 \end{bmatrix} = \begin{bmatrix} mul_0 \\ mul_3 + mul_1 + mul_0 \\ mul_1 + mul_5 + mul_0 + mul_2 \\ mul_4 + mul_1 + mul_2 \\ mul_2 \end{bmatrix} \tag{5.29}$$

The computation of d', therefore, requires a total of six $\mathbb{F}_{2^m}$ multiplications and 13 $\mathbb{F}_{2^m}$ additions.

Once calculated, the P, Q and R polynomials of Equation (5.28) can be used to evaluate c', the degree 10 composition polynomial of the original $\mathbb{F}_{2^{6m}}$ multiplication. This polynomial is given by

$$c' = \sum_{i=0}^{4} p_i w^i + \sum_{i=3}^{7} (p_{i-3} + q_{i-3} + r_{i-3}) w^i + \sum_{i=6}^{10} r_{i-6} w^i \tag{5.30}$$

This composition polynomial must be reduced modulo the irreducible polynomial $x^6 + x^5 + x^3 + x^2 + 1$. The final, reduced, result $c = a.b$ is returned according to

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} c'_0 + c'_6 + c'_7 + c'_8 \\ c'_1 + c'_7 + c'_8 + c'_9 \\ c'_2 + c'_6 + c'_7 + c'_9 + c'_{10} \\ c'_3 + c'_6 + c'_{10} \\ c'_4 + c'_7 \\ c'_5 + c'_6 + c'_7 \end{bmatrix} \tag{5.31}$$

The degree 2 polynomial multiplication required during the calculation of the P, Q and R polynomials of Equation (5.28) is the most expensive aspect of $\mathbb{F}_{2^{6m}}$ multiplication. A hardware architecture has been created for accelerated degree 2 multiplication and is illustrated in Figure 5.4. The module contains six digit-serial $\mathbb{F}_{2^m}$ multipliers and 13 $\mathbb{F}_{2^m}$ adders. The multipliers operate in parallel. A degree 2 multiplication result is returned in $(m/D + 2)$ clock cycles, where D is the digit size of the multipliers.

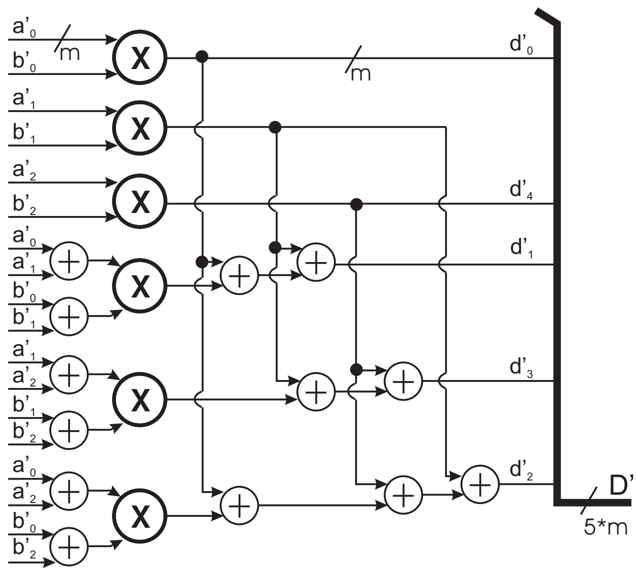

Figure 5.4: Module for degree 2 polynomial multiplication using the Karatsuba method

The hardware unit dedicated to the fast implementation of $\mathbb{F}_{2^{6m}}$ multiplication is illustrated in Figure 5.5. The unit receives a pair of $6m$-bit signals A and B and returns $C = A.B$ on a $6m$-bit output bus. The architecture contains three degree 2 multiplication modules so that P, Q and R can be computed in parallel. The $\mathbb{F}_{2^m}$ additions that are required at the end of the composition stage and during reduction are combined, since some cancellations occur and nesting is possible. The unit returns an $\mathbb{F}_{2^{6m}}$ multiplication result in $(m/D + 4)$ clock cycles.

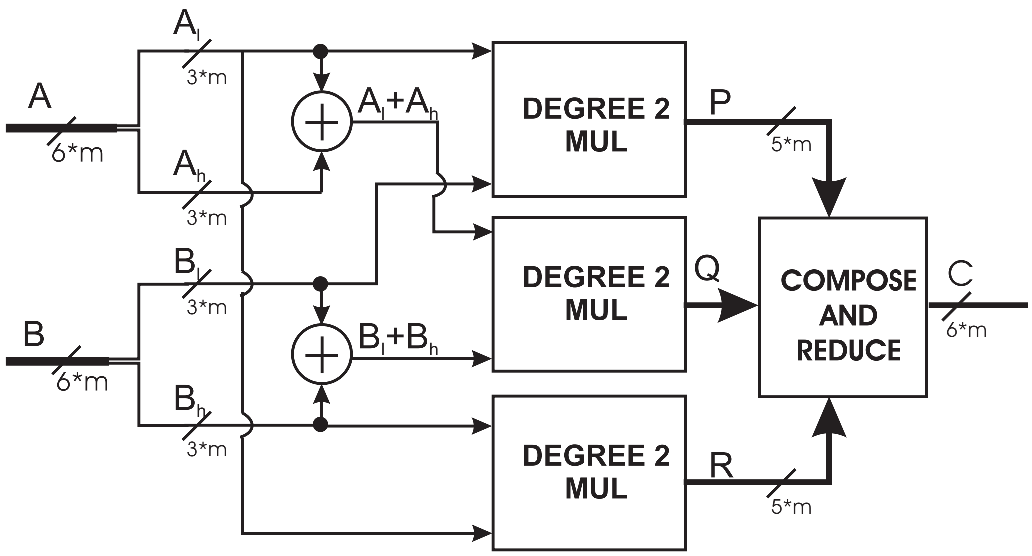

Figure 5.5: The $\mathbb{F}_{2^{6m}}$ Multiplication Unit

Dual Mode Multiplication Architecture

A hardware architecture has been designed that implements both the *cmul* routine and $\mathbb{F}_{2^{12m}}$ multiplication. This is known as the *dual mode multiplication unit* and is shown in Figure 5.6. The unit takes as input two $12m$-bit operands and separates each input into $6m$-bit u and v components. The result of a *cmul* routine or an $\mathbb{F}_{2^{12m}}$ multiplication appears on the $12m$-bit *res* output bus on completion.

The most costly arithmetic operation is $\mathbb{F}_{2^{6m}}$ multiplication. The *cmul* computation requires one $\mathbb{F}_{2^{6m}}$ multiplication, while multiplication on $\mathbb{F}_{2^{12m}}$ requires three. The $\mathbb{F}_{2^{6m}}$ multiplier has a relatively large area footprint due to the quantity of parallel $\mathbb{F}_{2^m}$ multipliers required to ensure a high speed computation. The dual mode multiplier contains only one $\mathbb{F}_{2^{6m}}$ multiplier for this reason.

In the first mode of operation, $cmul(\alpha, \beta)$ is computed according to Equation (5.25). The $6m$-bit u components of α and β appear at the u_1 and u_2 unit inputs. The $sel1$ line is set to 0 and $u_1.u_2$ is computed by the $\mathbb{F}_{2^{6m}}$ multiplier. A 1 is added to the w^5 and w^3

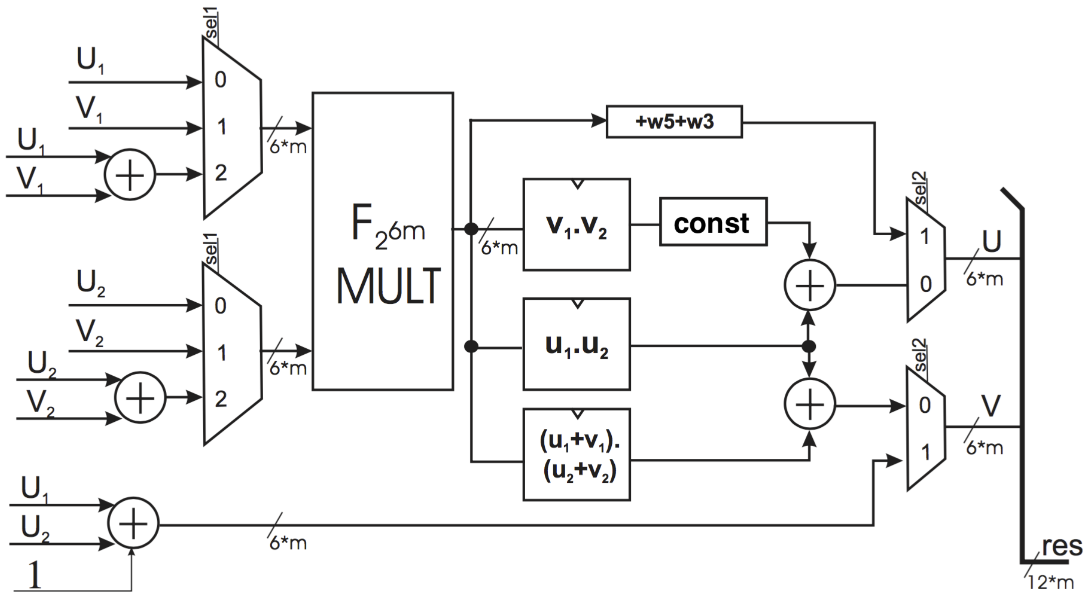

Figure 5.6: The Dual Mode Multiplier

components of the result before it reaches the multiplexer controlling the bus sent to the u component of the result. The v component of $cmul(\alpha, \beta)$ is given by $(u_1 + u_2 + 1)$, which can be implemented using two m-bit adders and one 1-bit adder. The $sel2$ line is set to 1 so that the correct outputs are collected on the u and v output buses.

In the second mode of operation, an $\mathbb{F}_{2^{12m}}$ multiplication is performed according to Equation (5.22). Let $a = u_1 + v_1 s_0$, $b = u_2 + v_2 s_0$ and $c = a.b = u_3 + v_3 s_0$, where $a, b, c \in \mathbb{F}_{2^{12m}}$. The values of $u_1, u_2, v_1, v_2 \in \mathbb{F}_{2^{6m}}$ appear at the input buses before the $\mathbb{F}_{2^{12m}}$ multiplication $c = a.b$ is initiated. The $sel1$ line is first set to 0 and the $\mathbb{F}_{2^{6m}}$ multiplication $u_1.u_2$ performed. Once available, the result is stored in a $6m$-bit register at the output of the $\mathbb{F}_{2^{6m}}$ multiplier. The $sel1$ line is then set to 1 and $v_1.v_2$ is computed and stored. Finally, the $sel1$ line is set to 2 and $(u_1 + v_1).(u_2 + v_2)$ is computed and stored. The value of u_3 is given by $u_1.u_2 + (w^5 + w^3)(v_1.v_2)$. The constant multiplication of $(w^5 + w^3)$ by $(v_1.v_2)$ is implemented combinatorially using XOR gates contained within the $const$ module. The result is added to $(u_1.u_2)$ using a $6m$-bit adder. The value of v_3 is given by $(u_1 + v_1).(u_2 + v_2) + (u_1.u_2)$, which is implemented using a $6m$-bit adder. The $sel2$ line is set to 0 so that the correct u and v values are collected on the res output bus.

The dual mode multiplication unit returns a result very quickly while ensuring that resources are used efficiently. The $cmul(\alpha, \beta)$ routine is performed in $(m/D+8)$ clock cycles, where D is the digit size of the multipliers. An $\mathbb{F}_{2^{12m}}$ multiplication result can be returned in $(3m/D + 20)$ clock cycles.

5.3.4 The Exponentiation Unit

Exponentiation of the η_T result to a Tate pairing value is performed according to Algorithm 11. As discussed in Subsection 5.2.2, exponentiation requires four Frobenius actions, $(m + 1)/2$ squarings, two conjugations, three multiplications and one inversion, all of which are performed on $\mathbb{F}_{2^{12m}}$. Multiplication on $\mathbb{F}_{2^{12m}}$ has already been discussed. The other operations can be expressed in terms of arithmetic on $\mathbb{F}_{2^{6m}}$. These $\mathbb{F}_{2^{6m}}$ arithmetic operations are then implemented in the same way as in the $\mathbb{F}_{2^{4m}}$ case of the previous chapter.

Let $a = u + s_0 v$, where $a \in \mathbb{F}_{2^{12m}}$ and $u, v \in \mathbb{F}_{2^{6m}}$. The conjugate, $\tilde{a}$, of a is given by

$$\tilde{a} = (u + v) + s_0 v \tag{5.32}$$

The value of a^2 is computed according to

$$a^2 = u^2 + (w^5 + w^3)v^2 + s_0 v^2 \tag{5.33}$$

The Frobenius computation depends on the value of m modulo 12. The next section contains results returned by a pairing processor when implemented using a field size of $m = 103$, which means that $m \mod 12 \equiv 7$. The Frobenius map in this case is

$$a^q = u^q + (1 + w^3 + w^5)v^q + s_0 v^q \tag{5.34}$$

Inversion on $\mathbb{F}_{2^{12m}}$ can be performed with only one $\mathbb{F}_{2^{6m}}$ inversion and a number of less expensive operations. The computation

$$e = \Big(u.(u+v) + (w^5 + w^3)v^2\Big)^{-1} \tag{5.35}$$

is first performed, where $e \in \mathbb{F}_{2^{6m}}$. The inverse of a is then given by

$$a^{-1} = (u+v).e + s_0(v.e) \tag{5.36}$$

The inversion on $\mathbb{F}_{2^{12m}}$, therefore, requires one $\mathbb{F}_{2^{6m}}$ inversion, three $\mathbb{F}_{2^{6m}}$ multiplications, one $\mathbb{F}_{2^{6m}}$ addition and the 13 $\mathbb{F}_{2^m}$ additions required for the constant multiplication of v^2 by $(w^5 + w^3)$. The $\mathbb{F}_{2^{6m}}$ inversion is computed using the Extended Euclidean Algorithm (previously discussed in Subsection 4.4.5) and requires one $\mathbb{F}_{2^m}$ inversion and several multiplications and additions on $\mathbb{F}_{2^m}$.

To ensure a fast exponentiation, the pairing processor contains a dedicated exponentiation unit. This unit contains modules implementing conjugation, squaring and the Frobenius map. These operations require combinatorial logic alone. The unit also contains an $\mathbb{F}_{2^m}$ inverter. The required extension field multiplications are performed by the dual mode multiplier. The sequence of operations is controlled by a system at the top level of the pairing processor.

5.4 The Characteristic 2 Genus 2 Tate Pairing Processor

The hardware processor performing a fast genus 2 Tate pairing computation is presented in Figure 5.7. The processor receives four m-bit input buses on which the x_P, y_P, x_Q, y_Q input coordinates are loaded. The Tate pairing $\langle P, \psi(Q)\rangle_N^M$ appears on the $12m$-bit *result* output bus on computation completion.

The processor contains the precomputation unit, the unit for the calculation of α and β, the dual mode multiplication unit and the exponentiation unit. Recall that, in the elliptic

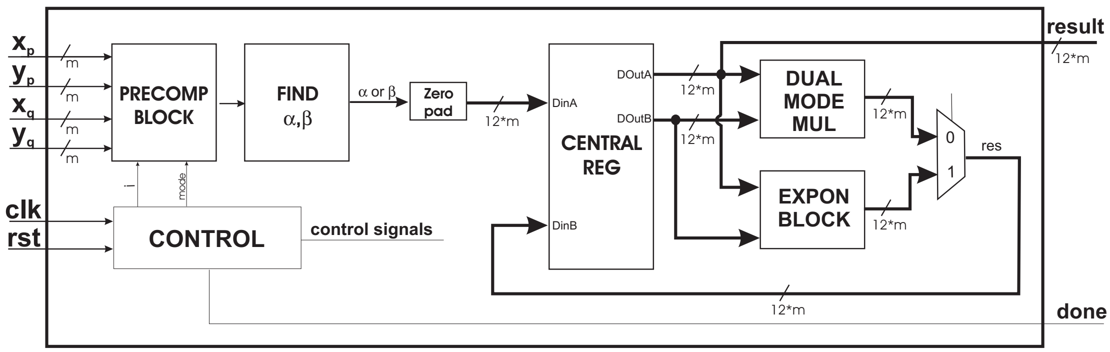

Figure 5.7: The Genus 2 Tate Pairing Processor

curve processor described in the previous chapter, all variables were stored in registers at the inputs and outputs of the computation units. The genus 2 processor employs a different storage scheme. In this case, all computation stages of the main for loop in Algorithm 10 cannot be performed in parallel. The dual mode multiplication unit must perform both the *cmul* operation and subsequent $\mathbb{F}_{2^{12m}}$ multiplication of the result by the accumulating function f. A central storage system provides the best solution in this case. Dual port $12m$-bit RAM is used instead of registers. The values of α and β, padded to $12m$ bits, are stored using the $DInA$ RAM input. The RAM outputs are each connected to both the dual mode multiplication unit and the exponentiation unit. A multiplexer is used to select the result to be sent to the $DInB$ RAM input on the $12m$-bit res bus. Exponentiation can be performed with ease by performing the required $\mathbb{F}_{2^m}$ inversion and combinatorial operations in the exponentiation unit, storing the intermediate variables in RAM and accessing the dual mode multiplication unit when required. This system also provides a straightforward solution to the problem of routing large buses at the top level of the processor.

The control scheme is relatively uncomplicated. The system contains a finite state machine that outputs all necessary control signals while stepping through the required operations. These signals are used to handle reading from RAM, to manage the reset and enable sequences of the arithmetic units and to store results after the correct number of clock cycles.

On processor reset, all required powers of squares of the input coordinates are immediately computed and stored in the precomputation block. The controller then initiates the operations that are required during the for loop of Algorithm 10. Note that the computation of α and β does not depend on the results from any previous iteration of the loop as, after precomputation, all input operands are available. On the first iteration the values of α and β are calculated and stored. The value of $f_{current} = cmul(\alpha, \beta)$ is then computed. After this, the $\mathbb{F}_{2^{12m}}$ multiplication $f.f_{current}$ begins. The values of α and β that will be required on the second iteration are computed in parallel with this extension field multiplication. This means that the *cmul* operation of the second iteration can begin immediately after the $\mathbb{F}_{2^{12m}}$ multiplication of the first iteration. This system provides a saving of $2(m/D + 2)$ clock cycles on each loop iteration after the first. Recall from Subsection 5.3.3 that the *cmul* operation can be performed in $(m/D + 8)$ clock cycles, while $\mathbb{F}_{2^{12m}}$ multiplication requires $(3m/D + 20)$ clock cycles. This means that each iteration is completed in $(4m/D + 28 + \theta)$ clock cycles, where θ is a relatively small number of cycles required by the control system and for reads from and writes to RAM.

On completion of the *for* loop, the final operations of Algorithm 10 must be performed. These computations are implemented using both the dual mode multiplier and the combinatorial logic within the exponentiation unit. Exponentiation of the η_T value to a Tate result is then performed according to Algorithm 11. On completion, the Tate pairing value $\langle P, \psi(Q)\rangle_N^M$ is available on the $12m$-bit *result* bus at the output of the processor.

5.4.1 Results and Comparisons

The architectures described in this chapter were defined at the VHDL RTL level. The code was automatically generated by the *C++* design system. The processor was implemented on the Xilinx Virtex-II Pro FPGA (xc2vp100-6ff1696) containing 44,096 slices. This is the same FPGA on which the characteristic 2 elliptic curve processor was implemented in the previous chapter. The results in this subsection are for an implementation on the field $\mathbb{F}_{2^{103}}$, which has irreducible polynomial $x^{103}+x^9+1$. The extension field is $12\times 103 = 1236$

bits in size.

The results returned by processor implementations with $\mathbb{F}_{2^m}$ multipliers of digit size 4, 8, 12 and 16 are presented in Table 5.1. A full Tate pairing $\langle P, \psi(Q)\rangle_N^M$ is performed in each case.

Table 5.1: Results returned by the genus 2 Tate pairing processor for $m = 103$

D (Bits)	**Area** (Slices)	**Utilisation**	**Clock Cycles**	**AC Product** (Msl.cycles)
4	21021	48%	10506	221
8	24290	55%	6992	170
12	27182	62%	5805	157
16	30464	69%	5412	165

The $D = 16$ case returns a Tate pairing in 5,412 clock cycles. This implementation requires 30,464 slices and has an AC product of 165 Mslice.cycles. The $D = 12$ implementation has a lower AC product as the extra 3,282 slices required in the $D = 16$ case results in a saving of only 393 clock cycles. As the number of cycles required for multiplication decreases, the impact of the relatively constant number of cycles required for combinatorial logic, storage and control becomes more significant. Increasing the digit size beyond 16 further reduces efficiency. It can also result in a reduction in clock frequency. The $D = 12$ case is the most attractive option, requiring 27,182 slices and returning a Tate pairing result in only 5,805 cycles. As in the processor of the previous chapter, it would be interesting to investigate slice usage on larger FPGAs due to the relatively high occupancy of these implementations.

Barreto et al. provide the results of a software computation of the genus 2 η_T pairing in [63]. An η_T pairing is returned in 1.8 ms when computed on a Pentium IV processor operating at 3 GHz. Note that exponentiation to the Tate pairing is not performed. The custom hardware processor provides a significant acceleration over this result, even for low frequencies.

At the time, there had been no published hardware implementations of any genus 2 pairing. The results can, however, be compared fairly against those returned by the characteristic 2 elliptic curve processor of the previous chapter. That processor returned its fastest pairing in 4,818 clock cycles with a corresponding AC product of 212 Mslice.cycles, the lowest of those implementations. The $D = 12$ genus 2 processor has a significantly smaller AC product of 157 Mslice.cycles. Furthermore, the genus 2 $D = 12$ case outperforms all but one of the elliptic curve processors. It can, for example, return a pairing in 5,805 cycles using 27,182 slices. Conversely, the $D = 8$ elliptic implementation requires a similar 6,200 clock cycles but it is larger by 13,896 slices. The genus 2 implementations are the more efficient in all cases.

5.5 Conclusions

Computation of the genus 2 Tate pairing using the η_T method has been discussed in this chapter. The required operations have been separated into distinct stages. The extension field operations that underpin these stages have been expressed in a manner that makes them suited to parallel hardware implementation. Custom hardware units that implement the pairing computation stages rapidly and efficiently have been presented. A genus 2 Tate pairing processor comprising these units has been implemented on an FPGA and results presented.

The more complex pairing computation and higher embedding degree previously provided a barrier to Tate pairing implementation on genus 2 hyperelliptic curves. This work shows that genus 2 curves offer a viable alternative to elliptic curves for Tate pairing implementation on hardware platforms. Furthermore, if the security level of a system based on the Tate pairing must be increased, the field size will grow at a lower rate in the genus 2 case. This means that a hardware processor accelerating the genus 2 Tate pairing may become increasingly attractive.

The topics and architectures discussed in this chapter formed the basis for papers that were published in [9] and [10].

Chapter 6

A Design System for Pairing Computation Using Flexible Processors

6.1 Introduction

Flexibility is an important aspect of hardware processors that are used in a cryptographic context. If the security level of an application must be changed, the size of the finite fields on which the algorithms are implemented must also be changed. It should also be kept in mind that cryptographic systems must operate in a broad range of environments. A large server side system may, for example, have to communicate with client side devices that are much more constrained in terms of area. Speed of pairing computation may be the most desirable property in the former environment, while a very small footprint may be required in the latter. The characteristic 2 elliptic and genus 2 processors of the previous chapters return pairing results in a very low number of clock cycles. Their efficient use may, however, require area resources that are not available in some systems.

The computation and hardware implementation of the Tate pairing on characteristic 3 elliptic curves has not been discussed until this point in the thesis. A supersingular characteristic 3 elliptic curve has an embedding degree of 6, larger than the characteristic 2 case of 4. It is generally more difficult to perform attacks on characteristic 3 pairings. These factors mean that a smaller field size can be used in the characteristic 3 elliptic case. Hardware is, however, a binary environment and characteristic 3 storage and arithmetic operations have an area overhead. As an example, Kerins et al. [98] describe a characteristic 3 architecture that implements the Tate pairing using the Duursma Lee methods on $\mathbb{F}_{3^{97}}$. Their architecture contains several units for computation on $\mathbb{F}_{3^{6m}}$. They return a Tate result in 12,750 clock cycles but estimate a large area footprint of 55,616 slices. A different approach was taken to the hardware implementation of characteristic 3 pairings in this work to reduce the required resources.

A flexible architecture for Tate pairing computation has been created. Its architectural parameters can be varied in a manner that supports both characteristic 2 and 3 Tate pairing implementation. The aim is to provide implementation solutions for a large range of environments. The architecture does not contain any units that are dedicated to extension field arithmetic operations: it contains a variable quantity of subfield modules alone. A flexible *C++* software system has also been created for the efficient generation, analysis and implementation of these processors. It contains a class for the automatic generation of instruction sequences according to high level algorithmic descriptions. The system significantly reduces the level of intervention required by a user when designing the architectures discussed in this chapter.

A processor with some similar features was presented by Byrne et al. in [99]. Their architecture can be selected to contain modules performing arithmetic on $\mathbb{F}_{2^m}$, on $\mathbb{F}_{p^m}$ for $p \geq 2$, and on $\mathbb{F}_p$, where p is a large prime. However, in each case the quantity of modules cannot be varied. Implementation results are returned for elliptic curve point scalar multiplications: pairings are not performed. Although the VHDL for the processor is generated in software, the instruction sequences must be defined manually. The large variety of subfield and extension field operations required for pairing computation means that the manual creation of instruction sequences would be a significant challenge. It should be noted that the author of this thesis provided some assistance during the initial stages of the creation of this system. Subsequently, Byrne et al. discussed side channel attacks (described in Subsection 7.2.2) on systems implementing elliptic curve scalar multiplication and pairings [100], [101].

Characteristic 3 elliptic curve Tate pairing computation using the η_T method has not yet been introduced and is outlined in Section 6.2. The hardware implementation of arithmetic on $\mathbb{F}_3$ and $\mathbb{F}_{3^m}$ is discussed in Section 6.3. Arithmetic on $\mathbb{F}_{3^{6m}}$ is also discussed with regards to its computation using $\mathbb{F}_{3^m}$ modules alone. The pairing processor is described in Section 6.4. The design system that has been created to aid in the creation and prototyping of the processor is also outlined. Characteristic 2 and 3 pairing results returned by the processor when implemented using various architectural parameters are also presented.

6.2 The Characteristic 3 Elliptic Curve η_T and Tate Pairings

Duursma and Lee provide efficient techniques for the computation of pairings for $p \geq 3$ on a small subset of curves in [62]. Barreto et al. generalise these methods in [63] and show that the size of the computational loop can be halved on supersingular curves of characteristic 3 using the η_T method. This section provides an overview of Tate pairing calculation on characteristic 3 elliptic curves using these techniques.

Let $E(\mathbb{F}_q) : y^2 = x^3 - x + b$ be an elliptic curve defined on $\mathbb{F}_q = \mathbb{F}_{3^m}$, where $b = \pm 1$ and m is not divisible by 6. This curve has embedding degree $k = 6$. The order of the curve is $\#E(\mathbb{F}_{3^m}) = 3^m + 1 + b3^{(m+1)/2}$ if $m \equiv (1, 11) \mod 12$ or $\#E(\mathbb{F}_{3^m}) = 3^m + 1 - b3^{(m+1)/2}$ if $m \equiv (5, 7) \mod 12$. The η_T and Tate pairings of this section are calculated on the points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, where $P, Q \in E(\mathbb{F}_q)$.

A distortion map that allows denominator elimination is available. This map is $\psi(x, y) = (\rho - x, \sigma y)$ where $\rho, \sigma \in \mathbb{F}_{3^{6m}}$ such that $\sigma^2 = -1$ and $\rho^3 = \rho + b$. The relationships $\rho^{3^2} = \rho + 2b$ and $\rho^{3^3} = \rho$ are useful for pairing computation. A fast tripling formula exists on the curve. Given a point $P = (x, y)$, then $[3]P = (x_3, y_3)$ where $x_3 = (x^3)^3 - b$ and $y_3 = -(y^3)^3$. Let π be the 3-power Frobenius on $\mathbb{F}_q$ and let ϕ be a function such that $\phi(x, y) = (x - b, -y)$. Then

$$[3]P = \phi\pi^2(x, y) \tag{6.1}$$

Multiplication of a point by $q = 3^m$ is given by $[q]P = [3^m]P = \phi^m\pi^{2m}(x, y) = \phi^m(x, y)$ since $\pi^{2m}(x, y) = (x, y)$. The automorphism on the curve is, therefore, given by $\gamma = \phi^m$.

The $\mathbb{F}_{3^{6m}}$ field is constructed as a degree 3 extension of $\mathbb{F}_{3^{2m}}$. The $\mathbb{F}_{3^{2m}}$ field is generated by an irreducible polynomial g such that

$$\mathbb{F}_{3^{2m}} \equiv \mathbb{F}_{3^m}[y]/g(y) \text{ where } g(y) = y^2 + 1 \tag{6.2}$$

The $\mathbb{F}_{3^{6m}}$ extension is then given by

$$\mathbb{F}_{3^{6m}} \equiv \mathbb{F}_{3^{2m}}[z]/h(z) \text{ where } h(z) = z^3 - z - b \tag{6.3}$$

The use of this extension and these particular irreducible polynomials means that the distortion map is simplified and that the complexity of arithmetic on the extension field is minimised.

Members of $\mathbb{F}_{3^{6m}}$ can be represented using the basis $(1, \sigma, \rho, \sigma\rho, \rho^2, \sigma\rho^2)$. An element $a \in \mathbb{F}_{3^{6m}}$ can be written as

$$a = a_0 + a_1\sigma + a_2\rho + a_3\sigma\rho + a_4\rho^2 + a_5\sigma\rho^2 \tag{6.4}$$

where $a_0, a_1, a_2, a_3, a_4, a_5 \in \mathbb{F}_{3^m}$. A 3-tuple representation can also be used:

$$a = \hat{a}_0 + \hat{a}_1\rho + \hat{a}_2\rho^2 \tag{6.5}$$

where $\hat{a}_0 = a_0 + a_1\sigma$, $\hat{a}_1 = a_2 + a_3\sigma$, $\hat{a}_2 = a_4 + a_5\sigma$ and $\hat{a}_1, \hat{a}_1, \hat{a}_2 \in \mathbb{F}_{3^{2m}}$.

Duursma and Lee use a group of order $N = q^3 + 1$ to compute the Tate pairing. Barreto et al. relate this to the η_T pairing using Theorem 3.3.1. In this case, $T = q = 3^m$. The remaining variables are given by $a = 3$, $L = 1$ and $c = 0$. The value of M is $(q^k - 1)/N = (q^6 - 1)/(q^3 + 1) = q^3 - 1$. The relationship between the η_T and Tate pairings is given by

$$\left(\eta_T(P,Q)^M\right)^{3q^2} = \langle P, \psi(Q)\rangle_N^M \tag{6.6}$$

Further optimisations are possible. Let $N = \#E(\mathbb{F}_{3^m}) = 3^m + 1 \pm 3^{(m+1)/2}$. Given a point $P \in E(\mathbb{F}_q)$ then $[N]P = \infty$. It has already been established that an automorphism exists on the curve such that $\gamma(P) = [q]P$. It follows that

$$\begin{aligned}
\gamma(P) &= [q]P \\
&= [q]P - [N]P \\
&= [q - N]P \\
&= [3^m - (3^m + 1 \pm 3^{(m+1)/2})]P \\
&= [\mp 3^{(m+1)/2} - 1]P
\end{aligned}$$

The η_T pairing can, therefore, be calculated with $T = \mp 3^{(m+1)/2} - 1$. The remaining variables can now be selected. Since $T = q + cN$ then c must equal -1. From $T^a + 1 = LN$, choose $a = 3$. This means that $L = \mp 3^{(m+3)/2}$. The value of M is given by $M = (3^{6m} - 1)/N = (3^{6m} - 1)/(3^m + 1 \pm 3^{(m+1)/2}) = (3^{3m} - 1)(3^m + 1)(3^m \mp 3^{(m+1)/2} + 1)$. In this case, the η_T pairing is related to the Tate pairing by

$$\left(\eta_T(P,Q)^M\right)^{3T^2} = \left(\langle P, \psi(Q)\rangle_N^M\right)^L \tag{6.7}$$

The computation of the η_T pairing, followed by the conversion of the result to a Tate pairing value, are discussed for the remainder of this section.

6.2.1 Computation of the η_T Pairing

Given the points $P, Q \in E(\mathbb{F}_{3^m})$, then the η_T pairing is calculated according to $\eta_T(P,Q) = f_{T,P}(P, \psi(Q))$, where $f_{T,P}$ is the Miller function to be evaluated. The intermediate functions are computed using the fast tripling operation and the evaluations at $\psi(Q)$ are accumulated. For any point $A = (x_A, y_A)$ on the curve then

$$g_A(x,y) = y_A^3 y - (x_A^3 - x + b)^2 \tag{6.8}$$

where $div(g_A(x,y)) = 3(A) + (-3A) - 4(\infty)$.

If $\eta_T(P,Q)$ is to be computed and T is negative, then the input point P should be replaced by $-P$ and $-T$ should be used for computation. In practice, this means that $T = 3^{(m+1)/2} + b$ and $-P$ are used when $m \equiv (1,11) \mod 12$ whilst $T = 3^{(m+1)/2} - b$ and P are used when $m \equiv (5,7) \mod 12$. From Equations (3.13) and (3.14), the Miller function can be evaluated according to

$$f_{T,P}(\psi(Q)) = \left(\prod_{i=0}^{(m-1)/2} \left(g_{[3^i]P}(\psi(Q)) \right)^{3^{(m-1)/2-i}} \right) . l(\psi(Q) \tag{6.9}$$

where $l(\psi(Q))$ is the line function arising from the final addition of $[3^{(m+1)/2}]P$ with $\pm P$. As in the characteristic 2 case, the exponentiations to $3^{(m-1)/2-i}$ on each iteration can be avoided if the Miller function is accumulated in a different manner. Let $P' = [3^{(m-1)/2}]P$. From Equation (6.1), the value of P' can be computed on each iteration according to

$$P' = [\phi^{(m-1)/2} \pi^{m-1}]P = \phi^{(m-1)/2}(x_P^{1/3}, y_P^{1/3}) \tag{6.10}$$

The Miller function can now be computed according to

$$f_{T,P}(\psi(Q)) = l(\psi(Q)). \left(\prod_{j=0}^{(m-1)/2} g_{[3^{-j}]P'}(\psi(Q))^{3^j} \right) \tag{6.11}$$

The definition of the line function depends on the value of $m \mod 12$. The results at the end of this chapter are returned by implementations of the Tate pairing processor on fields with $m = 79$ and $m = 97$. These values return $m \mod 12 \equiv 7$ and $m \mod 12 \equiv 1$, respectively. In this section, only the line function for the latter case is provided for simplicity. The former case differs very slightly. Let $l(x, y)$ be the final line function. This has slope $\lambda = -y_P$. It can be shown that

$$l(x, y) = \lambda(x - x_P) + by_P \tag{6.12}$$

The cube roots that are required for the calculation of P' on each iteration can be avoided by precomputing a set of cubes and accessing them in reverse order. This means that a cube rooting module is not required in a hardware implementation.

The intermediate functions are calculated using Equation (6.8). For simplicity, the notation $a^{(i)} = a^{3^i}$ is used. Now

$$g_{[3^{-j}]P'}(\psi(Q))^{3^j} = (\sigma y_P^{(-j)} y_Q^{(j)} - u^2) - \rho u - \rho^2 \tag{6.13}$$

where $u = x_P^{(-j)} + x_Q^{(j)} + b$.

The operations required for the computation of $\eta_T(P, Q)$ in the $m \mod 12 \equiv 1$ case are provided in Algorithm 12. Note that the notation used on Line 15 refers to a ceiling function. The ceiling of a rational number a, denoted $\lceil a \rceil$, is the the least integer that is greater than or equal to a. The elliptic curve used for this work is $y^2 = x^3 - x - 1$, which means that $b = -1$. The required cubes of x_Q and y_Q are first precomputed. The values calculated in the *for* loop of Lines 2-4 do not need to be stored. The cubes of the second loop are stored in reverse order so that cube roots can be accessed with ease. The *for* loop that evaluates the intermediate functions $g \in \mathbb{F}_{3^{6m}}$ and accumulates the results begins on Line 9. The individual components of g are calculated on Lines 10-12. Note that squaring is not a trivial operation and is performed by multiplication of an element with itself to avoid the necessity of a subfield squaring module in the pairing processor. A total of one addition, two negations and two multiplications on $\mathbb{F}_{3^m}$ are required for the computation of g. A 1-bit addition is also required. The value of g is multiplied by the accumulating function f on Line 14 using a special multiplication routine $dmul(f, g)$. This routine takes advantage of the sparse nature of the g polynomial to minimise the quantity of required $\mathbb{F}_{3^m}$ multiplications. This special multiplication is described in the next section. On loop exit, the line function is calculated, requiring one addition, two negations and one multiplication on $\mathbb{F}_{3^m}$. A 1-bit addition must also be performed. Finally, the result is multiplied by the accumulated function f and the value of $\eta_T(P,Q)$ is returned.

Algorithm 12 Computation of $\eta_T(P,Q)$ on $E(\mathbb{F}_{3^m}) : y^2 = x^3 - x - 1$, $m \mod 12 \equiv 1$

INPUT: $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$, where $P, Q \in E(\mathbb{F}_{3^m})$

OUTPUT: $f = \eta_T(P,Q)$, where $f \in \mathbb{F}_{3^{6m}}$

Initialise: $f \leftarrow 1$

for $(i \leftarrow m-1,\ i > (m+1)/2,\ i \leftarrow i-1)$ **do**

    $x_Q \leftarrow {x_Q}^3$, $y_Q \leftarrow {y_Q}^3$

end for

for $(i \leftarrow (m+1)/2,\ i \geq 0,\ i \leftarrow i-1)$ **do**

    $x_Q \leftarrow {x_Q}^3$, $y_Q \leftarrow {y_Q}^3$

    $x'_Q[i] \leftarrow x_Q$, $y'_Q[i] \leftarrow y_Q$

end for

for $(i \leftarrow 1,\ i \leq (m+1)/2,\ i \leftarrow i+1)$ **do**

    $u \leftarrow x_P + x'_Q[i] + 1$

    $c_0 \leftarrow -u.u$

    $c_1 \leftarrow -y_P.y'_Q[i]$

    $g \leftarrow c_0 + (c_1)\sigma + (-u)\rho + (0)\sigma\rho + (-1)\rho^2 + (0)\sigma\rho^2$

    $f \leftarrow dmul(f, g)$

    if $i < \lceil \frac{m+1}{2} \rceil$ **then**

        $x_P \leftarrow x_P^3$, $y_P \leftarrow y_P^3$

    end if

end for

$g \leftarrow -y_P.(x'_Q[i] + x_P + 1) + (-y'_Q[i])\sigma + (y_P)\rho$

$f \leftarrow f.g$

RETURN: f

6.2.2 Exponentiation to the Tate Pairing

The value of $f = \eta_T(P,Q)$ must be exponentiated to return a unique Tate pairing result. Recall from Equation (6.7) that $((\eta_T(P,Q)^M)^{3T^2} = (\langle P, \psi(Q)\rangle_N^M)^L$. This means that the Tate pairing $\langle P, \psi(Q)\rangle_N^M$ can be calculated by raising f to the power $M3T^2/L$. This exponentiation can be performed by unrolling the different factors and cancelling when possible. This process was already discussed in detail in Chapters 4 and 5 and will not be described in detail in this case.

The exponentiation to M requires $((m+1)/2+1)$ cubings, 10 multiplications, one squaring, and an inversion, all performed on the extension field. The squaring is again performed by multiplication. The trivial 3^m-power Frobenius operation must also be performed nine times. Exponentiation to $3T^2/L$ is computed by first raising $\eta_T(P,Q)^M$ to the power $3T^2q/L$ and then applying the (inexpensive) inverse q-power Frobenius map. Another m cubings, one squaring, three multiplications and three conjugations, all on the extension field, are required to return the unique Tate pairing value $\langle P, \psi(Q)\rangle_N^M$.

6.3 Implementation of Arithmetic on $\mathbb{F}_3$, $\mathbb{F}_{3^m}$ and $\mathbb{F}_{3^{6m}}$

The characteristic 3 arrangement of the processor discussed here contains modules for $\mathbb{F}_{3^m}$ addition, subtraction, cubing, inversion and multiplication. Extension field operations are computed using these modules. The hardware implementation of arithmetic on $\mathbb{F}_3$ and $\mathbb{F}_{3^m}$ is discussed in this section. The extension field arithmetic operations required by the Tate pairing are then presented in terms of the necessary subfield operations.

6.3.1 Hardware Implementation of Arithmetic on $\mathbb{F}_3$

The binary nature of FPGAs means that storage and arithmetic operations on the $\mathbb{F}_2$ field is convenient. It contains the elements 0 and 1, each of which can be stored in one bit. The addition and multiplication operations are performed using one *XOR* and one *AND* gate, respectively. Implementation of the $\mathbb{F}_3$ field is more complicated as it contains the members 0, 1 and 2. These elements require two hardware bits for storage. In this work, the representations $\{00, 01\} = 0$, $\{10\} = 1$ and $\{11\} = 2$ are used. This mapping system means that a *check if zero* operation can be performed by inspecting the most significant bit of an $\mathbb{F}_3$ element.

Addition, subtraction, multiplication and negation must be performed on $\mathbb{F}_3$. Fortunately, FPGAs provide a relatively efficient means for the implementation of these operations. Input-output maps for each of the 2-bit arithmetic operations can be placed on a pair of 4-input, 1-output Look Up Tables (LUTs). Negation can be computed by subtraction from 0 to save area.

6.3.2 Computation and Implementation of Arithmetic on $\mathbb{F}_{3^m}$

Tate pairing computation requires addition, subtraction, cubing, multiplication and inversion on $\mathbb{F}_{3^m}$. These operations can be implemented in hardware in a similar manner to that of the $\mathbb{F}_{2^m}$ field and are briefly discussed here.

Addition is performed coefficient-wise and is implemented on $2m$ LUTs, each with input-output maps corresponding to $\mathbb{F}_3$ addition. Subtraction requires $2m$ of the $\mathbb{F}_3$ subtraction LUTs. These operations are combinatorial and a result can be returned in one clock cycle.

Cubing is implemented using the techniques of Bertoni et al. [102]. Cubing is a linear operation on $\mathbb{F}_3$. The $\mathbb{F}_{3^m}$ input polynomial is first padded with zeros, resulting in a composition polynomial of degree $(3m-3)$. This is then separated into lower, middle and upper intermediate polynomials, each of degree $(m-1)$. Let $f = x^m \pm x^t \pm 1$ be the irreducible polynomial defining $\mathbb{F}_{3^m}$. If $t < m/3$ then the lower intermediate polynomial does not need to be reduced. This is the case for the implementations of this work. The middle and upper polynomials can be reduced in one step using combinatorial logic. A total area of $4m$ LUTs are required and a result is returned in just one clock cycle.

Multiplication is performed using a characteristic 3 version of the digit-serial multipliers previously described in Subsection 4.3.2. The multipliers operate on D coefficients of one of the input polynomials at a time. A result is returned in m/D clock cycles. In the characteristic 3 case, the input polynomials have twice the number of bits. As before, a definition of the required subfield operations is generated by product and reduction matrices that are dependent on the field size and irreducible polynomial. The characteristic 3 digit-serial multipliers require more area than their characteristic 2 counterparts as LUTs must be used instead of gates. An overview of the characteristic 3 digit-serial multipliers is available in [102].

Inversion on $\mathbb{F}_{3^m}$ is computed using a characteristic 3 variant of the extended Euclidean algorithm. Implementation using this method is discussed in [103]. The $\mathbb{F}_{3^m}$ inverter contains two chains of $\mathbb{F}_3$ arithmetic modules and some storage and control circuitry. A result is returned in $2m$ clock cycles.

6.3.3 Arithmetic on $\mathbb{F}_{3^{6m}}$

The η_T pairing of Algorithm 12 requires multiplication of the g polynomial by the accumulating f polynomial on each iteration of the loop using the $dmul(f, g)$ routine. An extension field multiplication is also required after loop exit. Exponentiation of the result to the Tate pairing requires addition, subtraction, multiplication, squaring, cubing, powering to q (the Frobenius mapping) and conjugation, all on $\mathbb{F}_{3^{6m}}$.

Addition and subtraction are again performed coefficient-wise. Conjugation can be performed by first converting the operand to a pair of $\mathbb{F}_{3^{3m}}$ elements. This requires addition and subtraction alone. Extension field squaring is expensive and is performed by multiplying elements by themselves. The other extension field operations are described in more detail in this subsection. Note that the equations of this subsection are for a field with $m \mod 3 \equiv 1$ (as is the case for $m = 97$) with $b = -1$ on the elliptic curve $E(\mathbb{F}_{3^m}) : y^2 - x^3 - x + b$. These are the parameters used by Barreto et al. in [63]. Minor changes are required in other cases.

The $dmul(f, g)$ Routine

The multiplication of f by g can be performed by regular extension field multiplication. This would not, however, exploit the sparse form of g. It is important that this multiplication be performed as efficiently as possible as it is performed on each iteration of the main *for* loop.

The intermediate polynomials g have the structure

$$g = (g_0 + g_1\sigma + g_2\rho + (0)\sigma\rho + (-1)\rho^2 + (0)\sigma\rho^2)$$

Consider $f = (f_0 + f_1\sigma + f_2\rho + f_3\sigma\rho + f_4\rho^2 + f_5\sigma\rho^2)$, where $f_0, f_1, f_2, f_3, f_4, f_5 \in \mathbb{F}_{3^m}$. The $dmul(f, g)$ routine proceeds with 13 $\mathbb{F}_{3^m}$ multiplications, all of which can be performed in parallel, if desired.

$$\begin{aligned}
mul_0 &= f_0.g_0 & mul_7 &= (f_0 + f_1 + f_2 + f_3).(g_0 + g_1 + g_2) \\
mul_1 &= f_1.g_1 & mul_8 &= (f_0 + f_4).(g_0 + 2) \\
mul_2 &= (f_0 + f_1).(g_0 + g_1) & mul_9 &= (f_1 + f_5).(g_1) \\
mul_3 &= f_2.g_2 & mul_{10} &= (f_0 + f_1 + f_4 + f_5).(g_0 + g_1 + 2) \\
mul_4 &= f_3.g_2 & mul_{11} &= (f_2 + f_4).(g_2 + 2) \\
mul_5 &= (f_0 + f_2).(g_0 + g_2) & mul_{12} &= (f_3 + f_5).(g_2 + 2) \\
mul_6 &= (f_1 + f_3).(g_1) & &
\end{aligned} \tag{6.14}$$

The final result $c = dmul(f,g)$, where $c \in \mathbb{F}_{3^{6m}}$, is returned by the following two steps, which require $\mathbb{F}_{3^m}$ addition, negation and subtraction alone.

$$\begin{aligned}
t00_r &= mul_0 - mul_1 & t00_i &= mul_2 - mul_0 - mul_1 \\
t11_r &= mul_3 & t11_i &= mul_4 \\
t22_r &= -f_4 & t22_i &= -f_5 \\
t01_r &= mul_5 - mul_6 & t01_i &= mul_7 - mul_5 - mul_6 \\
t02_r &= mul_8 - mul_9 & t02_i &= mul_{10} - mul_8 - mul_9 \\
t12_r &= mul_{11} & t12_i &= mul_{12}
\end{aligned} \tag{6.15}$$

$$\begin{aligned}
c_0 &= t00_r - t12_r + t11_r + t22_r \\
c_1 &= t00_i - t12_i + t11_i + t22_i \\
c_2 &= t01_r - t00_r + t11_r + t12_r + t22_r \\
c_3 &= t01_i - t00_i + t11_i + t12_i + t22_i \\
c_4 &= t02_r - t00_r + t11_r \\
c_5 &= t02_i - t00_i + t11_i
\end{aligned} \tag{6.16}$$

Note that the addition and subtraction of elements of $\mathbb{F}_{3^m}$ with members of $\mathbb{F}_3$ is performed using $\mathbb{F}_{3^m}$ modules and are counted as arithmetic on $\mathbb{F}_{3^m}$. The $dmul(f,g)$ routine requires a total of 13 $\mathbb{F}_{3^m}$ multiplications and 51 $\mathbb{F}_{3^m}$ additions/subtractions.

$\mathbb{F}_{3^{6m}}$ Multiplication

Extension field multiplication using Karatsuba methods requires six multiplications on $\mathbb{F}_{3^{2m}}$. Each of these operations require three multiplications on $\mathbb{F}_{3^m}$. A total of 18 $\mathbb{F}_{3^m}$ multiplications and 72 additions/subtractions are required. Gorla et al. show how extension field multiplication can be performed using techniques based on the fast Fourier transform [104]. The fourth roots of unity of $\mathbb{F}_{3^{6m}}$ are used to reduce the number of required $\mathbb{F}_{3^{2m}}$ multiplications to 15. They also show how a discrete Fourier transform matrix can be used to devise explicit formulae for extension field multiplication. The first stage requires 15 multiplications on $\mathbb{F}_{3^m}$, all of which can be performed in parallel. Reduction is performed after this, requiring some additions and subtractions. A total of 15 multiplications and 90 additions/subtractions on $\mathbb{F}_{3^m}$ are required. This provides a saving of three multiplications over the Karatsuba method, at the expense of 18 more trivial additions/subtractions.

$\mathbb{F}_{3^{6m}}$ Cubing

The extension field cubing operation uses the 3-tuple representation of Equation (6.5). Let $a = \hat{a}_0 + \hat{a}_1\rho + \hat{a}_2\rho^2$, where $\hat{a}_0 = a_0 + a_1\sigma$, $\hat{a}_1 = a_2 + a_3\sigma$, $\hat{a}_2 = a_4 + a_5\sigma$ and $\hat{a}_0, \hat{a_1}, \hat{a_2} \in \mathbb{F}_{3^{2m}}$ and $a_0, a_1, a_2, a_3, a_4, a_5 \in \mathbb{F}_{3^m}$. First

$$a^3 = \hat{a}_0^3 + \hat{a}_1^3\rho^3 + \hat{a}_2^3\rho^6 \tag{6.17}$$

From $h(z) = z^3 - z + 1$, the irreducible polynomial generating the extension from $\mathbb{F}_{3^{2m}}$ to $\mathbb{F}_{3^{6m}}$, the relationships $\rho^3 = \rho - 1$ and $\rho^6 = \rho^2 + \rho + 1$ hold. Now

$$\begin{aligned} a^3 =& \hat{a}_0^3 + \hat{a}_1^3(\rho - 1) + \hat{a_2}^3(\rho^2 + \rho + 1) \\ =& (\hat{a}_0^3 - \hat{a}_1^3 + \hat{a}_2^3) + (\hat{a}_1^3 + \hat{a}_2^3)\rho + \hat{a}_2^3\rho^2 \end{aligned} \tag{6.18}$$

The irreducible polynomial defining $\mathbb{F}_{3^{2m}}$ is $g(y) = y^2 + 1$. Then $\sigma^2 = -1$ and therefore

$$\begin{aligned} \hat{a}_0^3 =& ({a_0}^3 - \sigma {a_1}^3) \\ \hat{a}_1^3 =& ({a_2}^3 - \sigma {a_3}^3) \\ \hat{a}_2^3 =& ({a_4}^3 - \sigma {a_5}^3) \end{aligned} \tag{6.19}$$

Substituting (6.18) into (6.19) and rearranging gives

$$\begin{aligned} a^3 =& ({a_0}^3 - {a_2}^3 + {a_4}^3) + ({a_3}^3 - {a_1}^3 - {a_5}^3)\sigma \\ & + ({a_2}^3 + {a_4}^3)\rho + (-{a_3}^3 - {a_5}^3)\sigma\rho \\ & + ({a_4}^3)\rho^2 + (-{a_5}^3)\sigma\rho^2 \end{aligned} \tag{6.20}$$

A cubing on $\mathbb{F}_{3^{6m}}$,therefore, requires two additions, six subtractions and six cubings on $\mathbb{F}_{2^m}$, all of which can be performed using combinatorial logic.

$\mathbb{F}_{3^{6m}}$ Powering to q

Consider $a = \hat{a}_0 + \hat{a}_1\rho + \hat{a}_2\rho^2$, as before. Powering to q begins with the computation

$$\begin{aligned} a^q =& \hat{a}_0^{3^m} + \hat{a}_1^{3^m}\rho^{3^m} + \hat{a_2}^{3^m}(\rho^2)^{3^m} \\ =& \hat{a}_0^{3^m} + \hat{a}_1^{3^m}\rho^3 + \hat{a_2}^{3^m}\rho^6 \end{aligned} \tag{6.21}$$

The equations $\rho^3 = \rho - 1$, $\rho^6 = \rho^2 + \rho + 1$, $\sigma^2 = -1$ and $\sigma^3 = -\sigma$ can be derived from the irreducible polynomials. Now

$$\begin{aligned} a^q =& \hat{a}_0^{3^m} + \hat{a}_1^{3^m}(\rho - 1) + \hat{a}_2^{3^m}(\rho^2 + \rho + 1) \\ =& (\hat{a}_0^{3^m} - \hat{a}_1^{3^m} + \hat{a}_2^{3^m}) + (\hat{a}_1^{3^m} + \hat{a}_2^{3^m})\rho + \hat{a}_2^{3^m}\rho^2 \\ =& ((a_0^{3^m} - \sigma a_1^{3^m}) - (a_2^{3^m} - \sigma a_3^{3^m}) + (a_4^{3^m} - \sigma a_5^{3^m})) \\ &+ (a_2^{3^m} - \sigma a_3^{3^m} + a_4^{3^m} - \sigma a_5^{3^m})\rho + (a_4^{3^m} - \sigma a_5^{3^m})\rho^2 \end{aligned} \tag{6.22}$$

Since $a_i^q = a_i$ for all $a_i \in \mathbb{F}_{3^m}$

$$\begin{aligned} a^q =& (a_0 - a_2 + a_4) + (a_3 - a_1 - a_5)\sigma \\ &+ (a_2 + a_4)\rho + (-a_3 - a_5)\sigma\rho + a_4\rho^2 + (-a_5)\sigma\rho^2 \end{aligned} \tag{6.23}$$

Powering to q is, therefore, trivial and can be performed in two additions and six subtractions on $\mathbb{F}_{3^m}$.

$\mathbb{F}_{3^{6m}}$ Inversion

The techniques described by Kerins et al. are used here to perform $\mathbb{F}_{3^{6m}}$ inversion [103]. The $\mathbb{F}_{3^{6m}}$ field is reconstructed as a degree 2 extension of $\mathbb{F}_{3^{3m}}$:

$$\begin{aligned} \mathbb{F}_{3^{3m}} &\equiv \mathbb{F}_{3^m}[y]/h(y) \text{ where } h(y) = y^3 - y + 1 \\ \mathbb{F}_{3^{6m}} &\equiv \mathbb{F}_{3^{3m}}[z]/g(z) \text{ where } g(z) = z^2 + 1 \end{aligned} \tag{6.24}$$

An element $a \in \mathbb{F}_{3^{6m}}$ is written as $a = \hat{a}_0 + \hat{a}_1\sigma$ where $\hat{a}_0 = a_0 + a_1\rho + a_2\rho^2$ and $\hat{a}_1 = a_3 + a_4\rho + a_5\rho^2$ for $\hat{a}_0, \hat{a}_1 \in \mathbb{F}_{3^{3m}}$. Since $\sigma^2 = -1$, the inversion can be carried out efficiently using conjugate methods. A full $\mathbb{F}_{3^{6m}}$ inversion costs 33 multiplications, four cubings, 67 additions/subtractions and one inversion, all on $\mathbb{F}_{3^m}$.

6.4 The Flexible Tate Pairing Processor

The architecture of the flexible Tate pairing processor is discussed in this section. The processor can compute a Tate pairing on both characteristic 2 and 3 curves. The efficient sequencing of subfield operations is paramount. A flexible software subsystem that enables the rapid generation of instruction sequences is described in detail. Processors that can be used in a large number of applications and environments can also be analysed and created with ease by using the features of this subsystem in conjunction with the overall design system discussed in Subsection 3.5.3. Results that are returned by various implementations of the pairing processor are also presented in this section.

6.4.1 Architecture

The flexible processor architecture does not contain extension field arithmetic units. An ALU containing a number of subfield modules that operate in parallel is used instead. The characteristic 2 ALU contains an $\mathbb{F}_{2^m}$ adder, squarer, inverter and a number of $\mathbb{F}_{2^m}$ multipliers. The characteristic 3 ALU contains one module for each of addition, subtraction, cubing, inversion and a number of modules for $\mathbb{F}_{3^m}$ multiplication. As multiplication is performed so often and is a time consuming operation the system has been designed so that the number of multipliers in an ALU can be varied with little impact on the overall architecture. The digit size of the multipliers is also variable.

An example of a characteristic 3 Tate pairing processor, generated by the design system, is illustrated in Figure 6.1. In this case, the ALU contains k $\mathbb{F}_{3^m}$ digit-serial multipliers. The pairing inputs and intermediate variables are stored centrally in dual port $2m$-bit RAM. A pair of $2m$-bit buses are sent from the RAM to the ALU and are connected to the subfield modules. The ALU also contains $(k+3)$ $2m$-bit tri-state buffers that select the module output to be stored. The *buff_sel* control bus handles the buffers. On completion of the Tate pairing computation the *done* signal is asserted and the $\mathbb{F}_{3^{6m}}$ Tate pairing value is read serially from the first RAM output. Note that the *DOut* bus is tied to an output enable control signal so that intermediate values cannot be read from it if pairing computation halts before completion.

The architecture implementing the Tate pairing in characteristic 2 has a very similar structure as ease of architectural modification is a fundamental design principle. The modules within the ALUs perform $\mathbb{F}_{2^m}$ arithmetic. The data buses are m bits wide and m-bit RAM is used. The scheduling of operations is also modified to implement the new pairing.

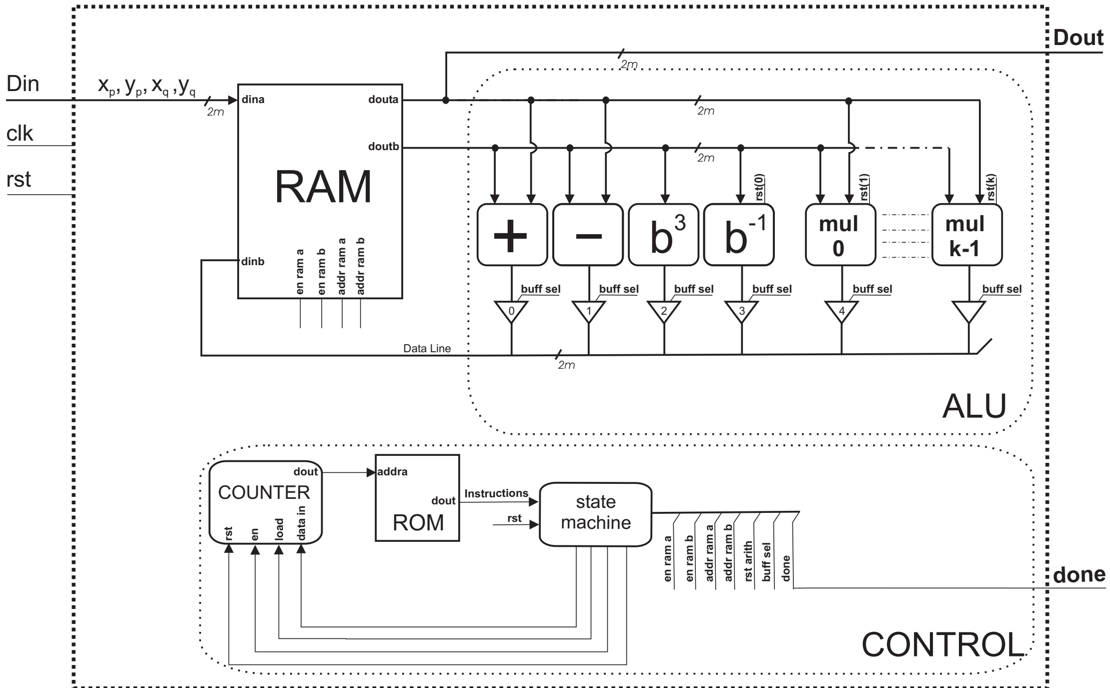

Figure 6.1: Characteristic 3 Tate pairing processor containing k multipliers

6.4.2 Operation Scheduling

Scheduling must be performed with care as a large amount of subfield arithmetic must be implemented. When possible, addition, subtraction, cubing and squaring are computed and stored while multiplication and inversion are in progress. Particular care is taken over the scheduling of operations within the main *for* loops of the η_T algorithms as these loops must be iterated many times and require operations that are expensive in terms of clock cycles.

As an example of the scheduling techniques employed, consider the characteristic 3 η_T pairing computation of Algorithm 12. The first values of $c_0 = -u.u$ and $c_1 = -y_P.y'_Q[i]$ are computed and stored in RAM before the hardware begins to implement the loop. After this, the computation $f \leftarrow dmul(f, g)$ of the first iteration can begin immediately as its inputs will be available. The values of c_0 and c_1 that will be required on the second iteration can be computed in parallel with the *dmul* operation of this first iteration. This scheduling technique means that 15 $\mathbb{F}_{3^m}$ multiplications required on each iteration can be performed in parallel, if desired. The additions required by the *dmul* routine are also carefully scheduled so that they are performed while the multipliers are in use.

The quantity of operations required for η_T computation far exceeds that required to perform the exponentiation to the Tate pairing value. This conversion should, however, also be scheduled intelligently as it is more serial in nature and there is less parallelism available as a result.

6.4.3 Control System

The aim of the control system is to minimise the quantity of clock cycles required to handle storage and module resets and enables. The hardware units are chosen to reduce the work that is required of a designer to generate the control signals that are necessary to perform the pairing. The control unit consists of a counter, a ROM and an FSM. A sequence of instructions that describe the operations required for Tate pairing computation is loaded into the ROM. The ROM output is connected to the FSM. At the beginning of a pairing computation, the FSM resets the counter and then enables it. The counter is then used to access the instructions consecutively.

Most bit sequences of the ROM instructions are used to handle RAM reads and writes, to reset and enable the modules within the ALU and to control the tri-state buffers. These bits pass straight through the FSM module and appear at one of its outputs. A small number of bits are used to communicate with the FSM. The FSM checks these bits on every clock cycle and will enter a particular state when required. If, for example, an $\mathbb{F}_{3^m}$ multiplication must be performed, an instruction will provide RAM read addresses and reset and enable signals for the chosen multiplier and indicate to the FSM that a *mul_mode* state should be entered. This state halts the counter for m/D clock cycles. This means that the next instruction, which will contain control signals for storage of the result, will not be accessed until the multiplication operation has been completed. The counter also contains a *load* control bit and a *data_in* bus. When the *load* bit is asserted the counter will change its output to the value on the *data_in* bus. An instruction can indicate to the FSM that it should enter a *jump_mode* and provides the desired counter output value. This is useful as this means that it is not necessary that the ROM be accessed in a consecutive fashion. Jumps can be used to return to the beginning of a long sequence of operations that must be repeated. As an example, a jump is performed at the end of each iteration of the main *for* loop of the η_T computation.

This system provides a simple method for the provision of flexibility. It eliminates the requirement for a large, complicated FSM as its sole purpose is to control the counter. The FSM does not need to be modified for the implementation of other algorithms. A new sequence of instructions can be generated and simply loaded into ROM. This versatility is very important as it means the processors can be used to implement other finite field based cryptographic operations with ease.

6.4.4 Flexible Design System for Processor Creation and Implementation

A software design system (*design_sys*) that has been created for software pairing computation, VHDL generation, hardware implementation, verification and benchmarking was discussed in Subsection 3.5.3. The software is written in the *C++* language. It contains a class, called *flex_sub_sys*, that has not been described up to this point. This provides added functionality when designing, analysing and implementing the processors described in this chapter. It is also written in *C++*. The features of this subsystem are described in this subsection.

The flexible processors do not contain extension field arithmetic units. This means that many streams of bits describing $\mathbb{F}_{2^m}$ or $\mathbb{F}_{3^m}$ operations must be defined to implement a pairing algorithm. These streams must be converted into sequences of instructions. This can be a very long, tedious process that has an inherently high probability of error. Furthermore, if errors are made, it is difficult to isolate the problem due to the large number of operations involved.

The overall design system already contains a class for software pairing computation. This contains base classes for performing arithmetic on $\mathbb{F}_{2^m}$, $\mathbb{F}_{2^{4m}}$, $\mathbb{F}_{3^m}$, $\mathbb{F}_{3^{2m}}$ and $\mathbb{F}_{3^{6m}}$. Algorithms are computed in software using functions within these classes. An instruction set generation class has been created that provides similar functionality. This contains a subfield base class in which sequences of instructions are defined for subfield arithmetic. An extension field base class has been created in which instruction sequences for extension field arithmetic can be defined using the instruction sequences within the subfield class. A full sequence of instructions can then be generated to implement a pairing algorithm that can be added to the system in a very similar fashion to that of the software computation case.

The subfield base class contains members of integer type. Variables hold the RAM addresses of subfield elements. Arithmetic operations are overloaded so that instruction sequences are automatically generated when they are used. The instructions define the RAM addresses and provide the necessary control signals. As an example, consider the class members v, w and z that contain the RAM locations of three subfield variables. Let $v = 10$, $w = 20$ and $z = 30$. If the operation $z = v + w$ is entered, two instructions are generated. The first puts *addr ram a* to 10, *addr ram b* to 20 and opens the tri-state buffer at the output of the adder. The next instruction puts the value of 30 on *addr ram b* and writes the result to that address. Multiplication is a special case as the number of modules is variable. The multiplicative operator is overloaded in such a manner that it can take arrays of class members as operands. For example, consider $v = [10, 11, 12, 13, 14]$, $w = [20, 21, 22, 23, 24]$ and $z = [30, 31, 32, 33, 34]$ containing the locations of five subfield elements each. Five multiplications must be performed. The software checks whether this exceeds the number of multipliers in the processor. If it does, the software minimises the number of multiplication runs required. If, for example, there are three multipliers, then

$30 = 10 \times 20$, $31 = 11 \times 21$ and $32 = 12 \times 22$ are first performed in parallel and the remaining multiplications performed on the next run.

The members of the extension field class are k-length arrays of subfield members, where k is the extension degree of the curve in use. Extension field arithmetic is defined in terms of operations on members of the subfield class. As an example, let $r = [40, 41, 42, 43]$, $s = [50, 51, 52, 53]$ and $t = [60, 61, 62, 64]$ be elements of the $\mathbb{F}_{2^{4m}}$ extension field. If the command $t = r + s$ is entered, a sequence of instructions is automatically generated that contains the address and control signals required to implement the four $\mathbb{F}_{2^m}$ additions of the members of s and t and to store the result in the RAM addresses defined by t. Inversion and multiplication are defined in the same manner. The combination of these classes means that the instruction sequences required for extension field arithmetic do not have to be written manually: the sequences for the subfield operations have already been defined and are used.

A designer can now define a desired pairing algorithm in terms of operations on members of the subfield and extension field classes. Commands can be entered using all of the functionality of the *C++* language. The instruction sequence is then automatically generated for any desired characteristic, field size, quantity of multipliers and digit size. This system significantly reduces design cycle time as many algorithms and processors can be implemented without having to manually update the sequences.

The automatic generation of the instruction sequences within the software system enables rapid analysis of the clock cycles required by various algorithms when implemented on a processor. A breakdown of the cost of all operations is available immediately after algorithmic definition. These costs can be returned for any desired version of the processor. This means that detailed analysis can be performed within the software system and avoids the necessity for hardware implementation. This can be very beneficial while exploring the use of hardware for various applications and environments.

All of the other features of the design system are available when working with the flexible processors. VHDL can be automatically generated by the *vhdl_sub_sys* class according to a desired characteristic, field size, digit size and quantity of multipliers. Xilinx tools can be called, bit files automatically generated and FPGA communication handled within the *imp_sub_sys*. Sets of elliptic curve points can be generated in the *soft_sys_class* and automatically sent to the FPGA. Verification is performed by sending these input points to the processor and comparing the hardware results to those of the software computations. Benchmarking and efficiency analysis be performed. If more than one implementation is required, an array can be used to define the desired architectures and implementation, verification and benchmarking performed automatically. This makes the system particularly desirable for the prototyping of these types of architectures.

6.4.5 Results and Comparisons

Various versions of the architecture discussed in this chapter were implemented on the Xilinx Virtex-II Pro FPGA (xc2vp100-6ff1696). This device contains 44,096 slices.

The results returned by other characteristic 3 elliptic curve pairing implementations in the literature are for fields with $m = 97$. For this reason, versions of the flexible processor were built using several digit sizes and quantities of multipliers on $\mathbb{F}_{3^{97}}$. This field returns approximately the same level of security as a characteristic 2 field with $m = 271$. A number of characteristic 2 processors were also implemented on $\mathbb{F}_{2^{271}}$ so that a direct comparison could be made. The fast characteristic 2 elliptic curve Tate pairing processor discussed in Chapter 4 was implemented on the field $\mathbb{F}_{2^{313}}$. Results returned by the flexible processor at this field size are also presented in this section so that the two types of processors can be compared.

The results returned by the flexible processor when implemented on $\mathbb{F}_{2^{271}}$ and $\mathbb{F}_{3^{97}}$ are presented in Tables 6.1 and 6.2, respectively. A full Tate pairing $\langle P, \psi(Q) \rangle_N^M$ is performed in each case. The processors were implemented with multipliers of digit size 4, 8, 12 and 16 in both cases. Larger digit sizes are not used as, with further increases, the efficiencies of the resultant processors fall dramatically as the clock cycles required for control and combinatorial operations are fixed. The quantities of subfield multiplication modules included in the implementations are shown in the first column. The characteristic 2 processors were built using one, two, three, four, five and seven multipliers. A total of 14 $\mathbb{F}_{2^m}$ multiplications must be performed during the main iterative loop of the η_T pairing computation (see Algorithm 5). All of these operations can be performed in parallel if sufficient resources are available. A processor containing five multipliers requires three runs to complete these 13 multiplications. A processor with six multipliers needs the same number of runs and is not constructed for this reason. In the characteristic 3 case, a total of 15 $\mathbb{F}_{3^m}$ multiplications are required during loop iteration (Algorithm 12). Processors with six and seven multipliers do not provide an advantage over a processor containing five modules and are, therefore, not implemented.

The fastest characteristic 2 processor contains seven multipliers of digit size 16 and returns a Tate pairing result in 17,347 cycles. A total of 20,591 slices are required, yielding an AC product of 358 Mslice.cycles. The lowest AC product is 160 Mslice.cycles and is returned by two versions. The first contains two multipliers of digit size 8 and provides a pairing result in 30,912 cycles. This implementation requires 5,170 slices. The second contains two multipliers of digit size 12 and provides a result in 25,110 cycles. This implementation occupies 6,365 slices. On analysis of the results presented in Table 6.1, it is clear that the architectures that require a relatively small area can return a Tate pairing value in a relatively low number of clock cycles.

The fastest characteristic 3 version contains four multipliers of digit size 16. This implementation returns a Tate pairing result in 16,127 cycles and requires 22,240 slices. Note that this is not the largest of the implementations. In the $\{\#M = 5, D = 16\}$ and $\{\#M = 8, D = 16\}$ cases the reduction in multiplicative runs is counteracted by the cost of the control and storage cycles associated with the extra multipliers. The lowest AC product is 140 Mslice.cycles. This is provided by an architecture with three multipliers of digit size 4 and the implementation occupies 6,690 slices. A Tate pairing result is returned in 20,996 clock cycles. At moderate clock frequencies this provides a substantial reduction

Characteristic 2, m=271				
	Digit Size			
	4	8	12	16
#M	**Area** (slices)			
1	3360	3748	4441	5106
2	4389	5170	6365	7716
3	5318	6519	8149	10362
4	6202	7905	10272	12920
5	7131	9146	12114	15632
7	8989	12111	15963	20591
	Cycles			
1	86508	51072	39870	33947
2	48924	30912	25110	22576
3	37692	24960	21510	18592
4	32616	22272	19800	19173
5	27108	20544	19170	18592
7	23760	18528	17460	17347
	AC Product (Mslice.cycles)			
1	291	192	178	174
2	215	160	160	175
3	201	163	176	193
4	203	177	204	248
5	194	188	233	291
7	214	225	279	358

Table 6.1: Tate pairing Results for $\mathbb{F}_{2^{271}}$

Characteristic 3, m=97				
	Digit Size			
	4	8	12	16
#M	**Area** (slices)			
1	4347	4979	6300	7670
2	5366	7016	9821	12639
3	6690	9052	13246	17441
4	7992	11093	16888	22240
5	9336	13147	19071	27049
8	13369	19368	29662	39956
	Cycles			
1	41249	28437	24155	22010
2	26646	20096	18316	17564
3	20996	17618	16941	16595
4	19794	16573	16436	16127
5	18409	16855	16329	16145
8	17835	16581	16376	16299
	AC Product (Mslice.cycles)			
1	179	142	152	169
2	143	141	180	222
3	140	159	224	289
4	158	184	278	359
5	172	222	311	437
8	238	321	486	651

Table 6.2: Tate pairing results for $\mathbb{F}_{3^{97}}$

in computation time in comparison to the software result of 2.72 ms provided in [63].

It is useful to examine how changes in architectural parameters affect the AC product. The AC products of the characteristic 2 processor implementations are plotted against the number of multipliers employed for digit sizes of 4, 8, 12 and 16 in Figure 6.2. It is clear that the product increases once the number of multipliers surpasses two in all but the $D = 4$ case. This comes about as the proportion of cycles required for control and storage grows as the number of multiplication runs decreases. Overall, the $D = 8$ case provides a high degree of efficiency throughout. The AC products of the characteristic 3 implementations are plotted in the same manner in Figure 6.3. In all but the $D = 4$ case, the products grow when the number of multipliers exceeds two. Increases in digit size have a larger area impact in the characteristic 3 case due to the architecture of the multipliers. The $D = 4$ case provides a relatively low product throughout. On examination of Figures 6.2 and 6.3, it is clear that the characteristic 3 version of the processor is the more efficient in the majority of cases. It can also be seen that, in general, maximal efficiency is returned by implementations with smaller area footprints.

The characteristic 2 and 3 processor implementations can also be directly compared. Clock cycles are plotted against area for versions of the processors on $\mathbb{F}_{2^{271}}$ and $\mathbb{F}_{3^{97}}$ in Figure 6.4. While both processors perform well, the characteristic 3 implementation returns the more desirable results in the overwhelming majority of cases. The plot points at which the lowest AC product occurs in the characteristic 2 and 3 cases are also noted. The design environment enables the rapid and automatic generation of many versions of the flexible processor, as demonstrated by the quantity of results returned. Solutions for various applications can be explored quickly and with minimum effort if desired.

Various versions of the processor were also implemented on an elliptic curve defined on the field $\mathbb{F}_{2^{313}}$. This means that a direct comparison with the characteristic 2 elliptic and genus 2 hyperelliptic processors of Chapters 4 and 5 can be made as those implementations provide the same security level. The results returned by the flexible processor when implementing the Tate pairing on $\mathbb{F}_{2^{313}}$ are presented in Table 6.3. The most efficient im-

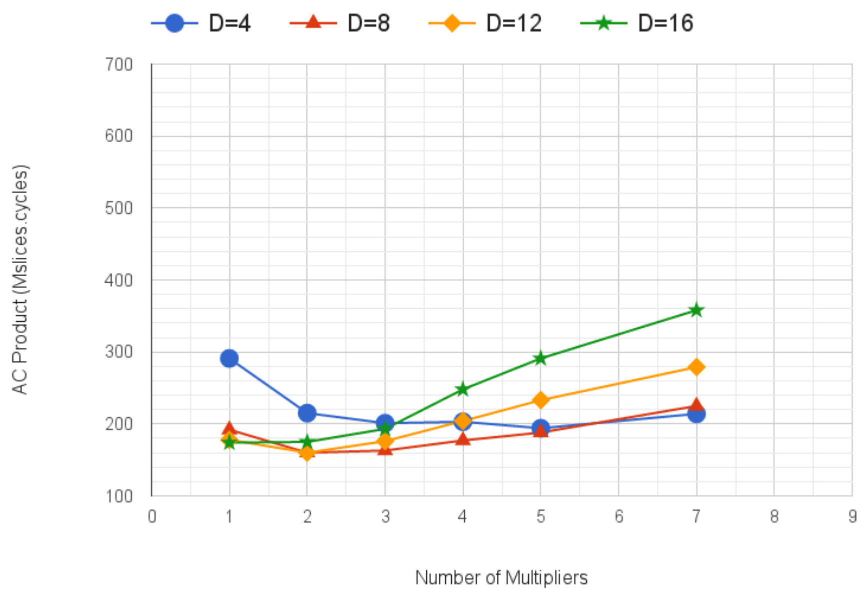

Figure 6.2: Processor AC products on $\mathbb{F}_{2^{271}}$

plementation is the $\{\#M = 2, D = 12\}$ case. This has an AC product of 223 Mslice.cycles, requires 7,120 slices, and returns a Tate pairing result in 31,239 clock cycles.

The AC products of the $\mathbb{F}_{2^{313}}$ implementations are illustrated in Figure 6.5. It is clear that the most efficient architectures are those with three multipliers or less. As in the $\mathbb{F}_{2^{271}}$ and $\mathbb{F}_{3^{97}}$ cases, the flexible processor is most efficient when a relatively small area footprint is used.

Comparisons

The performance of the flexible processor is compared with the processors discussed in Chapters 4 and 5 and with other hardware pairing architectures in the literature in this subsection. A summary of results is provided in Table 6.4. Note that only the most

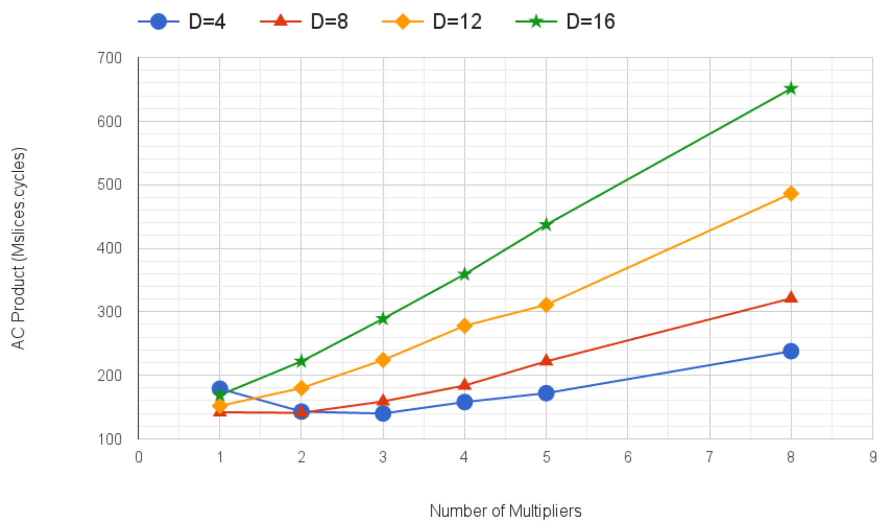

Figure 6.3: Processor AC products on $\mathbb{F}_{3^{97}}$

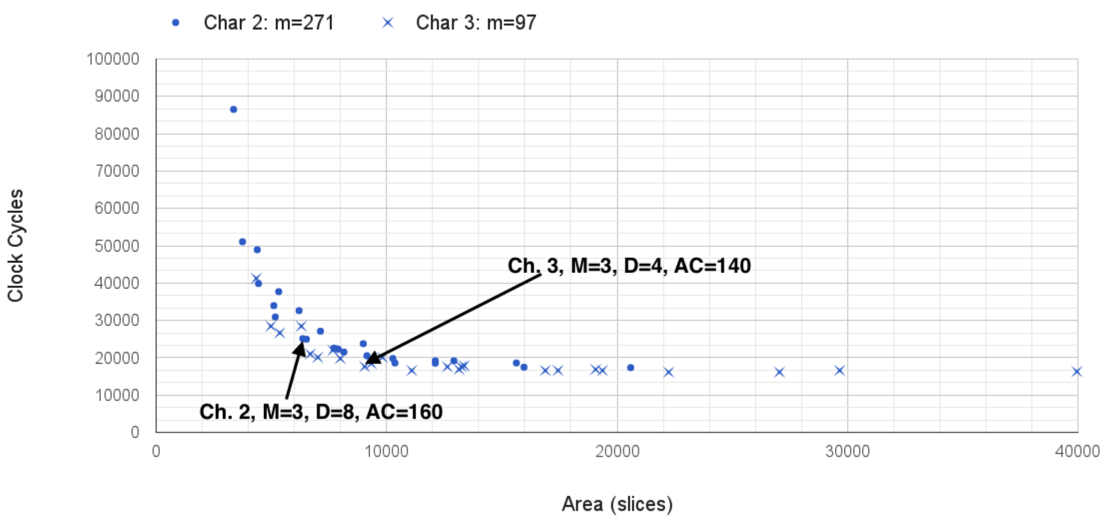

Figure 6.4: Clock cycles versus area for $\mathbb{F}_{2^{271}}$ and $\mathbb{F}_{3^{97}}$

efficient implementations results of Chapters 4, 5 and 6 are listed as these processors and their results have already been discussed in detail.

#M	**Area** (slices)				**Cycles**				**AC** (Msl.cycles)			
	Digit Size				**Digit Size**				**Digit Size**			
	4	8	12	16	4	8	12	16	4	8	12	16
1	3835	4283	4910	5838	112457	65740	50285	42444	432	282	247	248
2	5026	5806	7120	8839	62702	39140	31239	27135	316	228	223	240
3	6185	7232	9329	11858	47936	31160	25721	21708	297	226	240	258
4	7069	8770	11516	14840	41195	27645	23229	22437	292	243	268	333
5	8138	10409	13753	17858	33919	24700	22428	21708	277	258	309	388
7	10276	13543	18111	23723	29104	22040	20470	19926	300	299	371	473

Table 6.3: Implementation results for $\mathbb{F}_{2^{313}}$

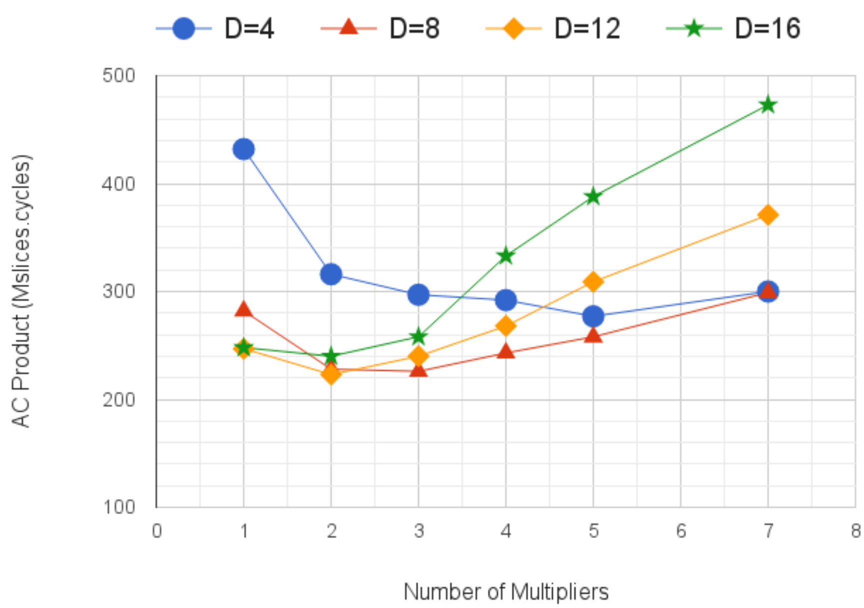

Figure 6.5: Processor AC products on $\mathbb{F}_{2^{313}}$

The characteristic 2 implementations of Keller et al. [8] and Shu et al. [95] have already been compared to the elliptic curve characteristic 2 processor of Chapter 4. The best AC product of the flexible processor of 160,000 slice.cycles is much lower than the 1,026,000 slice.cycles of the most efficient implementation of Keller et al. Their fastest architecture returns a pairing in 120,000 cycles and requires 15,065 slices. The flexible processor com-

Table 6.4: Comparisons with results returned by Tate pairing hardware implementations in the literature

Ref.	**m**	**Alg.**	**Device**	$\#M$	**D** (Bits)	**Area** (Slices)	**Cycles**	**AC** (kSl.cy)
Characteristic 2 Implementations from the Literature								
[8]	283	BKLS	V-II	Ma. 9	4	27411	68250	1871
				Ma. 9	6	29421	55110	1621
				Mi. 1	6	4273	240000	1026
				Mi. 9	6	15065	120000	1808
[95]	283	η	V-II Pro	7+1+1	16,4,8	22726	7308	166
				7+1+1	32,8,16	37803	4392	166
		η_T		7+1+1	16,4,8	33252	4368	145
Characteristic 3 Implementations from the Literature								
[103]	97	BKLS	V-II	18+1	1,4	33790	242526	8195
					4,4	46590	122610	5712
					4,8	50286	112480	5651
		DL	V-II Pro	18	4	55616	12900	717
[105]	97	DL	V-II Pro	1	4	4481	59946	269
	89	Kwon	V-II Pro	1	4	4481	64602	289
[106]	97	η_T (no exp)	V-II Pro	–	–	1888	32618	61
Implementations From This Thesis								
Ch. 4	313	η_T	V-II Pro	14	12	44060	4818	212
Ch. 5	103	η_T	V-II Pro	20	12	27182	5805	157
Ch. 6 c.2	271	η_T	V-II Pro	2	8	5170	30912	160
	271	η_T	V-II Pro	2	12	6365	25110	160
	313	η_T	V-II Pro	2	12	7120	31239	223
Ch. 6 c.3	97	η_T	V-II Pro	3	4	6690	20996	140
	97	η_T	V-II Pro	1	8	4979	28437	142

pares well with this, returning the same result in 25,110 cycles while requiring only 6,365 slices in the $\{\#M = 2, D = 12\}$ case. The implementations of Shu et al. have AC products of 166,000 slice.cycles and 145,000 slice.cycles. Each of their processors returns a result in a low number of clock cycles. An implementation employing 33,252 slices computes a Tate pairing in 4,368 cycles. It may, however, be difficult to substantially reduce the number of slices required by their architecture. Their processor has an ALU that contains an $\mathbb{F}_{2^{4m}}$ multiplication unit and an $\mathbb{F}_{2^m}$ multiplier. The $\mathbb{F}_{2^{4m}}$ multiplication unit contains six subfield multipliers. The ALU also contains an exponentiation subsystem that contains another two $\mathbb{F}_{2^m}$ multipliers. Shu et al. use seven multipliers with digit sizes of 16, one with a digit size of 4 and one with a digit size of 8 to return their best result. These selections result in a relatively high area footprint and, due to the fixed quantity of multipliers and the relatively large digit sizes required, it may be difficult to maintain efficiency while attempting to reduce area. The flexible processor can, when $\{\#M = 2, D = 8\}$, return a Tate result in 30,912 cycles using only 5,170 slices. This indicates that it may be more suited to environments in which a low area footprint is required.

Kerins et al. describe two distinct processors for Tate pairing computation on elliptic curves of characteristic 3 in [103]. The first implements the BKLS algorithm for Tate pairing computation discussed in [60]. The architecture contains a general purpose elliptic curve coprocessor and an $\mathbb{F}_{3^{6m}}$ multiplier. The coprocessor can be used to perform elliptic curve point multiplication or to perform the subfield operations required during iteration of the *for* loop of the pairing. The $\mathbb{F}_{3^{6m}}$ unit contains 18 $\mathbb{F}_{3^m}$ multipliers. The fastest and most efficient version of this processor has an AC product of 5,651,000 slice.cycles and returns a result in 112,480 cycles. A total of 50,286 slices are required. Although the number of clock cycles and slices required are high, an advantage of this architecture is that it can be used to perform the elliptic curve group operation and Tate pairing computation. Kerins et al. propose a second processor that could be used to implement the Duursma and Lee algorithm for Tate pairing computation. Their proposed architecture contains the $\mathbb{F}_{3^{6m}}$ multiplication unit, which contains 18 $\mathbb{F}_{3^m}$ multipliers and some subfield cubing, addition and negation circuitry. This processor is not implemented but, through calculation, the authors state that a total of 12,900 cycles would be required to return a pairing with the use of multipliers with digit sizes of 4. They suggest that such an architecture would require a high percentage of slice utilisation on a Virtex-II Pro FPGA containing 55,616 slices. This approximation returns an AC product of 717,000 slice.cycles. Factors such as the low digit sizes already in use and the fixed number of multipliers would, again, make it very difficult to reduce area requirement without a substantial increase in clock cycles. The most efficient characteristic 3 flexible processor architecture has an AC product of 140,000 slice.cycles. This requires only 6,690 slices and returns a result in 20,996 cycles. The processor of this chapter may, therefore, be more suited to constrained environments than both of the processor architectures described by Kerins et al.

Grabher and Page present a characteristic 3 Tate pairing processor in [105]. They use an $\mathbb{F}_{3^m}$ microprocessor architecture, implemented on an FPGA, connected to a general purpose serial processor through a Xilinx *microblaze core*. The FPGA contains storage registers and an ALU containing one multiplier and a number of other subfield units. The authors allocate 4,481 FPGA slices to their coprocessor. An implementation of the Duursma and Lee method returns a pairing in 59,946 cycles. An implementation of the Kwon algorithm [107] returns a result in 64,602 cycles. This system requires a small number of slices. An $\{\#M = 1, D = 8\}$ flexible processor can, however, return a pairing in a much lower 28,437 cycles with only a slightly higher area usage of 4,979 slices. The implementation of Grabher et al. is also not as versatile as the flexible processor as the quantity of subfield modules within the ALU is fixed.

Beuchat et al. implement an η_T pairing computation in [106]. This is an extension of their work described in [108] and [109]. A novel η_T pairing algorithm is proposed. The authors suggest that the characteristic 3 η_T pairing algorithm described by Barreto et al. in [63] can be improved upon. They demonstrate that some of the more expensive field operations required during loop iteration can be replaced with less costly operations by employing some pre- and post-processing techniques. They also develop a *unified arithmetic operator*. This contains an array multiplication architecture and some circuitry to enable resource sharing for addition, multiplication and cubing. This operator performs all required field arithmetic. An η_T pairing is returned in 32,618 cycles with an area usage of 1,888 slices. This is an excellent result, comparing very well with the characteristic 3 implementations of the flexible processor. The $\{\#M = 3, D = 4\}$ flexible processor returns a result in a lower 20,996 cycles but requires a larger area of 6,690 slices. Beuchat et al. do not, however, perform an exponentiation to a Tate pairing. During this research it was noted that the number of cycles required by the conversion is significant due to its serial nature and the large cost of the required extension field operations.

The flexible processor can also be compared with the fast characteristic 2 elliptic and genus 2 hyperelliptic curve processors of Chapters 4 and 5, respectively. The processor of Chapter 4 has an AC product of 212,000 slice.cycles, returns a Tate pairing in 4,818 cycles and requires 44,060 slices. The genus 2 processor has an AC product of 157,000 slice.cycles, returns a Tate pairing in 5,805 cycles and requires 27,182 slices. The most efficient $\mathbb{F}_{2^{313}}$ flexible processor is ($\{\#M = 2, D = 12\}$). This has an AC product of 223,000 slice.cycles. This implementation returns a pairing in a much larger 31,239 cycles but requires only 7,120 slices. It is difficult to substantially reduce the area of the processors of Chapters 4 and 5 due to the fixed quantity of multipliers. Again, the flexible processor is more suited to a low area environment in comparison to the other processors.

The results of this section indicate that pairing processors with units dedicated to extension field arithmetic can return a pairing in a very low number of clock cycles but are, due to their very nature, not suited to environments with stringent area requirements. The most efficient versions of the flexible processor are those that provide a low area implementation.

6.5 Conclusions

The implementation of the characteristic 3 elliptic curve Tate pairing, computed using the η_T methods, has been discussed in this chapter.

An overview of the underlying field and curve mathematics has been provided. The pairing algorithm and techniques for its efficient computation have been described. The hardware implementation of characteristic 3 arithmetic is not as straightforward as in the binary case: $2m$ bits are required to store an element of $\mathbb{F}_{3^m}$. Some intermediate mapping is also required. A useful representation for characteristic 3 elements in such an environment has been provided. The operations required to perform $\mathbb{F}_{3^m}$ and $\mathbb{F}_{3^{6m}}$ arithmetic have also been discussed with a hardware target in mind.

Initial exploration indicated that the runtime benefits of extension field units would not be offset by the resulting high cost resource requirements. A pairing processor with an ALU that contains subfield units operating in parallel was created instead. This ALU contains one module for addition, subtraction, cubing and inversion. The quantity of digit-serial multipliers and their digit sizes can be varied to return a result in a desired number of clock cycles or to satisfy a particular resource requirement. The ALU can be programmed to contain either $\mathbb{F}_{2^m}$ or $\mathbb{F}_{3^m}$ units, which means that it can be used to perform both characteristic 2 and 3 Tate pairings. Results are stored in dual port RAM. The control unit consists of a counter, a ROM unit and an FSM. An instruction sequence describing the subfield operations required to compute a pairing is flashed to the ROM. The FSM uses the counter to control access to the ROM. Operations are scheduled carefully: addition, subtraction and cubing are performed during multiplication, when possible. The implementation process does not have to be repeated if a different algorithm is to be implemented: the new set of instructions can simply be flashed to ROM.

A flexible *C++* software class has been written to reduce the effort required to produce desired processors and to define instruction sequences. The sequences required for $\mathbb{F}_{2^m}$, $\mathbb{F}_{2^{4m}}$, $\mathbb{F}_{3^m}$ and $\mathbb{F}_{3^{6m}}$ arithmetic and storage are defined in a software library. A user can then generate an instruction sequence using all of the capabilities and functionality of high-level software.

The Tate pairing has been computed on the supersingular elliptic curves $E(\mathbb{F}_{2^{271}})$, $E(\mathbb{F}_{3^{97}})$ and $E(\mathbb{F}_{2^{313}})$. Processors with various architectural parameters were implemented on a Virtex-II Pro FPGA. Results show that the flexible processor is excellently suited to systems in which area utilisation is a primary concern. A processor with a footprint of only 5,170 slices returns an $E(\mathbb{F}_{2^{271}})$ pairing in 30,912 clock cycles, while a characteristic 3 version that uses only 9,052 slices returns an $E(\mathbb{F}_{3^{97}})$ Tate pairing in 20,996 cycles.

The processor described in this chapter is an ideal platform on which to perform pairings when area is at a premium. There is a floor beyond which dedicated, highly parallel, processors cannot perform efficiently. The flexibility of the processor means that it is suited to embedded environments in which bandwidth may be costly. Algorithmic modifications can be performed by sending and remotely flashing an instruction sequence. This avoids the necessity for transferring a bit file, which is often significantly larger. The processor can also be programmed to perform both point scalar multiplication and Tate pairing computation, which is useful for many applications.

The subject matter of this chapter has been published in [11], [12] and [13].

Chapter 7

Modern Directions in Pairing-Based Cryptography: A Review

In this chapter, modern trends and developments in pairing-based cryptography are discussed. The computation and use of bilinear pairings for cryptographic applications has received much attention in the literature. A family of secure and efficiently computable pairings have been introduced. More attacks on pairing-based systems have been suggested and associated countermeasures devised. New optimisation techniques have been presented. There has been an ongoing investigation into the fast and efficient implementation of pairings in hardware. Pairings can now be computed very effectively in software due to advances in the processing power of general purpose sequential processors. The use of pairings in embedded systems in which area and energy are highly constrained has also been investigated. Proposals for novel and interesting pairing-based schemes have been made. The aim of this chapter is to provide an overview of these topics.

Some modern algorithms for pairing computation are outlined in Section 7.1. Security considerations for the use of pairings in cryptography are discussed in Section 7.2. An overview of side channel attacks, an area that has received much attention in recent years, is also provided. The software computation of pairings is discussed in Section 7.3. Hardware systems for pairing computation are described in 7.4. Some modern applications for pairings are discussed in Section 7.5. Suggestions for future work in the field are then provided in Section 7.6.

7.1 Pairings

An elliptic curve $E(\mathbb{F}_q)$ is known as pairing-friendly if, for pairing input points of order r, $r \geq \sqrt{q}$, where $r|\#E(\mathbb{F}_q)$ and $k \leq (log(r)/8)$. The former condition ensures that the DLP is sufficiently difficult while the latter enables computability.

In 2005, Barreto and Naehrig showed that pairing-friendly ordinary elliptic curves with embedding degree 12 can be constructed [110]. These curves are defined on the field $\mathbb{F}_p$, where p is a prime. They are known as BN or BN-p curves, where p is the value of the prime. The BN family have curve equation $E(\mathbb{F}_p) : y^2 = x^3 + b$, where $b \neq 0$. The number of rational points, denoted r, is also prime. The high embedding degree makes these curves very suitable to implementations at the 128-bit security level and above. The prime p, the group order r and the trace of Frobenius t are parameterised according to:

$$\begin{aligned} p(u) =& 36u^4 + 36u^3 + 24u^2 + 6u + 1 \\ r(u) =& 36u^4 + 36u^3 + 18u^2 + 6u + 1 \\ t(u) =& 6u^2 + 1 \end{aligned} \tag{7.1}$$

for some $u \in \mathbb{Z}$ that returns prime p and r.

In 2006, Hess et al. introduced the *Ate* pairing [111]. This simplifies the η_T pairing and extends the techniques to ordinary elliptic curves. The *Ate* pairing can be computed efficiently on elliptic curves that admit *twists*. Let E and E' be two elliptic curves defined on $\mathbb{F}_q$ and with embedded degree k. E' is a degree d twist of E if there exists an isomorphism $\psi_d : E' \to E$ defined over $\mathbb{F}_{p^d}$ and d is minimal. In *Ate* pairing computation one of the inputs is a point on $E(\mathbb{F}_p)$ while the other is a point on $E'(\mathbb{F}_{p^v})$, where $v = k/d$. Hess et al. show that at high security levels the *Ate* pairing can provide significant advantages over the Tate pairing if the curve has certain properties.

The r-*Ate* pairing was introduced by Lee et al. in 2009 [112]. This is a generalisation and an improvement of the *Ate* pairing techniques. A reduced Miller loop length can be used on some pairing-friendly elliptic curves. It is shown that the r-*Ate* pairing can be computed 50% faster than the *Ate* pairing on certain BN curves.

In 2010, Lubicz and Robert discussed the Tate and Weil pairings on general abelian varieties [113]. They show that, cryptographically, security is not restricted to the Jacobian of an algebraic curve. They describe efficient algorithms that use *theta* functions to perform pairing computation.

In 2010, Vercauteren presented *optimal pairings* [114]. A pairing is optimal if it can be computed in the theoretical minimum number of Miller iterations on the family of curves on which it is defined. An algorithm that can be used to construct optimal *Ate* pairings on all types of pairing-friendly curves is provided. Computation of an optimal *Ate* pairing is often written as a_{opt}.

Following these advances, much emphasis has been placed on the computation of the optimal *Ate* pairing on BN curves. The high embedding degree ($k = 12$) means that BN curves are an ideal candidate for pairing computation. For example, an *Ate* pairing that is computed on $p = 256$ returns 128-bit security while $p = 640$ returns 192-bit security. Another benefit is that there is a large quantity of curves within the family. Some of these curves have properties that can be exploited to return a fast pairing. As many of the implementations described later in this chapter compute the optimal *Ate* pairing on BN curves, the operations required to compute it are discussed briefly here.

BN curves admit a sextic twist, which means that $k/d = 2$. Let u be the value defining the parameters of a BN curve $E(F_p)$. Let $\psi : E' \to E$ be the isomorphism defining the twist and let π_p be the Frobenius automorphism on E such that $\pi_p(x, y) = (x^p, y^p)$. Let $z = (p^k - 1)/r = (p^{12} - 1)/r$. Consider the points $P \in E(\mathbb{F}_p)[r]$ and $Q \in E'(\mathbb{F}_{p^2})[r]$. The optimal *Ate* pairing is computed according to

$$a_{opt}(P, Q) = \Big((f_{s,Q}(P)).(g_{[s]Q,\pi_p(Q)}(P)).(g_{[s]Q+\pi_p(Q),-\pi^2(Q)}(P)) \Big)^{\frac{p^{12}-1}{r}} \tag{7.2}$$

where $f_{s,Q}$ is the Miller function such that $div(f_{s,Q}) = s(Q) - ([s]Q) - (s-1)(\infty)$ and s is the pairing loop length given by $s = 6u + 2$. A term $g_{Q_1,Q_2}(P) \in \mathbb{F}_{p^{12}}$ is the function of the line through Q_1 and Q_2, evaluated at P.

The operations required to compute the *Ate* pairing are listed in Algorithm 13. The value of $f_{s,Q}(P)$ is computed during the loop of Lines 7-14. The computations required for point doubling and addition on the twisted curve and for defining the line functions during these operations are performed on $\mathbb{F}_{p^2}$. The $E'(\mathbb{F}_{p^2})$ curve arithmetic can be performed in either affine or projective coordinates. Projective coordinates are usually used since inversions are not required during addition and doubling in this system. The line functions must, however, be evaluated at the point $P \in E(\mathbb{F}_p)$, which has projective coordinates. The overhead introduced by the incorporation of two different coordinate systems can be reduced using explicit formulae [115]. The evaluation of the line functions at P can also be performed in parallel with some of the operations required by the addition and doubling operations. The evaluation results are sparse members of $\mathbb{F}_{p^{12}}$. This sparseness reduces the number of extension field operations required during the accumulation step at the end of the iterative loop.

Good overviews of *Ate* pairing computation on BN curves are provided in [116] and [117].

Algorithm 13 Optimal *Ate* pairing computation on BN curves

INPUT: P, Q, s, where $P \in E(\mathbb{F}_p)[r]$, $Q \in E'(\mathbb{F}_{p^2})[r]$ and $s = 6u + 2 = \sum_{i=0}^{log_2(s)} s_i 2^i$

OUTPUT: $f = a_{opt}(P, Q)$, where $f \in \mathbb{F}_{p^{12}}$

Initialise: $d_1 \leftarrow 1$
$T \leftarrow [2]Q$ and evaluate $d_2 \leftarrow g_{Q,Q}(P)$
if $(s_{\lfloor log_2(s)-1 \rfloor} = 1)$ **then**
    $T \leftarrow T + Q$ and evaluate $d_1 \leftarrow g_{T,Q}(P)$
end if
$f \leftarrow d_1.d_2$
for $(i \leftarrow \lfloor log_2(s) \rfloor - 2,\ i \geq 0,\ i \leftarrow i - 1)$ **do**
    $T \leftarrow [2]T$ and evaluate $g_{T,T}(P)$
    $f \leftarrow f^2.g_{T,T}(P)$
    if $s_i = 1$ **then**
        $T \leftarrow T + Q$ and evaluate $g_{T,Q}(P)$
        $f \leftarrow f.g_{T,Q}(P)$
    end if
end for
$Q_1 \leftarrow \pi_p(Q)$
$Q_2 \leftarrow \pi^2(Q)$
if $u < 0$ **then**
    $T \leftarrow -T$
    $f \leftarrow f_{p^6}$
end if
$T \leftarrow T + Q_1$ and evaluate $d_1 \leftarrow g_{T,Q_1}(P)$
$T \leftarrow T - Q_2$ and evaluate $d_2 \leftarrow g_{T,-Q_2}(P)$
$f \leftarrow d_1.d_2$
$f \leftarrow f^{(p^{12}-1)/r}$

RETURN: f

7.2 Security

Computational attacks on finite field and curve-based cryptography until the end of 2007 have already been discussed in Subsections 2.6.1 and 3.4.1. In this section, the most notable modern computational attacks are discussed. Cryptographic systems that are implemented in insecure locations must be secured against physical attack. This applies to software processors, to constrained devices such as phones and smart cards and to hardware architectures. In recent years physical attacks on curve and pairing-based systems have received a lot of attention. Some of these attacks and their countermeasures are discussed in this section.

7.2.1 Computational Attacks

The best known computational attacks on the DLP, the ECDLP and the HCDLP until the end of 2007 were discussed in Subsection 2.6.1. Each of these problems are susceptible to brute force and generic attacks, such as Pollard's rho algorithm. In 2013, Joux published a paper, made available in preprint form, that startled the cryptographic community [118]. He provides a new algorithm to solve the DLP in fields of form $\mathbb{F}_{q^{2n}}$, where $n \leq q + d$ and d is some small integer. The algorithm has heuristic complexity $L(1/4 + o(1), c)$, for some value of c. Prior to this discovery, the best known attack on the DLP required sub-exponential time. Joux's method reduces this to heuristic polynomial time. Barbulescu et al. [119] then showed that the DLP can be solved in quasi-polynomial time for fields of type $\mathbb{F}_{q^{kn}}$, where $k \geq 2$, d is a very small integer and $n \leq q + d$. Subsequent publications concentrated on attacking fields with features that enabled the fastest solution to the DLP. It was not, however, clear whether the new solution methods applied to fields on which pairings are constructed. In 2014, Granger et al. detailed new techniques for solving the DLP on some binary fields on which pairing security relies [120]. They apply their techniques on two finite fields to demonstrate their efficacy. The first is $\mathbb{F}_{2^{4.1223}}$, a field arising from an elliptic curve implementation with embedding degree 4. They show that this field, previously believed to have a 128-bit security level, is in fact only 94.6-bit secure. They also show that the $\mathbb{F}_{2^{12 \cdot 367}}$ field, applicable to genus 2 pairing implementation, could be fully broken.

Due to these significant advances, the use of fields of small characteristic for pairing-based cryptography is currently not recommended. A good overview of the discrete logarithm problem and its history of attacks was published by Joux et al. in 2014 [121].

7.2.2 Side Channel Attacks

Side channel attacks exploit the leakage of physical information from a device while it implements a cryptographic computation. Resource constrained platforms, such as smart cards, are particularly vulnerable to such attacks due to the ease of physical access that an adversary may have. Some such attacks are passive, which means that they do not interfere with the operation of the device. In active attacks, an adversary physically tampers with the device, influences its internal operation and aims to gain some information from the unintended operations. Such attacks are usually called *fault attacks*. A fault can be injected by laser, by electromagnetic pulse or by introducing power variations and clock glitches [122]. Much of the literature relevant to pairing-based systems concentrates on attacks against identity-based protocols. The attacks assume that the pairing algorithm and its result are known to an adversary. Security relies on keeping one of the pairing input arguments secret. These assumptions are also made in this subsection. Excellent overviews of side channel attacks on devices implementing pairings are available in [123] and [124].

In 1996, Kocher et al. were the first to describe how side channel attacks can be used on cryptographic devices [125]. They discuss a passive attack that analyses the time taken by private key operations in RSA and Diffie-Hellman protocols in order to return the secret information on which the security of the protocols rely. In 1999, Kocher et al. showed how the power dissipated by a device can be be measured and used to efficiently compute confidential data [126]. *Simple Power Analysis* (SPA) and *Differential Power Analysis* (DPA) attacks are discussed. SPA involves the analysis of one power consumption trace in order to gain some information about the instructions performed during computation. The implementation of some operations may, for example, be conditional on key bits and intermediate variables. In DPA, the aim is to find a correlation between the data under manipulation and variations in power consumption. More than one power trace is required. Statistical functions are used to distinguish minor variations from background noise. It is shown how SPA and DPA can be used to retrieve secret information from a device implementing the Data Encryption Standard [127]. They state that susceptibility to SPA can be reduced by removing conditional branches from algorithms. This can, however, result in a performance penalty. Kocher et al. suggest various countermeasures against DPA. These include the balancing of state transitions, the physical shielding of the device, the deliberate addition of noise and the use of hash functions to render the information garnered from the leaked information useless.

In 2006, Page and Vercauteren described the first known fault attack on a system employing pairings [128]. The reduced Tate pairing $e(P, Q)$, computed using the Duursma and Lee methods, is considered. The point Q is publicly available while security relies on the secrecy of P. A valid pairing result is first collected by the adversary. The attacker then injects a fault that changes the number of iterations of the Miller loop. The resultant pairing value is also collected by the attacker. The secret P can be computed from the quotient of the two pairing results. The Kwon-BGOS algorithm [107] can be attacked in a similar manner. Page and Vercauteren suggest that *point blinding techniques* can be used to defend against such attacks. The attacks that they describe are only successful if the adversary has knowledge of one of the input points. Fortunately, the bilinearity of the pairing provides an efficient solution to this problem. If the operation $e(P, Q)$ must be performed then, for the random integers a and b, $e(a[P], b[Q]) = e(P, Q)^{ab}$ is instead computed. The exponent can be eliminated if values of a and b can be found such that $ab \mod l \equiv 1$. This countermeasure may, however, be expensive as two extra point scalar multiplications are needed for every pairing computation.

Whelan and Scott discussed passive DPA attacks on the Tate, η_T and *Ate* pairings in the same year [129]. They show that some hardware architectures implementing finite field arithmetic may be vulnerable to attacks that use statistical correlation. The type of expansion and accumulation employed in modular multiplication architectures means that it is possible to efficiently compute the input operands from substrings of the multiplicative result. They also show how a square root implementation can leak information about the operand. These weaknesses can be exploited to extract information about the pairing input coordinates. Whelan and Scott hypothesise that the Tate and *Ate* pairings may be resistant to such attacks if the secret point is the first input. The η_T pairing is, however, susceptible to the attacks, regardless of the order of its input arguments. A countermeasure, in which intermediate results can be multiplied by random subfield elements, can be used to close this avenue of attack [130]. The final exponentiation eliminates this randomisation. Point blinding can also be employed, but with its inherent extra cost. Whelan and Scott considered a fault attack on the Weil, Tate and η_T pairings in 2007 [131]. They assume that an adversary is able to change the sign of one of the input coordinates by interfering with a single bit. The quotient of the corrupted and correct pairing results can be used to gain some secret information. The optimal time at which to inject the attack is on the final iteration of the Miller loop. The computation of the secret is more difficult, but remains possible, in other cases. This attack is possible on the η_T and Weil pairings. Whelan and Scott claim that the Tate pairing is not vulnerable to this type of attack due to its more complex final exponentiation.

In 2009, DPA attacks on the Tate, η_T and *Ate* pairings, when computed using the Jacobian coordinate system, were presented by El Mrabet et al. [132]. A practical system, created to demonstrate the feasibility of the attack, is presented. An 8-bit architecture is synthesised and tested in a simulation environment. It is shown that the restriction of the secret value to the first pairing input does not provide adequate security. A DPA of one modular subtraction and one modular multiplication is required for attack. Point blinding does, however, provide protection. In the same year, El Mrabet extended her previous work and discussed the vulnerability of pairing implementations that use both affine and projective coordinates [133]. Miller's algorithm is attacked by modifying the number of loop iterations. This is a generalisation of the fault attack previously described by Page and Vercauteren [128] and can be performed on the Tate, η_T, *Ate* and optimal pairings. A fault is injected into the counter that controls loop iteration. The aim is to gain access to intermediate values produced by two successive iterations of the loop. Pairing computations are corrupted until this is achieved. The retrieved values can be used to extract secret information. A DPA of a modular addition and a modular multiplication is required to perform this attack. As in the previous case, point blinding provides protection against this attack.

In 2013, Blomer et al. discussed a DPA attack that exploits the modular addition operation required during pairing computation. The attack requires analysis of either one modular addition or one modular multiplication, but not both. A point blinding countermeasure relevant to these attacks that is slighlty more efficient than that of Page and Vercauteren [128] is also provided. In the same year, Bae et al. described an attack on Miller's algorithm in which a fault is injected into the instruction sequence that controls the operations to be performed [134]. The fault removes the point addition performed during the final iteration of the Miller loop by skipping the instruction defining the last *if* statement. Only one successful attack of this type is required to enable the efficient computation of the secret. The corrupted Miller output can be compared to the correct result to retrieve the secret information. To investigate the practicality of their attack, Bae et al. perform a laser fault injection on a microchip implementing a simple test algorithm. It should be noted that this attack assumes that the Miller results can be accessed before the final exponentiation. Lashermes et al. presented a fault attack on the final exponentiation in the same year [135]. This attack is applicable to pairings with complex exponentiations that were previously thought to protect them from SCA. In [136], it was suggested that the final exponentiation on pairing-friendly ordinary curves can be computed efficiently by decomposing it into three distinct parts. Lashermes et al. show that exponentiation using this technique can be reversed by injecting three different faults into the final exponentiation. These must be introduced on three separate implementations of the same pairing.

7.3 Software Computation of Pairings

Since the beginning of 2008, significant strives have been made in the area of software pairing computation. Prime field pairing computation is, in particular, very suited to software implementation as field arithmetic can be performed very efficiently using integer, floating point and other native operations. The computation of both small characteristic and prime characteristic pairings on general purpose sequential processors is discussed in this section. Note that in the case of small characteristic implementations, it is assumed that the DLP security breach described in Subsection 7.2.1 is not available. It would not be possible to compare the implementations in terms of their assumed security otherwise.

Pairings can also be computed on highly constrained devices, such as smart cards and microprocessor systems. Implementation on these devices will be discussed in Section 7.5.

The most notable literature contributions to the software computation of pairings on small characteristic curves are described in this section. The most significant results of each of the publications that are discussed are listed in Table 7.1.

In 2008, Hankerson et al. [137] discussed the software computation of pairings on small characteristic and prime field elliptic curves at the 128-bit security level. This is a detailed and interesting paper that focuses on the implementation of small characteristic and large characteristic pairings on general purpose processors using various available features. The strengths and weaknesses of the 32-bit Intel Pentium 4, the 64-bit AMD Opteron and the 64-bit Intel Core 2 architectures are discussed. The most attractive results are returned by a C code implementation on the Intel Core 2 processor. In the prime field case, a pairing is performed on a BN curve of 256 bits. Field elements are stored as integers, which enables the efficient use of a relatively fast 64-bit multiplier for modular arithmetic. An implementation of a 128-bit secure η_T pairing, computed on $E(\mathbb{F}_{2^{1223}})$, is also performed. In the Intel Core 2 case, Single Instruction, Multiple Data (SIMD) supplementary instruction sets are employed. These instruction sets are used to decompose an operation into

Table 7.1: Notable modern contributions to the software computation of pairings. An *(a)* after a BN curve definition indicates that curve arithmetic is performed using the affine coordinate system. All other BN curve implementations are performed using projective coordinates.

Ref.	**Year**	**Sec.** (Bits)	**Curve**	**Alg.**	**Device**	**F.** (GHz)	**Cycles** ($\times 10^6$)	**Time** (ms)
Computation on Curves of Small Characteristic								
[137]	2008	128	$E(\mathbb{F}_{2^{1223}})$	η_T	Intel C2	2.4	39	16.25
		128	$E(\mathbb{F}_{3^{509}})$	η_T	Intel C2	2.4	33	13.75
[138]	2010	128	$E(\mathbb{F}_{2^{1223}})$	η_T	Intel X	2	17.4	8.7
		128			Intel X (x8)	2	3.02	1.51
Computation on Curves of Large Characteristic								
[137]	2008	128	BN-256	*Ate*	Intel C2	2.4	15	6.25
				r-*Ate*	Intel C2	2.4	10	4.17
[139]	2008	128	BN-256	a_{opt}	Intel C2D	2.4	28.45	11.85
					Intel C2D (x2)	2.4	14.43	6.01
[115]	2010	128	BN-257	a_{opt}	Intel C2D	2.83	4.38	1.54
[116]	2010	128	BN-254	a_{opt}	Intel Ci7	2.8	2.33	0.83
[140]	2011	128	BN-254	a_{opt}	AMD PII	3	1.57	0.52
[141]	2012		BN-254(a)	a_{opt}	Intel C2D	2.4	14.21	5.92
		164	BN-446(a)				44.3	18.46
		192	BN-638(a)				136.5	56.88
[142]	2012	128	BN-254	a_{opt}	ARM C-A9	1.2	11.89	9.91
		164	BN-446	a_{opt}		1.2	47.46	39.55
		192	BN-638	a_{opt}		1.2	119.23	99.36
[143]	2015	128	BN-254	a_{opt}	ARM C-A15	1.7	7.89	4.64
		128	BN-254		ARM C-A15(N)	1.7	6.09	3.58
		164	BN-446		ARM C-A15(N)	1.7	30.07	17.69
		192	BN-638		ARM C-A15(N)	1.7	79.84	46.97

sub operations that can be performed in parallel. On the Core 2 processor, the finite field members can be packed into groups of 128-bit registers using SIMD Within a Register

(SWAR). The Streaming SIMD Extensions (SSE) platform is then used to perform 128-bit operations when possible. This is called a *vectorised* system. The finite field elements are called vectors, while a string contained in one of the registers is known as a scalar. A 128-bit secure characteristic 3 pairing, computed on $E(\mathbb{F}_{3^{509}})$, is also performed. This implementation also uses SIMD, SWAR and SSE. Characteristic 3 addition is written in assembly language to ensure that the operation is compiled efficiently. As expected, the prime field pairings provide the best results due to the availability of the 64-bit multiplier and the high embedding degree of BN curves. The fastest 128-bit secure r-*Ate* pairing result is returned in 4.17 ms. Interestingly, this is returned by a non-SSE implementation. The 128-bit secure characteristic 2 and 3 η_T pairings are returned in 16.25 ms and 13.75 ms, respectively. A significant contribution of this paper is the assertion that a faster prime field pairing result is returned by a non-SSE instruction set: the organisational and recombination operations that must be employed are not warranted as efficient use can be made of the 64-bit multiplier by using an integer representation.

In 2008, Grabher et al. [139] investigated techniques for the software computation of the optimal *Ate* pairing. The main focus is on the exploitation of available parallelism using SIMD, SWAR and SSE features and of the sharing of operations across two processor cores such as the Intel Core 2 Duo. The problems inherent in the use of SWAR storage in elliptic curve systems are discussed. Finite field arithmetic often requires operations on scalars that occupy different positions within their respective vectors. This may require a large amount of decomposition and recombination, which can be relatively expensive. Fortunately, the SSE instruction set can efficiently recode a SWAR register to a group of registers that are of a smaller size. Consider, for example, a 128-bit finite field element. This can be stored in one 128-bit register or four 32-bit registers. The widths of the individual registers are known as *digit sizes*. Grabher et al. show that arithmetic can be performed efficiently by separating the 128-bit elements into four 32-bit strings. Each string is stored as the first scalar of a 128-bit vector. The other three registers in each of the vectors remain empty. This means that 128-bit operations can be performed in terms of 32-bit strings that are aligned within their respective vectors. This can significantly reduce carry cost. For evaluation purposes, pairing computations are performed using

SIMD and non-SIMD instruction sets at the 128-bit security level. In the former case, a single core must be used and an optimal *Ate* pairing is returned in a minimum of 11.85 ms. Interestingly, a single core implementation using a non-SIMD instruction set requires only 9.72 ms. This is due to the natural 64-bit datapath of the Core 2 Duo. If the security level must be increased, however, the larger operands may mean that the SIMD implementation will become more attractive. The fastest computation overall is returned by two cores, operating in parallel, using non-SIMD instruction sets. The *OpenMP* multiprocessing API is used to perform $\mathbb{F}_{p^{12}}$ arithmetic in terms of parallel $\mathbb{F}_{p^6}$ operations and $\mathbb{F}_{p^2}$ operations in terms of parallel $\mathbb{F}_p$ operations. This system provides an optimal *Ate* result in 14.43 MCycles, which corresponds to 6.01 ms. These results support the assertion of Hankerson et al. [137] that non-SSE instruction sets can be efficiently employed for prime field pairing computation. A single core 128-bit r-*Ate* computation is performed in 10 Mcycles in their case, while the implementation performed by Grabher et al. requires 44% more clock cycles, despite the use of twice as many cores and the more efficient optimal *Ate* pairing construction.

In 2010, Aranha et al. [138] described techniques for the software computation of the reduced characteristic 2 η_T pairing at the 128-bit security level. While the 2008 publication of Hankerson et al. [137] endeavoured to clarify the cost differences between small characteristic and prime field characteristic pairings on various processors in a general sense, the purpose of this paper is to perform a high-speed pairing (of small characteristic) using all available means. The η_T pairing is implemented on a dual quad-core Intel Xeon processor. The use of SIMD vector instruction sets for characteristic 2 finite field arithmetic is explored. Detailed algorithms, written in terms of the SSE instructions required to implement them, are provided for squaring, multiplication, routing and inversion on $\mathbb{F}_{2^m}$. The use of these algorithms means that inefficiencies that may be introduced by poor compilation of higher level code are avoided. A useful load balancing technique for multi core platforms is also presented. A parallel version of the Miller loop of the η_T algorithm, suitable for multi core implementation, is also developed. A relatively inexpensive precomputation reduces dependencies between different iterations of the Miller loop. Distinct iterations are then performed in parallel on separate cores. A recombination step accumulates the intermediate functions by extension field multiplication. A pairing computation is performed on $E(\mathbb{F}_{2^{1223}})$ using one core and eight cores of the Intel Xeon device. The former computation requires 8.7 ms (17.4 Mcycles), while the latter requires 1.51 ms (3.02 Mcycles). The 5.76x speed up demonstrates that the use of multi core processors is desirable if the extra cost is affordable. A clock cycle reduction of 55% is achieved in the one core case in comparison to the 2008 implementation of Hankerson et al. This reduction can largely be explained by the direct SSE definition of the field operations required for pairing computation.

In the same year, Naehrig et al. [115] discussed the computation of the optimal *Ate* pairing on BN curves. The parameters of the BN curve polynomials (Equation (7.1)) are carefully selected so that the number of point doubling and addition steps required during the Miller loop is minimised. It is also shown that some of the line functions required by the original definition of the optimal *Ate* pairing are not required and a more efficient algorithm is provided. Elements of $\mathbb{F}_p$ are represented in a polynomial form that dramatically reduces the number of operations required for multiplication and subsequent modular reduction. Double-precision floating-point arithmetic is employed (contrary to Hankerson et al. [137], whereby large integer arithmetic is performed by decomposition to integers and the use of a 64-bit multiplier). Naehrig et al. store the 12 coefficients of each $\mathbb{F}_p$ element in consecutive places of a double array of size 12. Members of $\mathbb{F}_{p^2}$ are interleaved in double arrays of size 24. This representation enables the widespread and efficient utilisation of SIMD operations for parallel extension field arithmetic. Techniques for avoiding overflow are also discussed. A 128-bit secure optimal *Ate* pairing is implemented on one core of an Intel Core 2 Duo processor. A result is returned in 1.54 ms (4.38 MCycles). This is a clock cycle reduction of 56% in comparison to the cycles required by the implementation of Hankerson et al. It should, however, be noted that the r-Ate pairing is performed in the latter case. There is also a restriction on the number of curves on which the optimisations of Naehrig et al. apply. A publically available and useful library for software computation using their methods is available at [144].

In 2010, Beuchet et al. [116] presented a useful software library for the computation of the optimal *Ate* pairing on BN curves at the 126-bit security level. The pairing is computed using a slightly modified version of the algorithm provided by Naehrig et al. [115]. The main difference is that Beuchat et al. perform both point addition and subtraction during Miller loop iteration. The $\mathbb{F}_{p^{12}}$ field is constructed as a quadratic extension of $\mathbb{F}_{p^6}$, which is itself a cubic extension of $\mathbb{F}_{p^2}$. This is a similar construction method to that employed in the genus 2 case of Chapter 5. The techniques for extension field arithmetic, previously described by Hankerson et al. [137], are also employed here. The value of u defining the BN curve parameters (Equation(7.1)) is selected in order to provide a low hamming weight Miller loop with sparse arithmetic (in a similar manner to Naehrig et al. [115]). Beuchat et al. do, however, choose a value of u that also speeds up the final exponentiation, which is performed using the three step approach of Scott et al. [136]. The computation of the optimal *Ate* pairing on software architectures that are more modern than those used by the publications already discussed in this section ([137], [139], [138] and [115]) is then described. An unsigned multiplication can be performed in only three clock cycles on a processor such as an Intel Core i7, a significant reduction in cost compared to the other processors. Field arithmetic techniques that take advantage of the smaller clock cycle gap between additive and multiplicative operations are discussed. Algorithms, written in assembly code, for the implementation of arithmetic on $\mathbb{F}_p$ and its extensions are provided. The software library is prototyped on an Intel Core i7 processor. A 126-bit optimal *Ate* pairing is returned in 0.83 ms (2.33 MCycles). This is a significant improvement on the 4.38 MCycles required by the optimal *Ate* pairing of Naehrig et al.. This is, however, partly due to the newer architecture and more efficient instruction set, which means that a meaningful comparison is difficult to make. The main contributions of this publication are the algorithms that are explicitly created with the small clock cycle gap between multiplication and addition in mind. The source code of the library is publically available [145].

In 2011, Aranha et al. [140] discussed the fast computation of the optimal *Ate* pairing. The tower construction of Beuchat et al. [116] is employed and it is shown how operations on the sub and extension fields can be further optimised. In general, an $\mathbb{F}_{p^k}$ Karatsuba multiplication requires $k(k+1)/2$ reductions modulo p. It is shown that the prime field lazy reduction technique, described in [93] and [146], for fields of type $\mathbb{F}_{p^2}$, can be generalised to $\mathbb{F}_{p^k}$. The use of lazy reduction reduces the number of required modular reductions to k. When arithmetic must be performed on the extensions, it is also possible to delay the reductions required by the associated sub field arithmetic until the results must be combined. The use of software words that are larger than the data that they hold is also suggested. The unassigned bits within the words means that a combination of single- and double-precision arithmetic can be employed. The optimal balance between the two types of arithmetic is extensively explored in the case of multiplication on the sub and extension fields. Miller's algorithm itself is analysed and it is shown that some of the $\mathbb{F}_p$ reductions required after the curve line evaluations can be delayed and merged with other reductions that are performed during the later $\mathbb{F}_{p^2}$ multiplications. It is also shown that the final exponentiation can be performed without inversion when the parameter defining the BN curve is negative. The three stage exponentiation of Scott et al. [136] is employed and it is shown that inversion on $\mathbb{F}_{p^{12}}$ can be avoided when BN curve parameter u (from Equation (7.1)) has a negative value. The optimal *Ate* pairing is implemented on various software platforms. In a similar manner to Beuchat et al. [116], algorithms for arithmetic on $\mathbb{F}_{p^2}$ are written in assembly while higher-level algorithms are written in *C* code. The fastest 128-bit secure optimal *Ate* pairing is returned on an AMD Phenom II processor in 0.52 ms. The corresponding 1.57 MCycles is significantly lower than the 2.33 MCycles required by the implementation of Beuchat et al.. It must be noted, however, that the techniques discussed in their publication form the basis for some of the more significant optimisations presented in the 2011 publication of Aranha et al. [140]

In 2012, Acar et al. [141] implemented the optimal *Ate* pairing on three ARM platforms at several security levels. The main aim is to compare the use of affine and projective coordinates for curve arithmetic during pairing evaluation in a quantitive manner. The implementations employ many of the computational techniques described by Lauter et al. [147]. A scalar multiplication on an elliptic curve using affine coordinates requires inversion in the field on which it is defined. Lauter et al. use two previously known techniques to reduce inversion cost. Inversion is moved from the extension field to the base field using the optimal towering methods of Baktir and Sunar [148]. Schroeppel and Beaver [149] show that the number of inversions required during elliptic curve scalar multiplication can be reduced by performing all point doublings first. The point additions can then be computed together and the required inversions shared. This reduces the cost difference between field inversion and multiplication and thus between affine and projective curve arithmetic. Acar et al. compute separate pairings, each using either projective or affine curve arithmetic. In the former case, computations are performed using the explicit formulae of Costello et al. [150]. Optimal *Ate* pairings are implemented on x86, x86-64 and ARM devices. Acar et al. state that optimisations that take advantage of the features of specific processors or instruction sets are not employed. This means that a fair comparison can be made across the three platforms and portability to other devices is not compromised. Computation is performed on BN-254, BN-446 and BN-638 curves, which provide security levels of 128 bits, 164 bits and 192 bits, respectively. Results from implementations on an Intel Core 2 Duo processor are discussed here. These are representative of the main results of Acer et al. Projective coordinate computations return optimal *Ate* pairings in 6.31 ms and 19.05 ms at the 128-bit and 164-bit security levels, respectively. The corresponding affine pairings are computed in 5.92 ms and 18.46 ms, a reduction of 6.2% and 3%, respectively. There is a more significant difference at the 192-bit security level. A pairing using projective curve arithmetic is computed in 65.95 ms, while an affine arithmetic pairing is computed in 56.88 ms. This is a much more significant reduction in computation time (13.8%), than in the lower security cases. This leads Acar et al. to hypothesise that as security requirements grow, the use of affine curve arithmetic will become increasingly attractive.

In the same year, Grewal et al. [142] analysed the use of affine and projective curve arithmetic for the optimal *Ate* pairing on ARM architectures. New optimisation techniques for tower field and curve arithmetic are presented. It is shown that the lazy reduction method, previously discussed by Aranha et al. [140] in the context of projective curve arithmetic and extension field multiplication, can be used to perform affine curve arithmetic and extension field inversion. Algorithms, employed during the multiplications of the Miller loop, exploit the sparse nature of the extension field elements. Two types of sextic twists that are available on BN curves are also discussed: D-type and M-type. The former is usually used for pairing computation as the untwisting operation is the more trivial of the two. However, if an M-type twist is used and the pairing is computed on the twisted curve itself, then untwisting can be performed at a very low cost. The two types of twist evaluate a pairing in a similar time. This increases the number of curves that are suited to efficient pairing computation. Techniques for efficient curve arithmetic in affine and projective coordinate systems are also discussed. The methods involve precomputation and a careful selection of various field operations in order to take advantage of the features of ARM processors. Optimal *Ate* pairings are computed on 254-bit, 446-bit and 638-bit BN curves, returning security levels of 128, 164 and 192 bits, respectively. Implementation is performed on a Cortex-A9 processor. Custom ARM assembly code for field addition and multiplication is employed in the 254-bit BN curve case. Techniques include loop unrolling to remove conditional statements, instruction reordering to maximise pipelining and the avoidance of memory fetching by employing a maximal number of registers. Without assembly optimisation, 254-bit secure optimal *Ate* affine and projective arithmetic pairings are returned in 11.84 ms and 11.24 ms, respectively. The use of assembly code reduces the affine and projective computation times to 10.57 ms and 9.91 ms, respectively. The projective computation times are 5% faster in the non-assembly case and 6% faster in the assembly case. Contrary to the results of Acer et al. [141], there is a negligible difference in computation time between affine and projective arithmetic at the two higher security levels. Grewal et al. state that theirs is a fairer comparison as the optimisations in field and curve arithmetic provide a more level playing field. This claim is agreed with here.

In 2015, Azarderakhsh et al. [143] discussed the implementation of the optimal *Ate* pairing on several x86-64 PC and ARM processors. This publication is essentially an implementation of the optimisations discussed by Grewal et al. [142] in 2012 on more modern platforms that have features that benefit pairing computation. Optimised ARM assembly code for field multiplication is employed in BN 254-bit and BN-446 cases. Optimised addition is also performed in the former case. The 128-bit NEON SIMD engine, included in some modern ARM processors, is also utilised to accelerate computation. Implementations on a Cortex-A15 demonstrate a considerable acceleration in pairing computation when the NEON engine is used. In the BN-254 case, which provides 128-bit security, the employment of NEON reduces pairing computation from 4.64 ms to 3.58 ms, a 23% decrease. In the affine arithmetic case, computation time is reduced from 4.86 ms to 3.85 ms, a decrease of 21%. On the BN-446 curve, affine and projective curve pairings are returned in 18.04 ms and 17.69 ms (a 1.9% difference) while on BN-638, results are returned in 47.79 ms and 46.97 ms (a 1.7% difference), respectively. This further shows that an optimal *Ate* computation using projective coordinates is significantly more attractive in the BN-254 case (there is a 7% difference), while the gap between the two coordinate systems becomes relatively insignificant as security increases.

7.4 Hardware Implementation of Pairings

Modern advances in the hardware implementation of pairings are summarised in this section. The more notable literature contributions to the field are described. Some of these methods are discussed in terms of their progression from the ideas, concepts and architectural strategies that are presented in this thesis. It is, however, often difficult to directly compare hardware implementations in terms of area, speed and AT product as the various pairing algorithms are computed on devices with different features and at several security levels. The aim of this section is to give an overview of the main contributions of each of the discussed publications and to perform fair comparisons when possible.

Small characteristic implementations are discussed in Subsection 7.4.1, while prime characteristic implementations are discussed in Subsection 7.4.2

7.4.1 Hardware Implementation of Pairings on Curves of Small Characteristic

The most notable modern publications describing the hardware computation of pairings on curves of small characteristic are presented in Table 7.2. The AES security levels listed assume that the attack on small characteristic implementations of the DLP (described in Subsection 7.2.1) is not available as it would not otherwise be possible to compare implementations.

Table 7.2: Modern contributions to the computation of pairings in hardware on curves of small characteristic. Area is measured in either slices (Sl) or gates (G). Note that E represents an elliptic curve while C represents a genus 2 hyperelliptic curve.

Ref.	Year	**Sec.** (Bits)	**Curve**	**Alg.**	**Device**	**Area**	**F.** (MHz)	**Cycles** ($\times 10^3$)	**Time** (μs)
[151]	2009	< 60	$E(\mathbb{F}_{2^{163}})$	BKLS	ASIC	27430 G	10.3	515	50,000
						28155 G	5.44	272	50,000
[152]	2009	68	$E(\mathbb{F}_{2^{251}})$	BKLS	ASIC	329088 G	50	75.15	1503
[153]	2010	66	$E(\mathbb{F}_{3^{97}})$	η_T	V-II Pro	10897 Sl	147	4.85	33
					V-II Pro	10262 Sl	142	9.09	64
					ASIC	193765 G	200	9.34	46.7
[154]	2010	128	$E(\mathbb{F}_{3^{95.5}})$	η_T	V-4	4755 Sl	192	427.6	2227
[155]	2011	97	$E(\mathbb{F}_{3^{239}})$	η_T	V-4	66631 Sl	179	2.06	11.5
		96	$E(\mathbb{F}_{2^{557}})$			55156 Sl	149	1.97	13.2
[156]	2011	128	$E(\mathbb{F}_{2^{1223}})$	η_T	V-4	35458 Sl	168	48048	286
					V-6	15167 Sl	250	47500	190
[157]	2012	128	$C(\mathbb{F}_{2^{367}})$	η_T	V-4	4518 Sl	220	773.96	3518
[158]	2012	128	$E(\mathbb{F}_{2^{1223}})$	η_T	V-6	16403 Sl	267	27.23	102
					ASIC	524286 G	500	27.3	54.6
[80]	2013	128	$E(\mathbb{F}_{2^{1223}})$	η_T	V-6	16402 Sl	180	57.6	320

In 2009, Van Herrewege et al. [151] discussed the computation of pairings in constrained environments. ASIC simulations of the BKLS algorithm on a supersingular curve $E(\mathbb{F}_{2^{163}})$ are described. The focus of the work is on energy efficiency and not speed. A simple unit for $\mathbb{F}_{2^m}$ addition forms the basis for all field operations. Multiplication is achieved by operand shifting and by performing accumulation through the addition unit. An architecture containing a memory unit (consisting of registers), a control unit and an $\mathbb{F}_{2^m}$ arithmetic core implements Miller's algorithm. More than one $\mathbb{F}_{2^m}$ addition unit can be included in the arithmetic core if a higher multiplicative throughput is required. The top level architecture is very similar to that of the flexible processor of Chapter 6. Two ASIC simulations, each with a pairing computation time of 50 ms, are performed in order to investigate power consumption and area utilisation. The first employs one addition unit. A clock frequency of 10.3 MHz is required to return a pairing in the desired time. This processor has a dynamic power consumption of 98.3 µW. Two addition units are used in the second simulation. A clock frequency of 5.44 MHz is required. The dynamic power consumption in this case is 48.5 µW, which is approximately half that of the implementation using one addition unit. The areas required are 27,430 and 28,155 gates, respectively. The increase of only 725 gates is more than justified by the reduction in power consumption. Van Herrewege et al. state that their results may be improved upon by investigating whether the footprint of the FSM can be reduced and by further maximising the efficiency of register usage within the system. Later publications that concentrate on low power consumption (such as those discussed in Section 7.5) implement the BN-256 optimal *Ate* pairing on 8-, 16- and 32-bit microprocessors. It would be useful to implement the same pairing using the architecture of Van Herrewege et al. as there is a greater level of architectural flexibility available and, therefore, design freedom available. It may also, perhaps, be interesting to simulate low-area ASIC implementations of the flexible processor of Chapter 6 at low frequencies in order to investigate power consumption and speed trade-offs.

In 2009, English et al. [152] implemented a characteristic 2 Tate pairing on an ASIC. The primary design concern is power consumption. A 65 nm CMOS standard cell implementation of the *macro* Tate pairing processor described by Keller et al. [8] is performed. Keller et al. implement the BKLS algorithm on an FPGA using several $\mathbb{F}_{2^{4m}}$ arithmetic units. The author of this thesis was involved in the work required to design that processor. The $\mathbb{F}_{2^{4m}}$ inversion technique, previously discussed in Subsection 4.4.5, forms the basis for the extension field inverter used. A description of this inverter was co-published in [7]. The methods used by English et al. to reduce the power consumption of the ASIC implementation include the isolation of unused functional units from the data bus, intermediate operand storage to reduce unnecessary switching and careful clock gating. A processor performing a Tate pairing computation on $E(\mathbb{F}_{2^{251}})$ requires 329,088 gates. A result is returned in 1.5 ms at a frequency of 50 MHz. This implementation consumes 2.63 mW of dynamic power and 1.23 mW of static power. The use of Network-on-Chip (NoC) Interconnect for ASIC implementations of the *macro* and *micro* pairing processors of Keller et al. was later explored by English et al. in 2011 [159]. The architectures have costly interconnect requirements, in large part due to the 251-bit cross chip data buses. NoCs can be used to replace these wide buses with a network of routers and short pipelined links. Network interface modules are used to provide a bridge between the serialised data and the 251-bit inputs and outputs of the arithmetic units. A result of an arithmetic operation is, for example, serialised and packetised before it is sent for storage. Results show that the inclusion of an integrated circuit-switched NoC to the *micro* architecture results in relatively large increases in computation time, area and energy although the total power is reduced. A NoC point-to-point interconnect system for the *micro* architecture is also described. In this case, the $\mathbb{F}_{2^m}$ multiplier and divider are provided with point-to-point links to the system (there is no general system NoC). Each of the two units has three links dedicated to the serial transmission of its operands and output. This facilitates the removal of the large, expensive NoC switch and interfaces. Four variants of the *micro* architecture, each with a different link width, are implemented. Results show that point-to-point interconnect performs significantly better than the integrated switched NoC. The implementations are also compared to the 2009 ASIC implementation. The throughput and area results are similar but a 64-bit point-to-point interconnect produces a 70% reduction in top level wirelength and a switching power decrease of 96%. There is also a reduction in congestion, which usually has a positive effect on yield and manufacturability. A custom NoC topology for the *macro* processor is also discussed. A bidirectional pipelined ring with packetised data circulates the system. The units use point-to-point links to connect to the ring. However, this configuration incurs significant penalties in terms of computation time and energy. The use of this NoC does, however, result in a 70% decrease in switching power and a 75% reduction in wirelength. English et al. provide a very detailed analysis of the utilisation of various methods for NoC interconnect in the context of the implementations of Keller et al. It would be interesting to extend this analysis to implementations of the η_T and *Ate* pairings although the effort involved may be considerable. The analysis of the *micro* architecture may, in particular, be very useful as the flexible architecture can be be easily modified to perform other elliptic curve operations.

Beuchat et al. [153] discussed the implementation of the characteristic 3 elliptic curve η_T pairing in 2010. Two hardware architectures, each implementing an $\mathbb{F}_{3^{97}}$ pairing, are discussed. Both processors are implemented on FPGA, while an ASIC implementation of one of the architectures is also described. A modified version of the η_T pairing algorithm, originally described by Beuchat et al. in [160], is used for computation. This algorithm does not require cube rooting and thus eliminates the requirement for its associated circuitry. The first processor contains an ALU that is carefully designed for the fast computation of the Miller loop. The most costly operation is the sparse $\mathbb{F}_{3^{6m}}$ multiplication required to accumulate the Miller variable. Operands are fed through three parallel units that each contain three $\mathbb{F}_{3^m}$ multipliers and some registers and multiplexers. The parallel outputs are fed into another unit that contains another three $\mathbb{F}_{3^m}$ multipliers and a different configuration of registers and multiplexers. These units are embedded in an ALU that contains combinatorial logic performing the other operations that are required on each iteration of the loop. The processors of Chapters 4 and 5 aim to reduce the time required by the Miller loop by performing the major operations of consecutive iterations of the loop in parallel. Beuchat et al. use a different approach: they use bespoke circuitry that accelerates the completion of each iteration. This results in a very fast datapath for each iteration of the loop. Beuchat et al. also discuss a second processor, which they state is very similar to the flexible processor of Chapter 6. While this is true, the number of multiplication units is not variable in their system. The ALU contains nine $\mathbb{F}_{3^m}$ multipliers and one unit for the $\mathbb{F}_{3^m}$ addition, subtraction, cubing and accumulation operations. The first processor is implemented on a Virtex-II Pro FPGA. Using a multiplier digit size of 3, an $\mathbb{F}_{3^{97}}$ η_T result is returned in 33 µs at a frequency of 147 MHz. A total of 10,897 slices are utilised. At that point in time, this implementation returned the fastest pairing computation in the literature. The techniques used to create the high throughput ALU had a significant influence on later publications. As will be seen throughout this section, several other processors have been created containing ALUs that aim to accelerate separate iterations of the Miller loop in a similar manner for various characteristics, security levels and pairing algorithms. The second processor returns a result in 64 µs at 142 MHz and requires

10,262 slices. A 180 nm ASIC implementation of this processor is also presented. This device contains 193,765 gates and has a power consumption of 671.74 mW at 200 MHz. A reduced η_T pairing is returned in a very impressive 46.7 µs.

In 2010, Estibals [154] discussed the resource-constrained hardware implementation of a 128-bit secure pairing on characteristic 3 supersingular curves. The curves that are used are defined on composite extension fields $\mathbb{F}_{q^n}$, where $q = 3^m$ and n is a small integer. This choice is motivated by the availability of many arithmetic optimisations on small characteristic fields and on small characteristic supersingular curves. The vulnerability of composite extension field curves to several attacks is considered. The time taken by all known attacks on all possible supersingular curves with extension degrees that are large enough to make 128-bit security a possibility are computed. The (assumed) secure supersingular curve $E(\mathbb{F}_{q^n}) : y^2 = x^3 - x - 1$, where $q = 3^{97}$ and n = 5, is used. A pairing computation on this curve provides 128-bit AES security. The η_T algorithm of Beuchat et al. [160] is used to perform the pairing. The algorithm must, in this case, be computed in terms of operations on $\mathbb{F}_{q^n}$. The composite extension degree means that the arithmetic can be performed efficiently in terms of $\mathbb{F}_q$ operations using a tower field construction. A compact processor implements the $\mathbb{F}_q$ computations required to return the pairing. This is the processor that Beuchat et al. use to perform the final exponentiation of the characteristic 3 η_T pairing in [161]. It contains dual port RAM, a unit that performs $\mathbb{F}_q$ additions and Frobenius operations, and an $\mathbb{F}_q$ digit-serial multiplier with $D = 14$. Estibals discusses various extension field multiplication techniques in terms of their suitability to this architecture. It is found that an algorithm described by Cenk and Özbudak [162] that is based on the Chinese Remainder Theorem is the most efficient. A Virtex-4 implementation of the 128-bit secure η_T pairing returns a result in 2.23 ms. The area utilisation is very low, at 4,755 slices. The main contribution of this paper is to show that the compact implementation of pairings on supersingular curves of composite extension degree is viable at relatively high security levels. Estibals states that a characteristic 2 implementation may provide even more attractive results due to the binary nature of hardware. Since only one multiplier is used, the architecture may also become even more attractive at higher security levels as one would expect that an extra multiplier could be efficiently utilised to maintain a practical computation time.

In 2011, Beuchat et al. [155] discussed the fast computation of characteristic 2 and 3 elliptic curve η_T pairings. This is an expansion of the implementation strategy previously described by Beuchat et al. in [161], which was used to implement a characteristic 3 η_T pairing in the previous year [153]. A custom, pipelined, Karatsuba multiplier is presented. The input operands are first split into upper and lower components in the usual manner and the first stage of a Karatsuba multiplication performed. The more efficient of either the Karatsuba method or the schoolbook method for polynomial multiplication is used to perform the next multiplicative stage. This process continues until a full result is returned. Registers can be inserted between stages and the depth of the pipeline adjusted according to the complexity of the operations to be performed. The most efficient configuration for the computation of a characteristic 3 η_T pairing is that with seven pipeline stages. The characteristic 3 η_T processor also contains multiplexers, registers and combinatorial logic to handle the irregular datapath to and from the multiplier. On each iteration of the Miller loop, the sparse $\mathbb{F}_{3^{6m}}$ multiplication and the computation of the coefficients for the sparse multiplication of the next iteration are computed in parallel. A scheduling system that begins an $\mathbb{F}_{3^m}$ multiplication at each clock cycle is used. The characteristic 2 η_T pairing processor contains an $\mathbb{F}_{2^{4m}}$ multiplier with five pipeline stages. It has a different configuration of multiplexers, registers and combinatorial logic to deal with the irregular datapath. A supplementary processor, presented by Beuchat et al. in [161], is used to perform the final exponentiation of each pairing. This processor has a very similar architecture to the flexible processor of Chapter 6. A 97-bit secure characteristic 3 η_T pairing, computed on $E(\mathbb{F}_{3^{239}})$, is implemented on a Xilinx Virtex-4 and returns a result in 11.5 µs. The processor has a footprint of 66,631 slices. A 96-bit secure characteristic 2 η_T pairing, computed on $E(\mathbb{F}_{2^{557}})$, is performed in 13.2 µs and occupies 55,156 slices. This publication is a culmination of the small characteristic design efforts that Beuchat et al. had presented in the years previous to 2012 (as already discussed in this section). Their work is an excellent resource if familiarisation with optimisation techniques and architectural strategies for the hardware implementation of small characteristic pairings is desired.

In 2011, Ghosh et al. [156] discussed the implementation of a 128-bit secure η_T pairing. Computation is performed on the elliptic curve $E(\mathbb{F}_{2^{1223}})$. An efficient unit for 1223-bit multiplication forms the basis for the η_T processor. An $\mathbb{F}_{2^{1223}}$ multiplication is first decomposed into three 612-bit multiplications using the Karatsuba method. Each 612-bit multiplication is further decomposed into three 306-bit multiplications. A module that performs 306-bit Karatsuba multiplication in only one clock cycle by continuously decomposing the 306-bit operands is discussed. The nine 306-bit multiplications that are required for 1223-bit multiplication are performed serially and the results recombined using combinatorial logic. The 306-bit multiplication module and its associated recomposition logic is embedded in a 1223-bit multiplication unit containing shift registers and multipexers. This unit is carefully designed so that input operands are immediately available to the multiplier when required. A full 1223-bit multiplication is returned in 10 clock cycles. The adopted strategy is very similar to that utilised to perform $\mathbb{F}_{2^{12m}}$ multiplication in the genus 2 processor of Chapter 5. In that case, the $\mathbb{F}_{2^{12m}}$ multiplication was first decomposed into three $\mathbb{F}_{2^{6m}}$ multiplications. An $\mathbb{F}_{2^{6m}}$ multiplication architecture containing three degree 2 Karatsuba multipliers, each returning a multiplication result in m/D clock cycles, was then created. The architecture of Chapter 5 requires $m/D + 8$ clock cycles to return an $\mathbb{F}_{2^{12m}}$ multiplication result (if fully parallel $\mathbb{F}_{2^m}$ multipliers were used, this could be reduced to nine clock cycles). The shift register strategy used by Ghosh et al. could be useful in that context although implementation may be more difficult due to its dual mode nature (it must also perform the $cmul(\alpha, \beta)$ routine of Algorithm 10). Ghosh et al. describe an η_T processor that is tailored to the efficient utilisation of the 1223-bit multiplication unit. There is a common datapath for the computation of the non-reduced η_T pairing and for the final exponentiation. A system of registers, multiplexers and $\mathbb{F}_{2^m}$ combinatorial logic units provide the 1223-bit multiplication unit with the required inputs. Control circuitry is used to select between operands for Miller computation and exponentiation. Each iteration of the Miller loop is performed in two steps: the evaluation of the intermediate function and accumulation by sparse $\mathbb{F}_{2^{4m}}$ multiplication. The former computation requires only 12 clock cycles, while the latter requires 61 clock cycles. An efficient method for performing the final exponentiation is also discussed. This requires a total of 98 multiplications on $\mathbb{F}_{2^{1223}}$ and some combinatorial operations. The processor is created in an exemplary fashion: Ghosh et al. endeavour not to waste even the smallest number of clock cycles, in particular during the Miller loop. This is reflected by very fast computation times. A 128-bit secure implementation on a Virtex-4 FPGA requires 35,458 slices and returns a reduced η_T pairing in 286 µs. A Virtex-6 implementation requires 15,167 slices and produces a result in 190 µs. Both implementations require approximately the same number of clock cycles: 47,500 in the former case and 48,048 in the latter case. The difference in computation time can be explained by a higher achievable clock frequency on the Virtex-6 device. The area discrepancy is due to the fact that each slice of a Virtex-6 contains four 6-input LUTs, while a Virtex-4 slice contains only two 4-input LUTs.

In 2012, Aranha et al. [157] presented new techniques for the computation of pairings on supersingular characteristic 2 genus 2 hyperelliptic curves. The embedding degree of 12 is a significant factor in the ability to perform an efficient 128-bit secure pairing. The optimal pairing technique of Vercauteren [114] is extended to the computation of the η_T pairing. Aranha et al. call their technique an *optimal η_T pairing* computation. Two Miller evaluations are required to compute their pairing. The first has a loop length of $(m-1)/2$ iterations, whilst the second is of length $(m+1)/2$. This provides a 33% saving over the original η_T computation, which requires $(3m+1)/2$ iterations of the Miller loop. Detailed costs associated with the optimal genus 2 η_T pairing computation are provided in terms of operations on $\mathbb{F}_{2^m}$. It would, however, have been helpful if the operations required for an η_T computation using their field and curve construction were provided. The pairing is implemented in both software and hardware. A 128-bit optimal η_T computation using degenerate divisors (on the genus 2 curve $C(\mathbb{F}_{2^{367}})$) returns a result in 0.98 ms (2.44 MCycles) when implemented on an Intel Core i5. For the purposes of comparison, a regular η_T result is returned in 2.7 ms (6.86 MCycles). The optimal genus 2 pairing is also computed using general divisors on a Xilinx Virtex-4 FPGA. The exponentiation coprocessor of Beuchat et al. [155] is used. A result (using a multiplier digit size of 16) is returned in 3.52 ms with a corresponding area utilisation of 4,518 slices. The same architecture was previously used by Estibals [154] to compute a 128-bit secure composite curve pairing (using a digit size of 14) in only 2.23 ms with a similar area requirement of 4,755 slices. Aranha et al. do, however, claim that computation using degenerate divisors would provide a 2x to 4x acceleration on the same platform.

In 2012, Adikhari et al. [158] described a hardware architecture for the characteristic 2 elliptic curve η_T pairing at the 128-bit security level. Computation is performed on $E(\mathbb{F}_{2^{1223}})$, the same curve that Ghosh et al. [156] used in their implementation of the previous year. Adikhari et al. do not, however, use Karatsuba methods for field multiplication. Instead, they use a *Toeplitz Matrix Vector Product* (TMVP) approach [83]. An $m \times m$ Toeplitz matrix contains $\mathbb{F}_2$ elements $s_{k,i}$, where $i \geq 0$, $k \leq m-1$ and satisfies the property that $s_{k,i} = s_{k-1,i-1}$ for $i \geq 1$ and $k \leq m-1$. Fan and Hasan [163] show that a Toeplitz matrix can be used to implement field multiplication faster than the Karatsuba method while maintaining subquadratic space complexity. The matrix used during regular matrix-vector polynomial multiplication is not of Toeplitz structure, but a conversion can be achieved by performing some $\mathbb{F}_2$ addition, by rearranging elements and by padding when necessary. The use of the Toeplitz matrix means that a polynomial multiplication can be performed by recursively splitting the product matrices and performing them in parallel if desired. An $\mathbb{F}_{2^m}$ multiplication can, for example, be performed by computing one TMVP multiplication of size m, three TMVP multiplications of size $m/2$, or six TMVP multiplications of size $m/3$. A multiplication unit that uses a three-way split is first discussed. It contains a fully parallel 408-bit TMVP multiplier that returns a result in one clock cycle. The six TMVP multiplications are performed sequentially. A multiplication unit that utilises two recursions of the two-way split is also discussed. This performs 1224-bit multiplication by computing nine 306-bit TMVPs sequentially. Both units are implemented on a Virtex-6 FPGA. The multiplier of Ghosh et al. requires 10 cycles per multiplication and occupies 30,148 LUTs. The two-way split multiplier returns a result in nine clock cycles and occupies only 19,721 LUTs. This is clearly a more desirable implementation in terms of both computational speed and area. The three-way split multiplier produces an overall result in six clock cycles but requires 33,546 LUTs. The processor of Adikhari et al. consists of three main blocks: a binary arithmetic unit, an input and squaring unit and a data handling unit. The first block contains some registers, the $\mathbb{F}_{2^{1223}}$ multiplication unit and $\mathbb{F}_{2^{1223}}$ addition and squaring units. The second block contains combinatorial logic for the squaring and square rooting of points during the Miller loop. It also contains registers that store the results. The third block contains rewiring circuitry and registers that handle the inputs to the first block and that store the intermediate evaluations during Miller loop implementation. The final exponentiation is also performed using these blocks. The processor is synthesised on a Virtex-6 FPGA and on 65nm CMOS ASIC technology. A Virtex-6 implementation that uses a two-way split multiplier returns a 128-bit secure η_T pairing in 148 µs and occupies 13,596 slices. A three-way split implementation computes a pairing in 102 µs and uses 16,403 slices. The latter implementation is the more attractive of the two due to its lower AT product. These results are a significant improvement on those of Ghosh et al., who return a 128-bit secure η_T pairing in 190 µs on a Virtex-6. An ASIC synthesis using the two-way split multiplier returns a result in 80.64 µs and has a gate count of 472,777. A three-way split implementation produces a value in 54.62 µs and has a gate count of 524,286. The TMVP multiplication method does seem to provide an improvement over the Karatsuba method as Adikhari et al. adopt a similar approach to Ghosh et al. in designing their processor. Synthesis results on FPGA are, however, highly optimistic in terms of achievable frequency and area utilisation: the complicated mapping and place and route steps have not yet been performed. FPGA implementation results or, at the very least, the post place and route metrics provided by the vendor tool (as used by Ghosh et al.) are required to verify and reliably quantify the results.

In 2013, Cuevas-Farfán et al. [80] discussed the design and implementation of a processor for characteristic 2 elliptic curve η_T pairing computation. The processor supports on-the-fly changes in the elliptic curve, the tower field and the distortion map. This is also the case for the flexible processor of Chapter 6: the ROM sequence can be defined in terms of these properties and flashed to the FPGA without recompilation. Cuevas-Farfán et al. first define a 16-bit instruction set for $\mathbb{F}_{2^m}$ arithmetic, loop control and jumps. The instructions are then sequenced according to the operations required by the pairing computation. Significant effort may, however, be required to complete the instruction sequence generation process as an automated system for sequence generation (as presented in Chapter 6) is not described. Registers are used for storage of $\mathbb{F}_{2^m}$ values. An ALU contains units for $\mathbb{F}_{2^m}$ addition, multiplication, squaring and square rooting. The controller contains a ROM on which the sequence of operations is stored. Two computations are implemented on the curve $E(\mathbb{F}_{2^{1223}})$, which returns a 128-bit security level. The first uses the η_T algorithm discussed by Barreto et al. in [63]. Cuevas-Farfán et al. state that the second computation is performed using the curve, the tower field representation and the field properties discussed in Chapter 4 of this thesis (and published in [6]). The final exponentiation is also computed using the operations discussed in that chapter. An implementation of the algorithms and field constructions of Chapter 4 returns an η_T pairing in 787 µs and requires 50,968 slices when implemented on a Virtex-4 FPGA. A Virtex-6 implementation requires 320 µs, with an area utilisation of 16,402 slices. A fair comparison can be made with the 128-bit secure characteristic 2 elliptic curve η_T implementations of Ghosh et al. [156] and Adikari et al. [158] as they have similar area requirements on Virtex-6 devices. These publications were previously discussed in this section. The processor described by Ghosh et al. computes a pairing in 190 µs and requires 15,167 slices, whilst the processor of Adikari et al. computes a pairing in 102 µs, requiring 16,403 slices (although the latter are synthesis results). Both processors contain large units that are custom built for pairing computation: they do not consist of an ALU containing distinct units for $\mathbb{F}_{2^m}$ arithmetic alone. This is a useful demonstration of the assertion made in Chapter 6 that processors with custom architectures are a better option for fast pairing computation only when a relatively large implementation footprint is available. It would be useful to implement the dedicated processors of Chapters 4 and 5 on the more modern Virtex-6 devices at the 128-bit security level. As discussed previously, it is relatively trivial to scale those processors to higher security levels due to the design strategies used. A comparison between the dedicated characteristic 2 elliptic and genus 2 processors at the 128-bit security level would, in particular, be very interesting as the underlying field size increases at a smaller rate in the genus 2 case.

In 2015, Chung et al. [164] discussed the fabrication of an ASIC test chip for computation of the reduced η_T pairing on $E(\mathbb{F}_{3^{97}})$. The algorithm of Beuchat et al. [160], which does not require cube root computation, is used to compute the η_T pairing. Digit-serial multipliers are employed for field multiplication. The sparse $\mathbb{F}_{3^{6m}}$ multiplication of the Miller loop is not performed using either Karatsuba multiplication or Lagrange interpolation. As discussed in Section 6.3.3, Gorla et al. [104] use the latter method to compute a sparse extension field multiplication with a cost of 15 multiplications and 90 additions/subtractions on $\mathbb{F}_{3^m}$. Chung et al. state that the use of this method would result in the requirement for a large adder with multiple inputs and an irregular datapath. Contrary to previous published implementations, they endeavour to reduce the number of combinatorial operations at the expense of multiplications in order to simplify the datapath of their processor. By rearranging the formulae, a sparse $\mathbb{F}_{3^{6m}}$ multiplication is performed in 17 multiplications, 10 cubings and 35 additions, all on $\mathbb{F}_{3^m}$. The reduction in the number of required combinatorial operations means that the architecture that surrounds their $\mathbb{F}_{3^m}$ multiplier can be simplified. Exponentiation is performed using a *torus* representation [165], the Lagrange methods of Gorla et al. [104] and Frobenius operations. In total, an exponentiation to the reduced η_T pairing requires 79 multiplications, 390 cubings and 180 additions, all on $\mathbb{F}_{3^m}$. Chung et al. compare these quantities with those required during the exponentiation to the reduced η_T pairing of Chapter 6 (as published in [11]), which requires 231 multiplications, 304 cubings and 1,321 additions. This is a significant improvement, and it would be useful to use this method to perform exponentiation on that processor (although a further exponentiation to the Tate pairing is still required). The pairing accelerator contains separate coprocessors for the computation of the Miller loop and the final exponentiation. The Miller coprocessor contains an $\mathbb{F}_{3^m}$ multiplier of digit size 7, a cubing unit and a 4-input addition unit. A Miller loop iteration is performed in 17 cycles, resulting in a total count of $17 \times (97 + 1)/2 = 833$ cycles for η_T computation. Chung et al. aim to perform the final exponentiation in the same number of cycles. For this reason, the exponentiation coprocessor contains three digit-serial multipliers of digit size 7, a cubing unit and an addition unit. The cost of Miller computation and exponentiation is balanced: the exponentiation of a particular pairing is performed while the Miller loop of the next pairing is implemented. Timing results are reported by averaging computation times over a large number of pairing computations. While this is sound practice, this means that these results cannot be compared to other published implementations in a fair manner as other results are listed in terms of the time taken by a single Miller loop computation, serially followed by an exponentiation. This is why the results of Chung et al. are not included in Table 7.2. Nevertheless, it is useful to list the results. A processor computing a reduced η_T pairing on $E(\mathbb{F}_{3^{97}})$ is fabricated in a 90nm CMOS process. A total of 336,000 gates and two memory blocks of size $0.102mm^2$ are used. The total chip area is $1.47mm^2$. The average power consumption is 78.36 mW at 175 MHz. A pipelined pairing result is returned in 4.76 µs.

7.4.2 Hardware Implementation of Pairings on Curves of Prime Characteristic

The most significant literature contributions to the hardware implementation of prime characteristic pairings are discussed in this subsection. The results returned by these implementations are listed in Table 7.3. Note that all implementations provide roughly the same level of security (126-128 bits). Note also that an effort is made to summarise the most interesting results reported by each publication: the publications themselves should be examined in order to view all reported results.

Table 7.3: Modern contributions to the computation of pairings in hardware on curves of prime characteristic. Area is measured in either slices (Sl) or gates (G). The term *DSP* refers to the dedicated Digital Signal Processing units available in some modern FPGAs.

Ref.	**Year**	**Curve**	**Alg.**	**Dev.**	**Area**	**F.** (MHz)	**Cycles** ($\times 10^3$)	**Time** (μs)
[166]	2008	$E(\mathbb{F}_{512})$	BKLS	V-II	33857 Sl	135	217.35	1610
[167]	2009	BN-256	*Ate*	ASIC	164000 G	338	7706	22800
			a_{opt}				5340	15800
[168]	2009	BN-256	*Ate*	ASIC	183000 G	204	861	4220
			r-*Ate*				594	2910
[79]	2012	BN-256	*Ate*	V-6	4014 Sl+42DSP	210	336.37	1600
			a_{opt}			210	245.43	1170
[169]	2012	BN-258	a_{opt}	V-6	5237 Sl+64DSP	230	82.34	358
[170]	2013	BN-256	BKLS	V-6	23000 Sl	145	173	11930
			Ate				120.6	8320
			a_{opt}				82.1	5660
[171]	2014	BN-256	a_{opt}	ASIC	116000 G	200	608	3040

In 2008, Barenghi et al. [166] discussed the FPGA implementation of the Tate pairing on a prime field elliptic curve with an embedding degree of 2. The BKLS algorithm is used to compute the pairing. The final exponentiation is implemented using the Lucas laddering technique [172] and by exploiting the unitary norm of the non-reduced pairing value. A field size of 512 bits is considered, returning 128-bit security. Modular multiplication is performed using the Montgomery method [173]. Each operand is first converted to the Montgomery domain through multiplication by $2^{2s+2} \mod p$, where s is the size of the field in bits. Multiplication in the Montgomery domain does not require division by the modulus, which can be a relatively expensive operation. Conversion of the result from the Montgomery domain is performed through Montgomery multiplication by 1. This technique is most efficient when the initial and final conversions do not need to be performed before and after each operation. For this reason, the pairing input coordinates are moved to the Montgomery domain before computation begins and all arithmetic operations performed there. Multiplication is implemented using the dedicated 18×18 ASIC multipliers that are available on Virtex-II FPGAs. A single hardware component performs both modular addition and subtraction. Subtraction is performed by converting the second operand to a two's compliment representation. Three carry-look ahead 512-bit adders perform modular reduction. A software tool has been created to investigate the scheduling of operations through different combinations of the arithmetic units using a *Direct Acyclic Graph* (DAG). VHDL is automatically generated when a particular configuration is selected. A control system employs an FSM. The architecture of the processor is very similar to that of the flexible processor of Chapter 6. The chosen configuration contains one addition unit and four multiplication units. Each unit receives two 512-bit inputs from memory. Tristate buffers are used to select one output result for write. A Virtex-II implementation requiring 33,857 slices returns a Tate pairing on the curve $E(\mathbb{F}_{512})$ in 1.61 ms. Since the software and processor architecture are very similar to those of Chapter 6, it would be very interesting to explore various scheduling options using a DAG in order to investigate whether a more efficient implementation of the flexible processor can be found.

In 2009, Kammler et al. [167] explored the use of an Application Specific Instruction Set Processor (ASIP) for pairing computation. Computation is performed on a Reduced Instruction Set Computing (RISC) core (simulated using a design tool) that is augmented with a dedicated $\mathbb{F}_p$ hardware unit. The core is a 32-bit five stage pipelined system. While the core itself contains a 32-bit integer multiplier, its use is not convenient due to the large word widths required by pairing-based applications. A scalable hardware

Montgomery multiplier consisting of an array of carry save multipliers is instead designed and included in the $\mathbb{F}_p$ unit. The height of the array defines the critical path delay and is set so that it has the same value as that of the RISC core. The width can be varied according to performance and area requirements. A modular addition/subtraction unit of variable width is also included in the $\mathbb{F}_p$ hardware unit. All arithmetic operations are performed in the Montgomery domain. An application specific instruction set containing $\mathbb{F}_p$ instructions is employed. Operations on extension fields are converted to sequences of these instructions using software, a similar approach to that used in Chapter 6. A hardware Memory Access Unit (MAU) is used to facilitate high data throughput throughout the system. This unit extends the number of ports that can be used to access the RISC core and organises storage into distinct blocks from which values can be accessed in parallel. Various configurations of the system are synthesised using a 130 nm CMOS standard cell library. *Ate*, optimal *Ate* and η_T pairings are computed at the 128-bit security level. Results returned by various multiplication array sizes, addition/subtraction widths and memory sizes are provided. The fastest optimal *Ate* computation is performed by a 128×8 multiplier and a 32-bit wide modular addition unit. A result is returned in 15.8 ms on a system with a footprint of 164,000 gates. The strategy employed by Kammler et al. is attractive for use in embedded systems: a relatively inexpensive microprocessor could be used for storage and control, while an $\mathbb{F}_p$ arithmetic unit can be designed and connected to accelerate arithmetic operations alone.

In 2009, Fan et al. [168] showed that if particular BN curve parameters are selected then modular multiplication on $\mathbb{F}_p$ can be performed significantly faster than in the general case. A BN-256 optimal *Ate* pairing implementation is discussed. On BN curves, the prime p is characterised by the polynomial $p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$ (from Equation (7.1)). A modular multiplication technique is described that exploits the relationship $p^{(-1)}(u) \equiv 1 \mod u$. The complexity of the required operations is minimised when u is a pseudo-Marsenne number of form $u = 2^l + s$ for any suitable $l \in \mathbb{Z}$ and for some integer s that should be minimised. An algorithm is provided that performs fast $\mathbb{F}_p$ modular multiplication when these properties are satisfied. This is implemented using a unit that they call a *Hybrid Montgomery Multiplier* (HMM). The HMM contains one 32×16, four

64×16 multipliers and units for modular reduction, accumulation and recombination. The selection of the curve parameters means that multiplication on $\mathbb{F}_{256}$ can be performed in a total of one 32×32, eight 32×64 and 16 64×64 multiplications are required. A total of 13 $\log_2(s) \times \log_2(\mu)$ multiplications are also required, where μ is an integer such that $\mu < 2^{k+6}$ and k is the embedding degree of the curve. The final 13 multiplications can be performed quickly using shift and add operations when s is small. In comparison, regular Montgomery multiplication on $\mathbb{F}_{256}$ is much more expensive, requiring 36 64×64 multiplications. The top level architecture is similar to the flexible processor described in Chapter 6. It has an ALU containing an adder and the HMM, 64-bit RAM and a microcontroller that accesses instructions within a ROM. The system is synthesised using a 130 nm standard cell library. An implementation containing 183,000 gates returns a BN-256 *Ate* pairing in 4.22 ms and an r-*Ate* pairing in 2.91 ms. This is a speed up of 5.4x over the *Ate* result reported by Kammler et al. [167] in the same year, with an area increase of only 11.56%. The subset of curves that can be used for pairing computation using the methods of Fan et al. is, however, very restricted. This may prove problematic if their attributes are successfully attacked in the future.

In 2012, Fan et al. [79] presented an efficient hardware architecture for the computation of the *Ate* and optimal *Ate* pairings on BN-256 curves with certain properties. The computation techniques are similar to those previously discussed by Fan et al. in 2009 [168]. Fast modular multiplication is performed when the parameter defining the BN curve is a pseudo-Mersenne number. In the previous paper, a HMM is described that performs interleaved multiplication and reduction. In this paper, the multiplication and reduction steps are separated. A new HMM is presented that performs $\mathbb{F}_p$ multiplication on fields of up to 260 bits in four phases. In Phase 1, one 65×32 and four 65×65 multipliers perform polynomial multiplication. A one-round reduction of each of the partial products is performed in Phase 2. The outputs of this step are always less than 77 bits in size. Phase 3 sees the accumulation of the partial products followed by one polynomial reduction. The output is a 93-bit polynomial, which is separated and reduced in Phase 4 to return the final result. The top-level architecture of [168], now containing the new HMM, is reused. A *C++* program schedules the arithmetic operations and converts them to microcode that is stored in an instruction ROM. *Ate* and optimal *Ate* pairings, defined on suitable 256-bit BN curves, are computed on a Virtex-6 FPGA. Use is made of the dedicated Digital Signal Processing (DSP) units available on these FPGAs. A total of 4,014 slices and 42 DSPs are used. 128-bit secure *Ate* and optimal *Ate* pairings are returned in 1.6 ms and 1.17 ms, respectively. This is a dramatic improvement on their previous 183,000 gate ASIC implementation, which exhibited computation times of 4.22 ms in the former case and 2.91 ms in the latter case. As is the case with the previous paper, the small number of curves that are suitable for these methods may prove restrictive.

In 2012, Yao et al. [169] discussed optimal *Ate* pairing computation using *Residue Number Systems* (RNSs). This is a continuation and extension of the work described by Cheung et al. [174] in the previous year. In an RNS, a set of smaller integers are used to represent a large integer. An RNS *base* contains n coprime integer constants, each of which is called an RNS *modulus*. Large integer arithmetic can be performed by operating on each of the moduli in parallel: there are no interdependencies. This facilitates wide scale parallelism. Modular reduction by the prime p on which the finite field is defined is not performed in the RNS setting. Instead, Yao et al. employ Montgomery reduction. The number of required reductions is minimised using the lazy reduction technique [140]. A hardware processor is presented that can perform optimal *Ate* pairings on finite field sizes of 260 bits or less. It contains a micro-coded sequencer and an ALU containing four parallel *rowers*. Each rower performs an operation on a particular modulus and contains four 69×18-bit signed multipliers and some addition units for reduction and recombination of the results. Extra addition and accumulation units are also included so that $\mathbb{F}_{p^2}$ operations can be performed efficiently using the rowers. The processor is implemented on a Xilinx Virtex-6 FPGA containing dedicated DSPs. The 25×18 multipliers within the DSPs are employed. A total of 64 DSPs are consumed. A processor computing a 126-bit secure optimal *Ate* pairing returns a result in 358 µs and occupies 5,237 slices. This is an extremely fast pairing computation at this security level. This level of acceleration is, in the main, due to the custom architectures that are designed to efficiently employ the RNS and lazy reduction techniques. The availability of dedicated DSPs on modern FPGAs also means that the area requirements of pairing processors can be significantly reduced as the units required for multiplication are usually the most expensive in terms of area.

In 2013, Ghosh et al. [170] discussed pairing implementation on a Virtex-6 FPGA. The vulnerability of the implementation to side channel attacks is considered and some countermeasures discussed. $\mathbb{F}_p$ multiplication is performed using a binary double-and-add algorithm. The result at the end of each iteration of the algorithm is maintained below the number of bits in p so that a final division can be avoided. The logic blocks of Virtex-6 FPGAs are organised into individual rows of 16 slices (or 32 LUTs) that are connected by a Fast Carry Chain (FCC). This means that two 32-bit operands can be added extremely quickly. An addition module that performs fast 256-bit addition using these carry chains is described. A unified arithmetic unit for addition, multiplication and subtraction is presented. The unit contains three of the 256-bit addition modules and a module for $\mathbb{F}_p$ doubling (performed with some simple rewiring). Several smaller units are included to control data flow and storage. The Montgomery ladder technique [175] is used to compute the double-and-add Blakley algorithm for modular multiplication [176]. A result is returned by the arithmetic unit in 256 clock cycles. A custom Configurable Arithmetic Unit (CAU) architecture is discussed. Each CAU contains three unified arithmetic units. A CAU has two modes: it can be used to perform three parallel, independent, $\mathbb{F}_p$ arithmetic operations or a single multiplication on $\mathbb{F}_{p^2}$. Two CAUs, operating in parallel, are utilised. A dedicated unit handles the complex data access requirements introduced by the parallel use of the CAUs. A processor computing 128-bit Tate (using the BKLS algorithm), *Ate* and optimal *Ate* pairings is implemented and occupies 23,000 slices on a Virtex-6 FPGA. Results are returned in 11.93 ms, 8.32 ms and 5.66 ms, respectively. DPA attacks on the implementation are also considered. The attack model involves an adversary collecting a set of random public points and analysing power traces in order to retrieve details about the Hamming weight of the intermediate data. Very small power variations can be used to retrieve the values of particular bits during iteration. Ghosh et al. successfully mount this attack on the FPGA. To counter this attack, they suggest that operations that are performed on both public and private information at the same time should be removed. Such operations are performed during computation of the line functions of Miller's algorithm but can be modified to counter the attack. This does, however, result in a significant increase in computation time. Using the countermeasure, pairings are returned in 182 ms, 108 ms and 32 ms for the Tate, Ate and optimal *Ate* pairings, respectively. Results are not returned as quickly as the processors described by Fan et al. in [168] and [79]. This implementation is, however, not restricted to a subset of BN curves. The architecture of [169] utilises the DSPs that are available on Virtex-6 FPGAs and returns a pairing in a much faster 358 µs. It would be interesting to investigate whether the use of DSPs could improve the results returned by Ghosh et al.

In 2014, Chang et al. [171] discussed the energy-efficient hardware implementation of prime characteristic pairings on BN curves. The top level architecture of their system is similar to that of a general purpose sequential processor. It contains a data cache, an instruction cache, an ALU for prime field computation, a control system and a decoder. While some other publications discussed in this chapter use scheduling techniques to accelerate pairing computation, Chang et al. concentrate on maximising resource utilisation. The ALU contains Arithmetic Units (AUs) that can perform only addition and subtraction, These are called AAUs. It also contains full AUs, or FAUs, that can perform addition, subtraction and Montgomery multiplication. The AUs are connected using a register file. During each computation cycle, an idle AU searches for an instruction for which the input operands are available. If there is enough space in working memory, the AU will perform this operation. A more stringent limit is placed on the memory available to the AAUs in comparison to the FAUs so that multiplicative operations will have higher priority when they can be performed. A scheduler and compiler are used to generate an optimised schedule for each of the AUs and to ensure that memory units are efficiently employed. Through simulation, Chang et al. find that the use of one FAU and two AAUs delivers the most desirable performance. The pairing processor is hand-coded in the Verilog language and synthesised on a TSMC 90nm technology. A version that is not hand-coded is implemented at 130nm. A 200 MHz, 128-bit BN curve optimal *Ate* pairing is returned by the 90nm implementation in 3.04 ms. A total of 116,000 gates are required. Chang et al. define their energy metric in terms of an $Area \times Time \times Cycle$ (ATC) product. A value of 353.6 is reported. The 130nm implementation returns a result in 5.88 ms, has a footprint of 166,000 gates and has an ATC product of 976.2. The only direct comparison that can be made is to that of the 130nm results published by Kammler et al. in 2009 [167]. They return an optimal *Ate* pairing in 15.8 ms on a 338 MHz implementation that has a footprint of 164,000 gates. This results in an ATC product of 2591.2.

7.5 Modern Applications of Pairings

In the last number of years, several interesting cryptographic schemes and applications that rely on bilinear pairings have been proposed. Some of these applications and the computation of pairings on devices that are suitable for their implementation are discussed in this section.

7.5.1 Attribute-Based Encryption

The use of bilinear pairings in Attribute-Based Encryption (ABE) has garnered much interest in recent times. In ABE, a user of a system must possess one or more attributes to be a member of a network and to gain access to particular information within that network. Cryptography based on ABE is a natural fit for securing confidential information in the medical field [177], [178]. A patient's medical information can, for example, be encrypted so that it can only be accessed by health care professionals that have certain attributes. The access policy is specified using a Boolean formula. For example, the policy for access to Patient A's medical records while staying in Hospital H for a surgical operation may be defined by:

((employed by H) AND (surgeon OR anaesthesiologist)) OR (Patient A's general practitioner)

In 2015, Zavattoni et al. [179] published a detailed description of ABE and its associated computational costs. The techniques used to perform point scalar multiplication, exponentiation and pairing computation are first discussed. When the point to be multiplied, P, is not known in advance, the *Non-Adjacent Form* (NAF) windowing method of [23] is used in conjunction with the methods that Gallant et al. [180] describe to accelerate scalar multiplication on BN curves. Consider the scalar multiplication $[n]P$, where n is some random integer. Let A be the cost of a point addition and D be the cost of a point doubling. Then unknown point scalar multiplication is performed at a cost of $\frac{l}{2}D + \frac{l}{w+1}A$, where w is the NAF window size and l is of size that is at most one bit larger than the bit size of n.

In ABE, an input argument for a scalar multiplication or a pairing is often known in advance. This is the case when, for example, a particular point is defined at system set up and rarely changes throughout the operating life of the system. If the point P is known in advance, a scalar multiplication by a random integer n can be performed using the comb method [23], which relies heavily on precomputation. This has a relatively large storage requirement of 2^w points but the cost of known point scalar multiplication is $d(A + D)$, where $d = l/w$. This is a much lower cost in comparison to the unknown point scalar multiplication operation.

When pairing computation must be performed and the input arguments are not known in advance, the optimal *Ate* pairing computation methods described by Aranha et al. [140] are used. A Miller loop is computed at a cost of $6785m_E + 3022r_E$, where m_E is the cost of an integer multiplication of two 256-bit integers and r_E is the cost of the Montgomery reduction from the 512-bit result to a 256-bit integer. The final exponentiation costs $3526m_E + 1932r_E$. When one of the inputs to the pairing is known in advance, then computation time can be reduced by precomputing the line functions and storing their values prior to pairing computation [181]. Clearly, this is expensive in terms of storage. When a product of pairings is required, the number of pairing computations can be reduced at the expense of more scalar multiplications by grouping pairing input arguments. The techniques of [182] and [130] can also be used to share the accumulating function and the final exponentiation step. Encryption and decryption require a combination of these operations.

Zavattoni et al. implement ABE on an Intel Core i7. For a six attribute implementation, encryption requires 2.38 MCycles, key generation requires 652 Kcycles, while decryption requires 4.61 MCycles. Interestingly, the number of cycles required by the pairing operations is only 57.3% of the overall cost due to the techniques used to reduce the complexity of pairing computation. This publication provides an excellent overview of ABE and its associated operations.

7.5.2 Mobile Devices

Until relatively recently, the use of pairing-based cryptography on mobile devices has not received much attention in the literature as limited processing power rendered implementation impractical. In 2011, however, De Caro and Iovino [183] presented a compact Java library, which they call jPBC, computing the operations required by PBC. The library is a port of the PBC library, written in *C*, presented by Lynn [64] and publically available at [184]. Since Java is widely used in the mobile community, jPBC can be installed on smartphones that use the Android operating system with minimal effort. The jPBC library supports the six curves available in Lynn's PBC library. The properties of these curves, called types A to G, are discussed by Galbraith et al. in [185]. The jPBC library has a hierarchy of access interfaces, at the top of which is a pairing interface that selects the pairing to be performed according to the curve that has been selected. In a similar fashion to PBC and the *C++* Miracl library [77], jPBC also has interfaces for various fields and associated arithmetic on their members. Preprocessing of exponentiation and pairing operations is performed in jPBC to compensate for the reduction in speed brought about by the use of Java instead *C*. This is sometimes possible when operations are performed on a particular point that does not change after system setup. De Caro and Iovino report results for a pairing computed on a type A supersingular curve of form $y^2 = x^3 + ax$, where $a \in \mathbb{F}_q$. This curve has an embedding degree of 2. A field size of 512 bits is used. The library is implemented on a Samsung 19000 Galaxy S and on a HTC Desire HD A9191 smartphone. Both implementations return similar results. On the Samsung Galaxy, a pairing is returned in 516.6 ms without preprocessing. The preprocessing techniques provide significant speed up, with a pairing result returned in 253.9 ms. De Caro and Iovino also show how the library can be used to perform the BLS signature scheme [186], although no timing results are provided. The creation of a version of the library that accesses assembly to perform the required operations may, perhaps, provide further acceleration of the required operations. The jPBC library is publically available at [187].

In 2015, Malina et al. [188] investigated the use of pairing-based cryptography on mobile phones running the Android operating system. Several optimisations for the implementation of PBC schemes are discussed. The pairing-based and modular arithmetic operations are performed using jPBC and the Java *Math.BigInteger* library. The BBS04 short signature scheme [189] is implemented. Malina et al. discuss how this scheme can be performed efficiently in this setting. Pairing precomputation is employed when a pairing must be performed on inputs that do not change after system setup. Bilinearity is used to reduce the number of pairing computations required when pairings are multiplied. The product $(e(P,Q)^T).(e(P,S)^W)$ is, for example, computed according to $e(P,Q^T.S^W)$. The signature stage requires three pairing computations as precomputation and use of the bilinearity property are not applicable. The number of pairing computations required during the verification stage is, however, reduced from five to one. Batch verification [190] can also be used to reduce the number of required pairing computations. It is possible to combine a set of verification equations, to perform point exponentiation instead of pairing exponentiation and to group pairings that must be performed on the same point(s) when batch verification is utilised. Pairing computation is performed on a type D curve with an order of 175-bits and an embedding degree of 6. The Java library is implemented on the Samsung Nexus i9250 and the LG Nexus 5 mobile devices. Without the optimisation techniques, a BBS04 signature is performed in 10.23 ms on the Nexus 5. The use of the techniques reduces the time to 2 ms. Essentially, this means that the use of BBS04 for signing is a practical proposition. However, even if 10 signatures are batch verified, the average time per verification is 14.275 ms. Malina et al. state that verification should, therefore, only be implemented in applications where time is not a critical issue.

7.5.3 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) have a large range of applications. Their use in battlefield, agricultural and industrial settings is continuously attracting interest. A WSN is an ad hoc network containing (many) devices, called sensor nodes, that monitor and periodically send information about the area in which they are embedded to other devices or to one or more base stations within their range in a territory of interest. This system can provide a much more comprehensive record of the territory than the use of one, usually more expensive and complicated, sensor could. Since a large quantity of nodes is often required, their cost is of primary concern. For this reason, microprocessors are an ideal candidate for use in WSN nodes. The nodes may also be embedded in inhospitable environments: once entrenched, a sensor may be very difficult to retrieve. For this reason, minimisation of the energy dissipated by the nodes is also of concern as it may not be possible to change a power source after deployment. The literature results for pairing implementation in the context of WSNs are listed in Table 7.4.

Table 7.4: Notable modern contributions to the implementation of pairings in the context of wireless sensor networks.

Ref.	**Year**	**Sec.** (Bits)	**Curve**	**Alg.**	**Device**	**F.** (MHz)	**Cycles** ($\times 10^6$)	**Time** (ms)
[191]	2011	72	$E(\mathbb{F}_{2^{271}})$	η_T	Atmel ATMega128	7.4	14.06	1900
					TI MSP430	8	10.4	1270
					Intel XScale	13	1.81	140
[192]	2011	72	$E(\mathbb{F}_{2^{283}})$	η_T	ASIC - $0.574mm^2$	200	0.14	0.70
[193]	2012	128	BN-254	a_{opt}	TI MSP430	8	79.44	9930
					TI MSP430X		67.68	8460
					TI PSP430X+GTY		47.76	5970
[194]	2014	128	BN-256	a_{opt}	Cort-M0+	48	47664	993
					+ MAC		17952	374
					+ h/w		7776	162

In 2011, Oliveira et al. [191] discussed the use of pairings in wireless sensor networks. In WSNs, authentication and secure key distribution is particularly difficult as the nodes do not have the ability to store a large number of keys. An *Identity-Based Non-Interactive Key Distribution Scheme* (ID-NIKDS), described by Sakai et al. in [27], can provide a solution to this problem. In such a scheme, a secret key can be established between two nodes using their public identities alone: they do not need to communicate with each other. A secure information exchange can then proceed using a traditional symmetric key scheme such as AES. As with most other ID-based schemes, each node is provided with a unique secret key before deployment. The unique keys are generated from one master key by a trusted authority. Pairing computation is by far the most costly operation in ID-NIKDS schemes. Oliveira et al. discuss the implementation of pairings on 8-, 16- and 32-bit RISC processors that are commonly used in sensor networks. The exploration of finite field implementation using small instruction sets, low memory availability and a limited number of lookup tables is very interesting. A characteristic 2 η_T pairing is performed on $E(\mathbb{F}_{2^{271}})$, which provides a 72-bit security level. Results are returned in 1.9 s, 1.27 s and 0.14 s on the 8-, 16- and 32-bit processors, respectively.

In the same year, McCusker and O'Connor [192] presented an IBE system for secure key distribution and access control in a WSN. A technique is also proposed that reduces the vulnerability of the system to node capturing. The system employs (ID-NIKDS) [27] and the Identity-Based Signature (IBS) scheme described by Barreto et al. in [195]. Prior to deployment, the necessary system parameters and private keys are stored on the sensor nodes by the Key Generation Centre (KGC). Immediately after deployment, pairs of neighbouring nodes establish a key. Each device then transmits a small signed message to all nodes within radio range. A device that can generate a valid signature is allowed to join the network. Each node then keeps a record of authenticated devices within its own radio range. If a node is to be added, the KGC broadcasts the identity of the new device along with a time-stamped signature. The nodes await a small signed message from the new node and add it to the network if correct. Node removal is performed in a similar manner. McCusker and O'Connor describe their system in terms of an environmental monitoring application in which the nodes are static but information is extracted by a mobile entity such as a trusted employee carrying a laptop or tablet device. When a reading is required, a request is relayed between the nodes until the desired device is reached. Information is then sent back from node to node using AES. Focus is placed on device area utilisation and energy usage. A Tate pairing using the Duursma and Lee method, performed on the supersingular elliptic curve $E(\mathbb{F}_{2^{283}})$, is used to explore the viability of the scheme. An implementation, performed on a 32-bit RISC ARM920T processor, computes a pairing in 177.1 ms. Symmetric key generation and signature verification require two Tate pairing computations, an exponentiation and a scalar multiplication. A total of 35.4 mJ and 444.5 ms are required to implement the scheme at 140 MHz. These values are too high for practical use. McCusker and O'Connor discuss the augmentation of the system with a hardware unit. Pairing computation is outsourced to this unit. The top level architecture contains an FSM, an ALU and some registers. An $\mathbb{F}_{2^m}$ bit-serial multiplier, a squaring module and a square rooting module are used to perform all necessary operations. The ALU also contains registers, multiplexers and combinatorial logic that control the datapaths through which extension field multiplication and inversion are performed. The hardware unit is interfaced to the ARM device using the Advanced Peripheral Bus (APB) scheme. A synchroniser is used to manage the two clock domains. The hardware processor is synthesised on a 65 nm CMOS technology. The ASIC computes a pairing in 698.11 µs, uses 29.6 µJ per computation and has a footprint of $0.574mm^2$. However, even with the significant reduction in pairing computation time, McCusker and O'Connor state that the scheme is not suited to use in WSNs as a total of 7.26 mJ and 91.7 ms are required to operate the scheme. They hypothesise that significant improvements could be attained if elliptic curve scalar multiplication and exponentiation were also accelerated in hardware. A resultant energy consumption of 80 µJ and a scheme operation time of 1.75 ms is predicted. If this were the case, it is claimed that this system would be appropriate for WSN use.

In 2012, Gouvêa et al [193] discussed the computation of several curve and pairing-based cryptographic protocols on platforms that are suited to WSN applications. Implementation is performed on three 16-bit TI microcontrollers from the MSP430 family. The first microcontroller, simply called the MSP430, has 12 general-purpose registers and an instruction set for addition, subtraction and 1-bit shifts. Integer multiplication is performed using a peripheral hardware multiplier that can be used on all microcontrollers in the family. The next device is the MSP430X that, in addition to the features of the MSP430, has an instruction set that enables multiple register read and writes and 4-bit shifts using a single instruction each. It also has an address space that is extended to 20 bits, has more memory and has faster data-memory transfer. The third implementation is performed on an MSP430X device that features a dedicated 32×32 bit hardware multiplier (called an MPY). A 256-bit BN curve optimal *Ate* pairing computation is performed on each of the devices. A result is returned in 9.93 s, 8.46 s and 5.97 s on the MSP430, the MSP430X and the MSP430X+MPY32 devices, respectively. The ID-KDNS [27] scheme is implemented on each of the microcontrollers. It should be noted that the use of the IBS scheme for node authentication as suggested by McCusker and O'Connor [192] is not employed and so only one pairing computation must be performed by each node. On the MSP430X+MPY32 implementation, a key agreement can be reached in a runtime of 6.13 s on one node and 7.12 s on the other. Note that the slight increase in computation time in comparison to the pairing is due to relatively inexpensive hashing functions and other similar operations. Keeping in mind that this is a 128-bit implementation, this is an excellent result that demonstrates that pairings returning high security levels can be implemented in a reasonable time on very small devices.

In 2014, Unterluggauer and Wenger [194] discussed the computation of pairings on an ARM microprocessor augmented with hardware peripherals. Three implementations, small enough for use in embedded systems, are presented. Arithmetic is performed using the techniques of Beuchat et al. [116] while pairings are computed using the explicit formulae of Costello et al. [150]. The inversion trick discussed by Aranha et al. [140], the $\mathbb{F}_{p^2}$ lazy reduction technique described by Beuchet et al. [116] and a variant of the fast exponentiation technique discussed by Fuentes-Castañeda et al. [196] are also employed. To counter side channel attacks, all implementations have data-independent runtime, use randomised projective coordinates, and perform intermediate point verifications. The first implementation is performed on a processor that is functionally equivalent to the 32-bit Cortex-M0+ microprocessor. This ARM processor has a very low area footprint (a minimum of 12KGates) and was designed with energy efficiency in mind. This makes it extremely suitable for embedded applications. The Cortex-M0+ contains a $32 \times 32 \rightarrow 32$ multiplier. Since the operands are small relative to the applicable field sizes, multiplication and reduction would take a relatively large amount of time if this multiplier were used alone. The second implementation utilises the same processor with an extra multiply-and-accumulate (MAC) extension module. The use of this module means that the result of a $32 \times 32 \rightarrow 64$ bit multiplication can be stored in three accumulation registers in one cycle. The third implementation consists of the original microprocessor architecture and a hardware unit for accelerated field addition, subtraction and Montgomery multiplication. The hardware unit is synthesised on a low-leakage 130 nm UMC technology. Optimal *Ate* pairings are computed at the 128-bit security level. Power and timing information is collected from the three implementations at a clock frequency of 48 MHz in each case. The basic microprocessor implementation requires 53,700 gates, consumes 5.8 mW and computes a pairing in 993 ms. The MAC extension increases the total area to 58,800 gates and the power to 7.33 mW but reduces the runtime to 374 ms. The implementation that includes the hardware accelerator requires 57,700 gates and consumes 9.96 mW. It does, however, decrease the runtime to 162 ms. Energy consumption is a major consideration for embedded systems. The implementations require 5.76 mJ, 2.74 mJ and 1.61 mJ per pairing computation, respectively. These are the lowest energy values that have been published to date. It should also be noted that the area count of each of the implementations is very low in comparison to most other implementations that have been discussed.

7.6 Future Directions

As seen in this chapter, the software and hardware implementation of pairings attracts much attention in the modern literature. This is the case due to the continuous suggestion of new, efficient, techniques for pairing computation and due to improvements in the processing power and architectural features of hardware and software platforms. It is highly probable that further optimisations will become available, at least into the near future, as PBC is a highly vibrant area in which novel applications are continuously proposed. This will provide the software and hardware designer with much challenging work in the years to come.

Many applications of pairings require computation in environments in which it may be difficult to access or retrieve a device once deployed. As a result, power consumption may be of primary concern. The careful design of a dedicated hardware processor that targets the implementation of cryptographic schemes that are suitable for such environments may prove beneficial. It may be useful to create a software system similar to that of Chapter 6 (in which area/time trade-offs are explored) to examine trade-offs between power consumption and computation time on various devices and architectures. As seen in the previous section, the use of PBC in WSNs is attractive due to the power savings that are provided by identity-based schemes. In 2015, Chen et al. [197] used PBC as a basis to propose a dynamic key management and authentication system for WSNs. The dynamic key management capabilities are used to update session keys when required. Pairings are used to enable the mutual authentication of nodes. Interestingly, the Global Positioning System (GPS) is used to define the most efficient route for information transmission through the network and can be used to intelligently group sensor nodes into clusters that can be arranged hierarchically. This is an excellent example of a very interesting and powerful use of pairings and an investigation into the efficient implementation of such a system may prove worthwhile.

Attribute-based encryption is a continuously evolving area. Many applications of ABE have yet to be explored. The use of ABE in a WSN setting may be an interesting area of research. It may be possible to utilise ABE in order to facilitate access control in different geographical areas or territories. Access to sensing data can also be controlled if, for example, a particular node is used to sense more than one parameter. This may be of particular interest as nodes are often placed in insecure locations. Access can also be readily modified if the location of a node, or nodes, must be changed.

As seen in the previous section, Zavattoni et al. [179] show that the cost of pairing computation in their ABE implementation is only 57.3% of the overall total. It would be interesting to investigate whether a dedicated hardware architecture could be designed that computes known and unknown point scalar multiplication, known and unknown pairing computation and the products of pairings in an efficient manner. These computations can be performed by similar arithmetic units in many cases. By combining and scheduling the operations in an intelligent fashion and exploring various design strategies, an ABE processor with a very high resource utilisation could be created. It may, therefore, be possible to design a very area-efficient ABE processor that performs encryption and decryption relatively quickly.

The practical implementation of PBC on highly constrained devices such as smart and SIM cards may open a very large area for research as pairing-based schemes that were, in most cases, originally only suitable for implementation on larger processors, could be employed in an end-to-end fashion. In 2014, Chung et al. [198] discussed the implementation of pairings on USB tokens. They demonstrated that PBC can be used in a practical fashion at the 128-bit security level. Much work is, however, required on the efficient implementation of pairings on highly-constrained devices before the widespread adoption of PBC will become attractive. This is an area in which much further work is required.

There have been relatively few investigations into the implementation of pairings on modern smartphones and tablets. Many of these devices have relatively powerful one core, or multi core, processors that are supplemented with fast memory access, efficient computation pipelines, wide data bandwidth and other appealing attributes. Such devices may be exceptional candidates for pairing computation due to the multitude of such features. Multi core implementation may, for example, mean that arithmetic on sub fields and their extensions can be performed in parallel. If two pairing computations are required, operations can also be shared across cores. An instruction-level simulation on an architecture that is very similar to modern smartphones may be useful as a proof of concept.

An investigation into the development of a pairing-based system on devices produced by Apple, Inc. may prove very beneficial. The open source *Swift* programming language [199], released in 2014, can be used for application development. This language is constructed on top of *C* and *Objective-C* code and provides support for the operations performed by these languages. Importantly, object-oriented functionality is also available. The open source LLVM compiler [200] can be used to convert Swift code into optimised native instructions on Apple devices that are built on the OS X and iOS operating systems. The open source nature of the language means that it can be ported to the web successfully. A good example of this is the IBM, Inc. *Kitura* web-based framework [201]. This is also open source and reduces the effort required to develop end-to-end solutions in modern mobile and internet environments. The development of a mobile phone application using Swift, in conjunction with a web framework, for secure communication using pairing-based cryptography could be very valuable.

In 2015, Jacobsen et al. [202] discussed the use of identity-based cryptography in home area networks. The rapid growth of the *Internet of Things* means that there is a constant need for new ways to secure devices that are connected wirelessly. These devices often have a very low energy capacity. This is an area of interest for pairing-based cryptography as non-interactive key distribution could provide significant savings in power consumption.

The outsourcing of computation to cloud servers is a rapidly expanding research area. In 2015, Chen et al. [203] proposed an algorithm for the secure outsourcing of point multiplication, exponentiation and pairing computation to an untrusted cloud server. They demonstrate that secure identity-based encryption and signature schemes can be implemented using their algorithm. It would be interesting to investigate whether some outsourcing could be used to make schemes that rely on pairings feasible on devices that have a very small area or stringent energy requirements.

7.7 Conclusions

The current state of the art of pairing-based cryptography has been presented in this chapter. Modern pairing computation techniques have been discussed. Computational and side channel attacks on cryptographic systems have also been outlined. Notable software and hardware implementations of the Tate, η_T, *Ate* and optimal *Ate* pairings have been described. Although the processors presented in this thesis perform small characteristic pairings, this does not diminish the contributions of this work. A significant influence can be seen in much of the literature since 2008. Tower fields are carefully defined with an efficient hardware implementation in mind. A variety of systems have been designed in such a fashion that arithmetic operations required on different iterations of the loop can be performed in parallel. Algorithms for pairing computation on BN curves are structurally similar to those of the η_T pairing: they mainly comprise an expensive Miller loop that performs many finite field operations. This means that many of the design strategies discussed in this thesis can still be used. In much of the literature, the sparse nature of extension field variables has been exploited to reduce the number of subfield operations that are required to perform arithmetic. At the time of publication, the flexible processor of Chapter 6 was among the first implementations to demonstrate that pairing-based cryptography is viable in environments in which area is very restricted. Several subsequent implementations have used similar architectures for pairing computation. Software systems that reduce the effort required for the prototyping of pairing processors have been used in conjunction with those systems in a similar fashion. These systems often utilise automatic VHDL conversion and rapid instruction sequence generation techniques.

As a suggested modern application for the work discussed in this thesis, consider an application in which a set of sensing units must be embedded in a very large habitat that consists of several inhospitable territories in which bandwidth is very limited. Once deployed, it may be difficult to retrieve the sensors. Before deployment, the sensing units are grouped by territory. Each group is associated with one base station device that communicates with and receives information from its designated nodes. A central server retrieves the collected data from each of the base stations at intervals. Before deployment, the sensors and base stations are allocated the keys and system parameters that are required for secure field communication. In this example, the sensor nodes should be inexpensive and compute a low-energy pairing due to retrieval difficulties. An architecture similar to the processor of Chapter 6, configured with a small number of low-digit multipliers and implemented on a low cost FPGA, would be ideal in this case. Computation speed is more of a consideration on the base station device as it must communicate with each of its designated sensors and send data to the central server. The base station must also be provided with more power. A suitable, most likely larger, configuration can be chosen and implemented on an FPGA (cost may be less of a consideration as only one base station is required in each territory). The software design system can be used to explore suitable implementations for the sensors and stations before deployment. The server itself may comprise an even larger flexible processor. If more speed is required, then a high throughput architecture similar to those presented in Chapters 4 and 5 can be used to quickly and securely collect data from the base stations within every territory of the habitat.

Chapter 8

Conclusions

The thesis is summarised in this section. The contributions of the research are also discussed.

In Chapter 2, the cryptographic use of elliptic and genus 2 hyperelliptic curves is discussed. The theory of groups, rings and finite fields is outlined. The mathematics necessary for an understanding of elliptic and genus 2 hyperelliptic curve cryptography is then explained. Curve and divisor theory is discussed. The operations necessary to perform scalar multiplication on these curves are also described. Attacks on the DLP, the ECDLP and the HCDLP are discussed, along with the measures that should be used to prevent them. Finally, the benefits of hyperelliptic curve cryptography are outlined.

In Chapter 3, the Tate and η_T pairings are introduced. The optimisations available in the literature, leading to the definition of the η_T pairing, are discussed. The Tate pairing can be performed efficiently by computing an η_T pairing, followed by a well-defined exponentiation. Security considerations for the use of pairings in cryptography are also presented. The Boneh-Franklin IBE scheme is outlined.

The motivation for the methodology used in this thesis is also discussed in Chapter 3. The relatively low cost and reconfigurability of FPGAs means that they are an ideal platform on which to implement the processors discussed in this thesis. Three processors have been created during this work. Each must be defined using detailed low level RTL VHDL. The tools used to synthesise, place and route the processors are resource intensive. Verification and benchmarking are also time-consuming due to the complexity of the operations involved. A software system has been designed to address these issues. The software is written using object-oriented *C++*. Classes for software pairing computation, RTL VHDL generation, efficiency analysis, automatic implementation, rapid benchmarking and robust verification have been created. A program for the automatic generation of instruction sets according to user-defined algorithms has also been written. This program can be used in conjunction with the flexible processor of Chapter 6 to implement any algorithm that relies on finite field operations.

In Chapter 4, a hardware processor for Tate pairing computation on a characteristic 2 elliptic curve is presented. The algorithms and operations required for pairing computation on such a curve are provided. The computation and implementation of arithmetic on $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^{4m}}$ is discussed. A generic algorithm for $\mathbb{F}_{2^{4m}}$ inversion requires many $\mathbb{F}_{2^m}$ operations and some conditional statements that can complicate a hardware control system. In this work, the sparse nature of the irreducible polynomial defining $\mathbb{F}_{2^{4m}}$ is exploited to dramatically reduce the number of required operations. Inversion can be performed by grouping computations into three fixed steps. A total of 33 $\mathbb{F}_{2^m}$ multiplications, four squarings, 16 additions and one inversion are necessary. The multiplications are efficiently scheduled through three multipliers. An $\mathbb{F}_{2^{4m}}$ inversion result is returned in $13(m/D)$ clock cycles if $3(m/D) > 2m$ or in $10(m/D) + 2m$ cycles otherwise. A hardware unit implementing this inversion technique was presented at the 3rd International Conference on Reconfigurable Computing and FPGAs in 2006 [7]. The inverter was then incorporated into a characteristic 2 Tate pairing processor architecture implementing the Tate pairing using the BKLS algorithm. This work was published in a special issue of the Computers & Engineering Journal in 2007 [8].

Dedicated hardware units for the fast computation of the main stages of the iterative loop of the characteristic 2 η_T pairing (Algorithm 5) are also presented in Chapter 4. A precomputation unit calculates and stores all squares and roots in $2m$ clock cycles. Units for the computation of u_0 and u_1, their sparse multiplication, and $\mathbb{F}_{2^{4m}}$ multiplication have also been created. The last three units were designed so that operations required on different iterations of the loop can be performed in parallel with ease, significantly reducing the number of cycles required for loop completion. A unit that performs the final exponentiation is also described. The processor that uses these units to compute a Tate pairing is then presented. It is implemented on a Virtex-II Pro FPGA. Results show that it can perform a full Tate pairing in a very low number of clock cycles. The custom processor was presented at the 2006 IEEE Conference on Field Programmable Technology [6].

In Chapter 5, the hardware implementation of the genus 2 hyperelliptic curve Tate pairing is discussed. Genus 2 curves offer an embedding degree of 12, which means that a smaller subfield can be used than in the elliptic cases. This benefit is, however, offset by a more complicated pairing construction and the challenges of computation on extensions of high degree. Algorithms for the Tate pairing, calculated using the η_T method, are provided in Chapter 5. Tower field constructions that enable fast extension field arithmetic are also discussed. Hardware units for the accelerated computation of the genus 2 iterative loop (Algorithm 10) are presented. A precomputation unit is carefully designed to supply squares and roots when necessary. Two sparse members of $\mathbb{F}_{2^{12m}}$ must be multiplied. The $\mathbb{F}_{2^{12m}}$ result is then multiplied by the accumulating Miller variable on each iteration. A custom dual mode multiplication unit performs both of these operations quickly while minimising resource utilisation. The genus 2 processor is implemented on a Virtex-II Pro FPGA. Results show that a pairing can be computed in the same order of clock cycles as in the characteristic 2 elliptic case. Furthermore, the genus 2 implementations are more efficient, returning the lowest AC products. The genus 2 implementation also scales better with increases in security level due to the high embedding degree. Prior to this work, there had been no published hardware implementation of a pairing using any type of algorithm on genus 2 curves. An early, more general purpose, version of the genus 2 processor was presented at the 2006 IEEE Conference on Information Technology: New Generations [9]. The architectures of the custom hardware units and the final genus 2 processor were published in The Journal of Systems Architecture in 2007 [10].

In Chapter 6, a flexible processor for characteristic 2 and 3 elliptic curve Tate pairing computation is presented. During the initial stages of the work, analysis showed that hardware implementation of parallelised characteristic 3 extension field arithmetic would lead to an unjustifiable area utilisation. This analysis provided the motivation for the creation of a processor that employs subfield modules alone. The top level architecture contains RAM for storage, an ALU, tristate buffers and a control system. The ALU can be programmed to contain either characteristic 2 or 3 arithmetic modules. The quantity of multiplication modules and their digit sizes can be varied using the flexible software design system. The control system contains an FSM, a counter and a ROM. The FSM handles the counter, which accesses an instruction set contained in the ROM. A *C++* class has been written to reduce the instruction set generation effort. An algorithm that is to be implemented can be defined in terms of subfield and extension field operations using all of the functionality of *C++*. The instruction set is then generated automatically and flashed to ROM. Characteristic 2 and 3 versions of the processor are implemented on a Xilinx Virtex-II Pro FPGA for various versions of the ALU. Results show that maximal efficiency is provided when area utilisation is low: implementations with small footprints can return pairings in relatively low numbers of clock cycles. The primary focus of the processors of Chapters 4 and 5 and, indeed, those discussed in much of the literature is fast pairing computation. Those hardware architectures are more fixed in nature and it can be difficult to maintain performance and efficiency when they must be scaled down in area. The flexible processor described here is an excellent computation platform for environments with small area profiles. The features of the software system and the ease of architectural modification enables the rapid and detailed exploration of solutions for various applications. The processor is also suited to embedded environments in which bandwidth is limited. If the algorithm to be implemented must be modified, the associated instruction set can be generated in a central system (such as a server). This can be sent to the embedded system in a very small file, along with a control signal indicating that the ROM should be updated. This means that a large architectural reprogramming file does not need to be sent to modify the operations performed by a remote device. A characteristic 3 version of the flexible processor, performing the reduced η_T pairing without conversion to the Tate pairing, was presented at the 2007 IEEE conference on Information Technology: New Generations [11]. An implementation computing the Tate pairing was published in the International Journal of High Performance Systems Architecture in 2007 [12]. Subsequent to this, the processor was further discussed and characteristic 2 and 3 pairing results published in a chapter of a 2009 IOS Press book titled *Identity-Based Cryptography* [13].

In chapter 7, the state of the art of pairing-based cryptography has been presented. Modern pairing computation techniques, security issues, and developments in software and hardware implementations have been described. The ideas, design strategies and architectures of this thesis have had an influence on many of these implementations. Many novel pairing-based applications have been proposed in the literature. The implementations and suggested use of pairings on such a wide breadth of devices as large, dedicated ASICs, expensive to low cost FPGAs, general purpose serial processors, microprocessors and even on USB tokens is a large cause for optimism that PBC will remain a rapid growth area.

Publications

The following is a list of the publications arising from this work.

- R. Ronan, C. Ó hÉigeartaigh, C. Murphy, M. Scott, T. Kerins, and W. P. Marnane. An embedded processor for a pairing-based cryptosystem. In *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*, pages 192–197. IEEE, 2006 Abstract:

- R. Ronan, C. Ó hÉigeartaigh, C. Murphy, M. Scott, and T. Kerins. FPGA acceleration of the Tate pairing in characteristic 2. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 213–220. IEEE, 2006

- M. Keller, R. Ronan, W. Marnane, and C. Murphy. A GF(2^{4m}) inverter and its application in a reconfigurable Tate pairing processor. In *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, pages 1–10. IEEE, 2006

- R. Ronan, C. Ó hÉigeartaigh, C. Murphy, T. Kerins, and P. S. L. M. Barretto. A reconfigurable processor for the cryptographic η_T pairing in characteristic 3. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*, pages 11–16. IEEE, 2007

- R. Ronan, C. Murphy, T. Kerins, C. Ó hÉigeartaigh, and P. S. L. M Barreto. A flexible processor for the characteristic 3 η_T pairing. *International Journal of High Performance Systems Architecture*, 1(2):79–88, 2007

- R. Ronan, C. Murphy, M. Scott, and T. Kerins. Hardware acceleration of the Tate pairing on a genus 2 hyperelliptic curve. *Journal of Systems Architecture*, 53(2):85–98, 2007

- M. Keller, R. Ronan, W. Marnane, and C. Murphy. Hardware architectures for the Tate pairing over GF(2^m). *Computers & Electrical Engineering*, 33(5):392–406, 2007

- M. Keller, R. Ronan, A. Byrne, C. Murphy, and W. P. Marnane. Hardware implementation of pairings. In M. Joye and G. Neven, editors, *Identity-based Cryptography*, Cryptology and information security series. IOS Press, 2009

Bibliography

[1] Network World. Chapter 2 of SSL VPN technology. `http://www.networkworld.com/article/2268575`.

[2] Wikipedia. ID-based encryption. `https://en.wikipedia.org/wiki/ID-based_encryption`.

[3] G. Kessler. An overview of cryptography. `http://www.garykessler.net/library/crypto.html`.

[4] S. Singh. *The Code Book: The Secret History of Codes and Code-breaking*. Harper Collins, 2002.

[5] R. Dutta, R. Barua, and P. Sarkar. Pairing-based cryptographic protocols: A survey. `http://eprint.iacr.org/2004/064.pdf`, 2004.

[6] R. Ronan, C. Ó hÉigeartaigh, C. Murphy, M. Scott, and T. Kerins. FPGA acceleration of the Tate pairing in characteristic 2. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 213–220. IEEE, 2006.

[7] M. Keller, R. Ronan, W. Marnane, and C. Murphy. A GF(2^{4m}) inverter and its application in a reconfigurable Tate pairing processor. In *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, pages 1–10. IEEE, 2006.

[8] M. Keller, R. Ronan, W. Marnane, and C. Murphy. Hardware architectures for the Tate pairing over GF(2^m). *Computers & Electrical Engineering*, 33(5):392–406, 2007.

[9] R. Ronan, C. Ó hÉigeartaigh, C. Murphy, M. Scott, T. Kerins, and W. P. Marnane. An embedded processor for a pairing-based cryptosystem. In *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*, pages 192–197. IEEE, 2006.

[10] R. Ronan, C. Murphy, M. Scott, and T. Kerins. Hardware acceleration of the Tate pairing on a genus 2 hyperelliptic curve. *Journal of Systems Architecture*, 53(2):85–98, 2007.

[11] R. Ronan, C. Ó hÉigeartaigh, C. Murphy, T. Kerins, and P. S. L. M. Barretto. A reconfigurable processor for the cryptographic η_T pairing in characteristic 3. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*, pages 11–16. IEEE, 2007.

[12] R. Ronan, C. Murphy, T. Kerins, C. Ó hÉigeartaigh, and P. S. L. M Barreto. A flexible processor for the characteristic 3 η_T pairing. *International Journal of High Performance Systems Architecture*, 1(2):79–88, 2007.

[13] M. Keller, R. Ronan, A. Byrne, C. Murphy, and W. P. Marnane. Hardware implementation of pairings. In M. Joye and G. Neven, editors, *Identity-based Cryptography*, Cryptology and information security series. IOS Press, 2009.

[14] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* Wiley, 1996.

[15] National Institute of Standards and Technology. *Announcing the Advanced Encryption Standard(AES).* 2001. FIPS PUB 197.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[17] W. Diffie and M. E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[18] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] V. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology-CRYPTO85 Proceedings*, pages 417–426. Springer, 1986.

[20] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[21] N. Koblitz. Hyperelliptic cryptosystems. *Journal of cryptology*, 1(3):139–150, 1989.

[22] J. A. Buchmann, E. Karatsiolis, and A. Wiesmaier. *Introduction to public key infrastructures*. Springer Science & Business Media, 2013.

[23] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[24] A. Joux. A one round protocol for tripartite Diffie–Hellman. In *Algorithmic number theory*, pages 385–393. Springer, 2000.

[25] A. Shamir. Identity-based cryptosystems and signature schemes. In *Advances in Cryptology–CRYPTO 1984*, pages 47–53. Springer, 1984.

[26] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *Advances in Cryptology-CRYPTO 2001*, pages 213–229. Springer, 2001.

[27] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairing. In *Symposium on Cryptography and Information Security (SCIS)*, pages 26–28, 2000.

[28] R. Canetti, S. Halevi, and J. Katz. Chosen-ciphertext security from identity-based encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–222. Springer, 2004.

[29] Kristin Lauter. The advantages of elliptic curve cryptography for wireless security. *IEEE Wireless communications*, 11(1):62–67, 2004.

[30] M. Maas. Pairing-based cryptography. Master's thesis, Technische Universiteit Eindhoven, 2004.

[31] M. Joye and G. Neven. *Identity-Based Cryptography.* IOS Press, 2009.

[32] R. Lidl and H. Niederreiter. *Finite Fields.* Cambridge University Press, 1996.

[33] K. Wong. *Applications of Finite Field Computation to Cryptology: Extension Field Arithmetic in Public Key Systems and Algebraic Attacks on Stream Ciphers.* PhD thesis, Queensland University of Technology, 2008.

[34] A. J. Menezes, Y-H. Wu, and R. J. Zuccherato. *An elementary introduction to hyperelliptic curves.* Faculty of Mathematics, University of Waterloo, 1996.

[35] H. Cohe and G. Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman & Hall, 2006.

[36] W. Fulton. *Algebraic curves: An introduction to algebraic geometry.* Addison-Wesley, 1989.

[37] D. Mumford and C. Musili. *Tata lectures on theta II*, volume 43. Springer, 2007.

[38] P. Gaudry. An algorithm for solving the discrete log problem on hyperelliptic curves. In *Advances in Cryptology, EUROCRYPT 2000*, pages 19–34. Springer, 2000.

[39] I. F. Blake, G. Seroussi, and N. Smart. *Elliptic curves in cryptography*, volume 265. Cambridge university press, 1999.

[40] A. J. Menezes. *Elliptic curve public key cryptosystems.* Kluwer Academic Publishing, 1993.

[41] S. Sadanandan. Addition in Jacobian of hyperelliptic curves. Master's thesis, Indian Institute of Technology, 2004.

[42] R. Harley. Fast arithmetic on genus 2 curves. `http://cristal.inria.fr/~harley/hyper`.

[43] P. Gaudry and R. Harley. Counting points on hyperelliptic curves over finite fields. In *Algorithmic number theory*, pages 313–332. Springer, 2000.

[44] T. Lange. *Efficient arithmetic on hyperelliptic curves.* PhD thesis, Universitat-Gesamthochshule Essen, 2001.

[45] T. Lange. Formulae for arithmetic on genus 2 hyperelliptic curves. *Applicable Algebra in Engineering, Communication and Computing*, 15(5):295–328, 2005.

[46] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in cryptology*, pages 10–18. Springer, 1985.

[47] J. Scholten and F. Vercauteren. An introduction to elliptic and hyperelliptic curve cryptography and the NTRU cryptosystem. `http://homes.esat.kuleuven.be/~fvercaut/papers/cc03.pdf`, 2003.

[48] U. M. Maurer and S. Wolf. The relationship between breaking the Diffie–Hellman protocol and computing discrete logarithms. *SIAM Journal on Computing*, 28(5):1689–1721, 1999.

[49] D. Meffert. Bilinear pairings in cryptography. Master's thesis, Radboud Universiteit Nijmegen, 2009.

[50] D. Shanks. Class number, a theory of factorization, and genera. In *Proc. Symp. Pure Math*, volume 20, pages 415–440, 1971.

[51] J. M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[52] V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology-EUROCRYPT 1997*, pages 256–266. Springer, 1997.

[53] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[54] L. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In *20th Annual Symposium on Foundations of Computer Science (SFCS 1979)*, pages 55–60, 1979.

[55] N. P. Smart. The discrete logarithm problem on elliptic curves of trace one. *Journal of cryptology*, 12(3):193–196, 1999.

[56] A. J. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *Information Theory, IEEE Transactions on*, 39(5):1639–1646, 1993.

[57] P. Gaudry, E. Thomé, N. Thériault, and C. Diem. A double large prime variation for small genus hyperelliptic index calculus. *Mathematics of Computation*, 76(257):475–492, 2007.

[58] G. Frey and H.-G Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of computation*, 62(206):865–874, 1994.

[59] V. S. Miller. Short programs for functions on curves. `http://homes.esat.kuleuven.be/~fvercaut/papers/cc03.pdf`, 1986.

[60] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In *Advances in cryptology-CRYPTO 2002*, pages 354–369. Springer, 2002.

[61] S. D. Galbraith, K. Harrison, and D. Soldera. Implementing the Tate pairing. In *Algorithmic number theory*, pages 324–337. Springer, 2002.

[62] I. Duursma and H. S. Lee. Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$. In *Advances in cryptology-AsiaCrypt 2003*, pages 111–123. Springer, 2003.

[63] P. L. S. M. Barreto, S. Galbraith, C. Ó hÉigeartaigh, and M. Scott. Efficient pairing computation on supersingular abelian varieties. *Designs, Codes and Cryptography*, 42(3):239–271, 2007.

[64] B. Lynn. *On the implementation of pairing-based cryptosystems.* PhD thesis, Stanford University, 2007.

[65] N. El Mrabet. Efficient computation for pairing based cryptography: A state of the art. *Theory and Practice of Cryptography and Network Security Protocols and Technologies*, page 51, 2013.

[66] H. G. Rück. On the discrete logarithm in the divisor class group of curves. *Mathematics of Computation of the American Mathematical Society*, 68(226):805–806, 1999.

[67] V. S. Miller. The Weil pairing, and its efficient calculation. *Journal of Cryptology*, 17(4):235–261, 2004.

[68] C. Ó hÉigeartaigh. *Pairing computation on hyperelliptic curves of genus 2*. PhD thesis, Dublin City University, 2006.

[69] R. Granger, F. Hess, R. Oyono, N. Thériault, and F. Vercauteren. *Ate* pairing on hyperelliptic curves. *Advances in Cryptology-EUROCRYPT 2007*, pages 430–447, 2007.

[70] S. D. Galbraith. Supersingular curves in cryptography. In *Advances in Cryptology-ASIACRYPT 2001*, pages 495–513. Springer, 2001.

[71] E. R. Verheul. Evidence that XTR is more secure than supersingular elliptic curve cryptosystems. In *Advances in Cryptology-EUROCRYPT 2001*, pages 195–210. Springer, 2001.

[72] Xilinx. Virtex-II Pro FPGA Datasheet. `http://www.xilinx.com/support/documentation`.

[73] Alpha-Data. ADM-XRC-II datasheet. `http://www.alpha-data.com`.

[74] Celoxica. RC-2000 datasheet. `http://www.celoxica.com`.

[75] Xilinx. FPGA Design Flow Overview. `http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm`.

[76] Modelsim. Modeslim XE. `http://www.xilinx.com/support/documentation`.

[77] Miracl. The Miracl library. `http://www.miracl.com`.

[78] A. Byrne. *Reconfigurable Architectures for Elliptic Curve and Pairing Based Cryptography*. PhD thesis, University College Cork, 2009.

[79] J. Fan, F. Vercauteren, and I. Verbauwhede. Efficient hardware implementation of $\mathbb{F}_p$-arithmetic for pairing-friendly curves. *Computers, IEEE Transactions on*, 61(5):676–685, 2012.

[80] E. Cuevas-Farfán, M. Morales-Sandoval, R. Cumplido, C. Feregrino-Uribe, and I. Algredo-Badillo. A programmable FPGA-based cryptoprocessor for bilinear pairings over $\mathbb{F}_{2^m}$. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, pages 1–8. IEEE, 2013.

[81] P. A. Scott, S. E. Tavares, and L. E. Peppard. A fast VLSI multiplier for $GF(2^m)$. *Selected Areas in Communications, IEEE Journal on*, 4(1):62–66, 1986.

[82] E. Ferrer, D. Bollman, and O. Moreno. A fast finite field multiplier. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 238–246. Springer, 2007.

[83] E. D. Mastrovito. *VLSI architectures for computations in Galois fields*. PhD thesis, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 1991.

[84] H. Fan and M. A. Hasan. Fast bit parallel-shifted polynomial basis multipliers in $GF(2^n)$. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 53(12):2606–2615, 2006.

[85] C. Negre. Efficient parallel multiplier in shifted polynomial basis. *Journal of Systems Architecture*, 53(2):109–116, 2007.

[86] L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *Journal of VLSI signal processing systems for signal, image and video technology*, 19(2):149–166, 1998.

[87] H. Wu. Bit-parallel finite field multiplier and squarer using polynomial basis. *Computers, IEEE Transactions on*, 51(7):750–758, 2002.

[88] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and computation*, 78(3):171–177, 1988.

[89] J. Guajardo and C. Paar. Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. *Designs, Codes and Cryptography*, 25(2):207–216, 2002.

[90] Kerins T. *Architectures for cryptography based on elliptic Curves.* PhD thesis, University College Cork, 2005.

[91] H. Brunner, A. Curiger, and M. Hofstetter. On computing multiplicative inverses in $GF(2^m)$. *Computers, IEEE Transactions on*, 42(8):1010–1015, 1993.

[92] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics Doklady*, volume 7, page 595, 1963.

[93] C. H. Lim and H. S. Hwang. Fast implementation of elliptic curve arithmetic in $GF(p^n)$. In *Public key cryptography*, pages 405–421. Springer, 2000.

[94] M. Keller, T. Kerins, F. Crowe, and W. Marnane. FPGA implementation of a $GF(2^m)$ Tate pairing architecture. In *Reconfigurable Computing: Architectures and Applications*, pages 358–369. Springer, 2006.

[95] C. Shu, S. Kwon, and K. Gaj. FPGA accelerated Tate pairing based cryptosystems over binary fields. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 173–180. IEEE, 2006.

[96] M. Katagi, I. Kitamura, T. Akishita, and T. Takagi. Novel efficient implementations of hyperelliptic curve cryptosystems using degenerate divisors. In *Information Security Applications*, pages 345–359. Springer, 2005.

[97] V. Afanasyev. On the complexity of finite field arithmetic. In *5th Joint Soviet-Swedish Intern. Workshop on Information Theory,(Moscow, USSR)*, pages 9–12, 1991.

[98] T. Kerins, W. P. Marnane, E. M. Popovici, and P. S. L. M Barreto. Efficient hardware for the Tate pairing calculation in characteristic three. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 412–426. Springer, 2005.

[99] A. Byrne and W. P. Marnane. Versatile processor for $GF(p^m)$ arithmetic for use in cryptographic applications. In *Norchip Conference, 2006. 24th*, pages 281–284. IEEE, 2006.

[100] A. Byrne, F. Crowe, W. P. Marnane, N. Meloni, A. Tisserand, and E. Popovici. SPA resistant elliptic curve cryptosystem using addition chains. *International Journal of High Performance Systems Architecture*, 1(2):133–142, 2007.

[101] A. Byrne, N. Meloni, A. Tisserand, E. M. Popovici, and W. P. Marnane. Comparison of simple power analysis attack resistant algorithms for an elliptic curve cryptosystem. *Journal of Computers*, 2(10):52–62, 2007.

[102] G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar, and T. Wollinger. Efficient $GF(p^m)$ arithmetic architectures for cryptographic applications. In *Topics in Cryptology CT-RSA 2003*, pages 158–175. Springer, 2003.

[103] T. Kerins, W. P. Marnane, E. M. Popovici, and P. S. L. M. Barreto. Hardware accelerators for pairing based cryptosystems. *IEE Proceedings-Information Security*, 152(1):47–56, 2005.

[104] E. Gorla, C. Puttmann, and J. Shokrollahi. Explicit formulas for efficient multiplication in $\mathbb{F}_{3^{6m}}$. In *Selected Areas in Cryptography*, pages 173–183. Springer, 2007.

[105] P. Grabher and D. Page. Hardware acceleration of the Tate pairing in characteristic three. In *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 398–411. Springer, 2005.

[106] J. L. Beuchat, N. Brisebarre, J. Detrey, and E. Okamoto. Arithmetic operators for pairing-based cryptography. *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 239–255, 2007.

[107] S. Kwon. Efficient Tate pairing computation for elliptic curves over binary fields. In *Information Security and privacy*, pages 134–145. Springer, 2005.

[108] J. L. Beuchat, M. Shirase, T. Takagi, and E. Okamoto. An algorithm for the η_T pairing calculation in characteristic three and its hardware implementation. In *IEEE Symposium on Computer Arithmetic –Arith 2007*, pages 97–104. IEEE, 2007.

[109] J. L. Beuchat, N. Brisebarre, M. Shirase, T. Takagi, and E. Okamoto. A coprocessor for the final exponentiation of the η_T pairing in characteristic three. *Arithmetic of Finite Fields*, pages 25–39, 2007.

[110] P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected areas in cryptography*, pages 319–331. Springer, 2005.

[111] F. Hess, N. P. Smart, and F. Vercauteren. The eta pairing revisited. *Information Theory, IEEE Transactions on*, 52(10):4595–4602, 2006.

[112] E. Lee, H. S. Lee, and C. M. Park. Efficient and generalized pairing computation on abelian varieties. *Information Theory, IEEE Transactions on*, 55(4):1793–1803, 2009.

[113] D. Lubicz and D. Robert. Efficient pairing computation with theta functions. In *Algorithmic number theory*, pages 251–269. Springer, 2010.

[114] F. Vercauteren. Optimal pairings. *Information Theory, IEEE Transactions on*, 56(1):455–461, 2010.

[115] M. Naehrig, R. Niederhagen, and P. Schwabe. New software speed records for cryptographic pairings. In *Progress in Cryptology–LATINCRYPT 2010*, pages 109–123. Springer, 2010.

[116] J. L. Beuchat, J. E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal *Ate* pairing over Barreto–Naehrig curves. In *Pairing-Based Cryptography-Pairing 2010*, pages 21–39. Springer, 2010.

[117] D. F. Aranha, P. S. L. M. Barreto, P. Longa, and J. E. Ricardini. The realm of the pairings. In *Selected Areas in Cryptography–SAC 2013*, pages 3–25. Springer, 2013.

[118] A. Joux. A new index calculus algorithm with complexity $L(1/4+o(1))$ in small characteristic. In *Selected Areas in Cryptography–SAC 2013*, pages 355–379. Springer, 2013.

[119] R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *Advances in Cryptology–EUROCRYPT 2014*, pages 1–16. Springer, 2014.

[120] R. Granger, T. Kleinjung, and J. Zumbrägel. Breaking '128-bit secure' supersingular binary curves. In *Advances in Cryptology–CRYPTO 2014*, pages 126–145. Springer, 2014.

[121] A. Joux, A. Odlyzko, and C. Pierrot. The past, evolving present, and future of the discrete logarithm. In *Open Problems in Mathematics and Computational Science*, pages 5–36. Springer, 2014.

[122] C. H. Kim and J. J. Quisquater. Faults, injection methods, and fault attacks. *Design & Test of Computers, IEEE*, 24(6):544–545, 2007.

[123] M. Joye and M. Tunstall. *Fault Analysis in Cryptography*, volume 7. Springer, 2012.

[124] N. El Mrabet, J. J. A. Fournier, L. Goubin, and R. Lashermes. A survey of fault attacks in pairing based cryptography. *Cryptography and Communications*, 7(1):185–205, 2015.

[125] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology-CRYPTO'96*, pages 104–113. Springer, 1996.

[126] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology-CRYPTO99*, pages 388–397. Springer, 1999.

[127] Data Encryption Standard. National bureau of standards. Federal Information Processing Standards Publication 46.

[128] D. Page and F. Vercauteren. A fault attack on pairing-based cryptography. *Computers, IEEE Transactions on*, 55(9):1075–1080, 2006.

[129] C. Whelan and M. Scott. Side channel analysis of practical pairing implementations: Which path is more secure? In *Progress in Cryptology-VIETCRYPT 2006*, pages 99–114. Springer, 2006.

[130] M. Scott. Computing the Tate pairing. In *Topics in Cryptology–CT-RSA 2005*, pages 293–304. Springer, 2005.

[131] C. Whelan and M. Scott. The importance of the final exponentiation in pairings when considering fault attacks. In *Pairing-Based Cryptography–Pairing 2007*, pages 225–246. Springer, 2007.

[132] N. El Mrabet, M. L. Flottes, and G. Di Natale. A practical differential power analysis attack against the Miller algorithm. In *Research in Microelectronics and Electronics, 2009. PRIME 2009*, pages 308–311. IEEE, 2009.

[133] N. El Mrabet. What about vulnerability to a fault attack of the Miller's algorithm during an identity based protocol? In *Advances in Information Security and Assurance*, pages 122–134. Springer, 2009.

[134] K. Bae, S. Moon, and J. Ha. Instruction fault attack on the Miller algorithm in a pairing-based cryptosystem. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on*, pages 167–174. IEEE, 2013.

[135] R. Lashermes, J. Fournier, and L. Goubin. Inverting the final exponentiation of Tate pairings on ordinary elliptic curves using faults. In *Cryptographic Hardware and Embedded Systems-CHES 2013*, pages 365–382. Springer, 2013.

[136] M. Scott, N. Benger, M. Charlemagne, L. J. D. Perez, and E. J. Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In *Pairing-Based Cryptography–Pairing 2009*, pages 78–88. Springer, 2009.

[137] D. Hankerson, A. Menezes, and M. Scott. Software implementation of pairings. *Identity-Based Cryptography*, 2:188–206, 2009.

[138] D. F. Aranha, J. López, and D. Hankerson. High-speed parallel software implementation of the η_T pairing. In *Topics in Cryptology-CT-RSA 2010*, pages 89–105. Springer, 2010.

[139] P. Grabher, J. Großschädl, and D. Page. On software parallel implementation of cryptographic pairings. In *Selected Areas in Cryptography*, pages 35–50. Springer, 2008.

[140] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López. Faster explicit formulas for computing pairings over ordinary curves. In *Advances in Cryptology–EUROCRYPT 2011*, pages 48–68. Springer, 2011.

[141] T. Acar, K. Lauter, M. Naehrig, and D. Shumow. Affine pairings on ARM. In *Pairing-Based Cryptography–Pairing 2012*, pages 203–209. Springer, 2012.

[142] G. Grewal, R. Azarderakhsh, P. Longa, S. Hu, and D. Jao. Efficient implementation of bilinear pairings on ARM processors. In *Selected Areas in Cryptography*, pages 149–165. Springer, 2012.

[143] R. Azarderakhsh, D. Fishbein, G. Grewal, S. Hu, D. Jao, P. Longa, and R. Verma. Fast software implementations of bilinear pairings. *IEEE Transactions on Dependable and Secure Computing*, (99), 2015.

[144] M. Naehrig, R. Niederhagen, and P. Schwabe. Software library for pairing computation on AMD 64 processors. `https://www.cryptojedi.org/crypto/`.

[145] J. L. Beuchat, J. E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. Software library for high-speed implementation of the optimal *Ate* pairing over barreto-naehrig curves. `http://homepage1.nifty.com/herumi/crypt/ate-pairing.html`.

[146] R. M. Avanzi. Aspects of hyperelliptic curves over large prime fields in software implementations. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 148–162. Springer, 2004.

[147] K. Lauter, P. L. Montgomery, and M. Naehrig. An analysis of affine coordinates for pairing computation. In *Pairing-Based Cryptography-Pairing 2010*, pages 1–20. Springer, 2010.

[148] S. Baktir and B. Sunar. Optimal tower fields. *Computers, IEEE Transactions on*, 53(10):1231–1243, 2004.

[149] R. Schroeppel and C. Beaver. Accelerating elliptic curve calculations with the reciprocal sharing trick. *Mathematics of Public-Key Cryptography (MPKC), University of Illinois at Chicago*, 2003.

[150] C. Costello, H. Hisil, C. Boyd, J. G. Nieto, and K. K. H. Wong. Faster pairings on special Weierstrass curves. In *Pairing-Based Cryptography–Pairing 2009*, pages 89–101. Springer, 2009.

[151] A. Van Herrewege, L. Batina, M. Knezevic, I. Verbauwhede, and B. Preneel. Compact implementations of pairings. In *4th Benelux Workshop on Information and System Security-WISSec 2009*, pages 19–20, 2009.

[152] T. English, M. Keller, K. L. Man, E. M. Popovici, M. Schellekens, and W. P. Marnane. A low-power pairing-based cryptographic accelerator for embedded security applications. In *SOC Conference, 2009. SOCC 2009. IEEE International*, pages 369–372. IEEE, 2009.

[153] J. L. Beuchat, H. Doi, K. Fujita, A. Inomata, P. Ith, A. Kanaoka, M. Katouno, M. Mambo, E. Okamoto, T. Okamoto, et al. FPGA and ASIC implementations of the η_T pairing in characteristic three. *Computers & electrical engineering*, 36(1):73–87, 2010.

[154] N. Estibals. Compact hardware for computing the Tate pairing over 128-bit-security supersingular curves. In *Pairing-Based Cryptography-Pairing 2010*, pages 397–416. Springer, 2010.

[155] J. L. Beuchat, J. Detrey, N. Estibals, E. Okamoto, and F. Rodríguez-Henríquez. Fast architectures for the η_T pairing over small-characteristic supersingular elliptic curves. *Computers, IEEE Transactions on*, 60(2):266–281, 2011.

[156] S. Ghosh, D. Roychowdhury, and A. Das. High speed cryptoprocessor for η_T pairing on 128-bit secure supersingular elliptic curves over characteristic two fields. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 442–458. Springer, 2011.

[157] D. F. Aranha, J. L. Beuchat, J. Detrey, and N. Estibals. Optimal η pairing on supersingular genus 2 binary hyperelliptic curves. In *Topics in Cryptology–CT-RSA 2012*, pages 98–115. Springer, 2012.

[158] J. Adikari, A. Hasan, and C. Negre. Towards faster and greener cryptoprocessor for Eta pairing on supersingular elliptic curve over $\mathbb{F}_{2^{1223}}$. In *Selected Areas in Cryptography*, pages 166–183. Springer, 2012.

[159] T. English, E. M. Popovici, M. Keller, and W. P. Marnane. Network-on-chip interconnect for pairing-based cryptographic IP cores. *Journal of Systems Architecture*, 57(1):95–108, 2011.

[160] J. L. Beuchat, N. Brisebarre, J. R. Detrey, E. Okamoto, M. Shirase, and T. Takagi. Algorithms and arithmetic operators for computing the η_T pairing in characteristic three. *Computers, IEEE Transactions on*, 57(11):1454–1468, 2008.

[161] J. L. Beuchat, J. Detrey, N. Estibals, E. Okamoto, and F. Rodríguez-Henríquez. Hardware accelerator for the Tate pairing in characteristic three based on Karatsuba-Ofman multipliers. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 225–239. Springer, 2009.

[162] M. Cenk and F. Özbudak. Efficient multiplication in $\mathbb{F}_{3^{lm}}$, $m \geq 1$ and $5 \leq l \leq 18$. In *Progress in Cryptology–AFRICACRYPT 2008*, pages 406–414. Springer, 2008.

[163] H. Fan and A. Hasan. A new approach to subquadratic space complexity parallel multipliers for extended binary fields. *Computers, IEEE Transactions on*, 56(2):224–233, 2007.

[164] S. C. Chung, J. Y. Wu, H. P. Fu, J. W. Lee, H. C. Chang, and C. Y. Lee. Efficient hardware architecture of η_T pairing accelerator over characteristic three. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 23(1):88–97, 2015.

[165] R. Granger, D. Page, and M. Stam. On small characteristic algebraic tori in pairing-based cryptography. *LMS Journal of Computation and Mathematics*, 9:64–85, 2006.

[166] A. Barenghi, G. Bertoni, L. Breveglieri, and G. Pelosi. A FPGA coprocessor for the cryptographic Tate pairing over $\mathbb{F}_p$. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 112–119. IEEE, 2008.

[167] D. Kammler, D. Zhang, P. Schwabe, H. Scharwaechter, M. Langenberg, D. Auras, G. Ascheid, and R. Mathar. Designing an ASIP for cryptographic pairings over

Barreto-Naehrig curves. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 254–271. Springer, 2009.

[168] J. Fan, F. Vercauteren, and I. Verbauwhede. Faster $\mathbb{F}_p$-arithmetic for cryptographic pairings on Barreto-Naehrig curves. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 240–253. Springer, 2009.

[169] G. X. Yao, J. Fan, R. C. C. Cheung, and I. Verbauwhede. Faster pairing coprocessor architecture. In *Pairing-Based Cryptography–Pairing 2012*, pages 160–176. Springer, 2012.

[170] S. Ghosh, D. Mukhopadhyay, and D. Roychowdhury. Secure dual-core cryptoprocessor for pairings over Barreto-Naehrig curves on FPGA platform. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(3):434–442, 2013.

[171] Y. A. Chang, W. C. Hong, M. C. Hsiao, B. Y. Yang, A. Y. Wu, and C. M. Cheng. Hydra: An energy-efficient programmable cryptographic coprocessor supporting elliptic-curve pairings over fields of large characteristics. In *Advances in Information and Computer Security*, pages 174–186. Springer, 2014.

[172] M. Scott and P. L. S. M. Barreto. Compressed pairings. In *Advances in Cryptology–CRYPTO 2004*, pages 140–156. Springer, 2004.

[173] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[174] R. C. C. Cheung, S. Duquesne, J. Fan, N. Guillermin, I. Verbauwhede, and G. X. Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *CHES*, volume 6917, pages 421–441. Springer, 2011.

[175] M. Joye and S. M. Yen. The Montgomery powering ladder. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 291–302. Springer, 2002.

[176] G. R. Blakely. A computer algorithm for calculating the product AB modulo M. *IEEE Transactions on Computers*, (5):497–500, 1983.

[177] J. A. Akinyele, M. W. Pagano, M. D. Green, C. U. Lehmann, Z. N. J. Peterson, and A. D. Rubin. Securing electronic medical records using attribute-based encryption on mobile devices. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 75–86. ACM, 2011.

[178] M. Irfan and S. Yasin. A novel framework for securing medical records in cloud computing. *Int. J. Modern Eng. Res*, 3:2697–2699, 2013.

[179] E. Zavattoni, L. J. D. Perez, S. Mitsunari, A. H. Sánchez-Ramırez, T. Teruya, and F. Rodríguez-Henríquez. Software implementation of an attribute-based encryption scheme. *Computers, IEEE Transactions on*, 64(5):1429–1441, 2015.

[180] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology-CRYPTO 2001*, pages 190–200. Springer, 2001.

[181] M. Scott. Implementing cryptographic pairings. *Lecture Notes in Computer Science*, 4575:177, 2007.

[182] J. A. Solinas. ID-based digital signature algorithms. In *Slide Show presented at 7th Workshop on Elliptic Curve Cryptography (ECC 2003)*, 2003.

[183] A. De Caro and V. Iovino. jPBC: Java pairing based cryptography. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 850–855. IEEE, 2011.

[184] B. Lynn. PBC: Pairing-based cryptography library. `https://crypto.stanford.edu/pbc/`.

[185] S. D. Galbraith, K. G. Paterson, and N. P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

[186] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology-ASIACRYPT 2001*, pages 514–532. Springer, 2001.

[187] A. De Caro and V. Iovino. jPBC: Java Pairing-Based Cryptography library. `http://gas.dia.unisa.it/projects/jpbc/`.

[188] L. Malina, J. Hajny, and V. Zeman. Usability of pairing-based cryptography on smartphones. In *Telecommunications and Signal Processing (TSP), 2015 38th International Conference on*, pages 617–621. IEEE, 2015.

[189] F. Hess. Efficient identity based signature schemes based on pairings. In *Selected areas in cryptography*, pages 310–324. Springer, 2002.

[190] A. L. Ferrara, M. Green, S. Hohenberger, and M. O. Pedersen. Practical short signature batch verification. In *Topics in Cryptology–CT-RSA 2009*, pages 309–324. Springer, 2009.

[191] L. B. Oliveira, D. F. Aranha, C. P. L. Gouvêa, M. Scott, D. F. Câmara, J. López, and R. Dahab. TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. *Computer Communications*, 34(3):485–493, 2011.

[192] K. McCusker and N. E. O'Connor. Low-energy symmetric key distribution in wireless sensor networks. *Dependable and Secure Computing, IEEE Transactions on*, 8(3):363–376, 2011.

[193] C. P. L. Gouvêa, L. B. Oliveira, and J. López. Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. *Journal of Cryptographic Engineering*, 2(1):19–29, 2012.

[194] T. Unterluggauer and E. Wenger. Efficient pairings and ECC for embedded systems. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 298–315. Springer, 2014.

[195] P. S. L. M. Barreto, B. Libert, N. McCullagh, and J. J. Quisquater. Efficient and provably-secure identity-based signatures and signcryption from bilinear maps. In *Advances in Cryptology-ASIACRYPT 2005*, pages 515–532. Springer, 2005.

[196] L. Fuentes-Castañeda, E. Knapp, and F. Rodríguez-Henríquez. Faster hashing to $\mathbb{G}_2$. In *Selected Areas in Cryptography*, pages 412–430. Springer, 2011.

[197] C. L. Chen, T. F. Shih, Y. T. Tsai, and D. K. Li. A bilinear pairing-based dynamic key management and authentication for wireless sensor networks. *Journal of Sensors*, 2015, 2015.

[198] Z. Cheng. Implementing pairing-based cryptosystems in USB tokens. *IACR Cryptology ePrint Archive*, 2014:71, 2014.

[199] Apple Inc. The Swift Programming Language. `https://developer.apple.com/swift/`.

[200] The LLVM Project. The LLVM Compiler Infrastructure. `http://llvm.org/`.

[201] IBM Inc. The Kitura Web-based framework. `https://developer.ibm.com/swift/products/kitura/`.

[202] R. H. Jacobsen, S. A. Mikkelsen, and N. H. Rasmussen. Towards the use of pairing-based cryptography for resource-constrained home area networks. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 233–240. IEEE, 2015.

[203] X. Chen, W. Susilo, J. Li, D. S. Wong, J. Ma, S. Tang, and Q. Tang. Efficient algorithms for secure outsourcing of bilinear pairings. *Theoretical Computer Science*, 562:112–121, 2015.