12-4-2015

Efficient Implementations of Pairing-Based Cryptography on Embedded Systems

Rajeev Verma
rv4560@rit.edu

Efficient Implementations of Pairing-Based Cryptography on Embedded Systems

by

Rajeev Verma

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree
of
Master of Science in Computer Engineering

Supervised by

Dr. Reza Azarderakhsh
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology, Rochester New York
December 4, 2015

Approved By:

Dr. Reza Azarderakhsh
Department of Computer Engineering, R.I.T
Primary Advisor

Dr. Marcin Lukowiak
Department of Computer Engineering, R.I.T
Secondary Advisor

Dr. Mehran Mozaffari Kermani
Department of Electrical and Microelectronics Engineering, R.I.T
Secondary Advisor

I would like to dedicate this thesis to the memories of my mother and sister. I would also like to dedicate this thesis and give special thank to my family to always support me in every possible way.

Acknowledgements

I would like to thank my supervisor Dr. Reza Azarderakhsh for his invaluable guidance throughout the process of creating this thesis. I would like to thank my readers Dr. Marcin Lukowiak and Dr. Mehran Mozaffari Kermani for their helpful feedback.

Abstract

Many cryptographic applications use bilinear pairing such as identity based signature, instance identity-based key agreement, searchable public-key encryption, short signature scheme, certificate less encryption and blind signature. Elliptic curves over finite field are the most secure and efficient way to implement bilinear pairings for the these applications. Pairing based cryptosystems are being implemented on different platforms such as low-power and mobile devices. Recently, hardware capabilities of embedded devices have been emerging which can support efficient and faster implementations of pairings on hand-held devices. In this thesis, the main focus is optimization of Optimal Ate-pairing using special class of ordinary curves, Barreto-Naehring (BN), for different security levels on low-resource devices with ARM processors. Latest ARM architectures are using SIMD instructions based NEON engine and are helpful to optimize basic algorithms. Pairing implementations are being done using tower field which use field multiplication as the most important computation. This work presents NEON implementation of two multipliers (Karatsuba and Schoolbook) and compare the performance of these multipliers with differnt multipliers present in the litrature for different field sizes. This work reports the fastest implementation timing of pairing for BN254, BN446 and BN638 curves for ARMv7 architecture which have secrity levels as 128-, 164-, and 192-bit, respectively. This work also presents comparison of code performance for ARMv8 architectures.

Contents

[1] NEON implementations of differnt algorithms can be found here.

List of Figures

List of Tables

Chapter 1

Introduction

Cryptography is a key technology for achieving information security in computer systems, electronic commerce, and in the emerging information security systems. Elliptic curve cryptography [1] are advantageous among public key cryptosystems for its faster key generation, shorter key size for same security level compared to RSA and low on CPU and memory consumption. The discrete logarithm Problem (DLP) is intractable for some group of points on elliptic curve defined over a finite field. Intractability of Diffie-Hellman problem (DHP) [2] is the basis of Diffie-Hellman key agreement protocol which allow two parties (Alice and bob) to establish a shared secret key by communicating over a public channel that is being monitored by eavesdropper (Eve). This protocol is efficient to share key among two parties in one round but if we have three parties to share the key over a public channel Diffie-Hellman key agreement protocol takes two step. Antoine Joux [3] devised a simple protocol to share the key between three parties in one round using pairings. Three party key exchange protocol using pairing is shown in in Figure 1.1. Alice, Bob and Chris have private keys as a,b,c and calculate aP, bP and cP using scalar multiplication over elliptic curve and share these values over public channel.

Finally all three parties calculate shared key in only one round as:

$$\begin{aligned} e(bP,cP) &= e(P,P)^{abc}(Alice) \\ e(aP,cP) &= e(PP)^{abc}(Bob) \\ e(aP,bP) &= e(P,P)^{abc}(Chris) \end{aligned}$$

where e(a,b) is pairing computation on curve. Elliptic curve discrete logarithm problem defines that if Eve get access to P, aP, bP and cP, it is not feasible to calculate the key. Later,

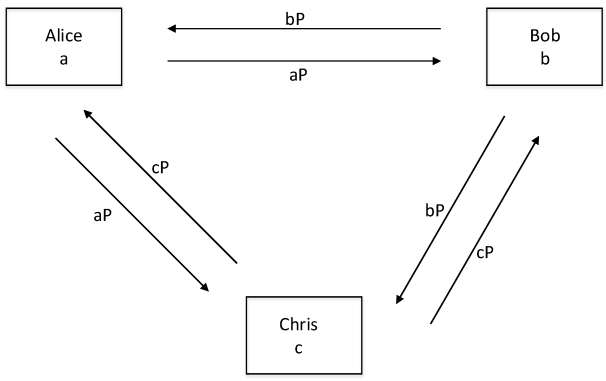

Figure 1.1: Tripartite Diffie–Hellman protocol over Elliptic curves.

identity based encryption scheme explained by Boneh and Franklin [4] increased the popularity of pairing based cryptography. Many pairing based cryptosystems are being proposed following Boneh and Franklin which would be difficult to design using conventional cryptograhic primitives. For instance [5] identity-based key agreement, searchable public-key encryption, short signature scheme, certificate less encryption, blind signature, attribute based encryption [6], are some of the interesting applications of pairing based cryptosystems. Elliptic curve cryptosystems provides relatively small block size over other cryptosystems, high-security public key schemes that can be efficiently implemented. Elliptical curves over finite field [7] are most secured and efficient way to implement bilinear pairings for these applications.

Pairing based cryptosystems are being implemented on different platforms such as low-power and mobile devices. Recently, hardware capabilities of embedded devices have been emerging which can support efficient and faster implementations of pairings. The basic idea of pairings is the construction of a mapping between two cryptographic groups which allows a cryptographic scheme based on reduction of one complex problem in one group to a different and easier problem in another group. The known implementations of these pairings – the Weil [8], ate [9], Tate and Optimal-ate [10] pairing, – involve fairly complex mathematics. All pairing based applications use a pairing-friendly elliptic curve of prime numbers. There are different coordinate systems [11] can be used to represent points on elliptic curves such as, Jacobian, Affine and Homogeneous. Inversion to multiplication ratio threshold can be used to decide the efficiency of coordinate system [12]. In this work timing results of pairing is being reported for both affine and projective coordinates using BN-curve [13] which is widely used curve for pairing-based algorithms. All fast algorithms to compute pairings on elliptic curves are based on Miller's algorithm [8] , [14].

Two type of implementations of pairing have been appeared in the literature, [15] [16]

[17] and [18] [12] [19] for PC and ARM processor based devices respectively. Computation of pairings over binary extension fields, i.e., $\mathbb{F}_{2^m}$ is not attractive anymore due to the recent attack presented in [20]. So in this work, we are focused on implementation of pairing on prime fields for low-resource devices using ARM processors. ARM processors are widely used core support for smart phones and tablets class CPU's. ARM has introduced NEON engine in ARMv6 onward. NEON is general purpose Single Instruction Multiple Data (SIMD) engine which can accelerate signal processing for pairing based protocols [21]. A. H. Sánchez and F. Rodríguez-Henríquez [19] for the first time used NEON to improve the timing results for pairing on ARM based devices but this work is only for 254-bit (128-bit security level). To date, there is no work available on higher security levels such as 446-bit and 638-bit which is using NEON engine to improve the pairing timings on low resource devices. General implementation of pairing-based algorithms uses tower like structure for computations in higher fields so lower level arithmetic computations such as addition, multiplication, squaring, inversion etc. are crucial to improve the timings for higher level algorithms. Among them multiplication plays an important role to determine the efficiency of pairings.

We have different multipliers present in literature, which can be used for lower level multiplication for pairing based algorithm other than multiplier in GMP library. One can refer to [22] for revisited montgomery multiplier which is NEON based implementation of Montgomery Multiplier [23]. BN446 and BN638 curve computations require 446-bit and 638-bit multiplication which is not efficient using revisited Montgomery as the algorithm uses the operands with field size as 256-bit, 512-bit. Another implementation is [24] which uses NEON for implementation of modified version of Karatsuba multiplier [25]. A. H. Sánchez and F. Rodríguez-Henríquez [19] uses NEON based implementation of Montgomery multiplier in higher field. In this work we present the implementation of Schoolbook [26] and Karatsuba multiplier [25] using NEON and ASM instructions for different fields and compare their performance with above multipliers.

This work[1] optimize the pairings algorithm implementations using NEON engine for hand-held devices based on ARM architectures. In this work, we optimize the implementation of pairings presented in [12] for different BN curves such as, BN254, BN446 and BN638. We present a comparison between different multipliers present in the literature and present the timing comparision of pairings computations using each multiplier. Timings are being measured for both affine and projective coordinates for O-Ate pairings for different security levels. Our work present 6% improvement over the previous NEON based best timing for BN254 curve [19]. We also present the timing improvement of pairing computations for BN446 and BN638

[1] NEON implementations of differnet algorithms can be found here

which are 45 and 50% faster than previous fastest implementations for same field size [12].

The remainder of this work is organized as follows. In Chapter 2 we introduce the Preliminaries of Pairings, explain the basics about pairings and their applications, elliptic curves and different pairings. In Chapter 3, we introduce basic computation algorithm in prime field such as, addition, multiplication and reduction. In Chapter 4, we give a brief overview of NEON engine in different ARM architecture and different methods of NEON implementation. In Chapter 5, we present our implementations of multipliers using NEON and different optimization at higher level algorithm of pairings. In chapter 6, we present the comparison of different multipliers in literature for different field size and present the improved timing results for pairings using different multipliers for Assembly and NEON implementations. In chapter 7, we present our conclusion and some opportunities for future research.

Chapter 2

Preliminaries of Pairings

In this chapter, we discuss about basics of pairings, their applications and implementations using elliptic curves.

2.1 Bilinear Pairings

Let G_1 and G_2 be cyclic groups of prime order n written additively with identity ∞, and let G_3 be a cyclic group of order n written multiplicatively with identity 1.

Definition 1. A bilinear pairings e can be defined as:

$$e : G_1 \times G_2 \to G_3$$

that satisfies the following additional properties:
(1) Bilinearity: For all $P, P' \in G_1$ and $Q, Q' \in G_2$,

$$\begin{aligned} e(P+P',Q) &= e(P,Q)e(P',Q), \\ e(P,Q+Q') &= e(P,Q)e(P,Q'). \end{aligned}$$

(2) Non-degeneracy: $e(P,P) \neq 1$

Additionally, one wants e to be efficiently computable. From the two properties above, we can have several other useful properties of e.

1. $e(P,\infty) = e(\infty,Q) = 1$.
2. $e(P,-Q) = e(-P,Q) = e(P,Q)^{-1}$.

3. $e(aP,bQ) = e(P,Q)^{ab}$ *for all* $a,b \in \mathbb{Z}$.
4. $e(P,Q) = e(Q,P)$ $(only\ if\ G_1 = G_2)$.
5. If $e(P,Q) = 1$ for all $Q \in G_2$ then $P = \infty$.

For detail explaination one can refer [27].

2.2 Finite Fields

A field [27] is a set that contains specific elements and equipped with two binary operations, +(addition) and . (multiplication), which has distinct additive and multiplicative identities, admit additive and multiplicative inverse, and satisfy associative, commutative and distributive laws. For example $\mathbb{Q}$ (rational numbers), $\mathbb{R}$ (real numbers), $\mathbb{C}$ (complex numbers) etc.

A *finite field* [7] is a field with a finite number of elements which has size equal to p^m, for some prime p. There is exactly one finite field of size $q = p^m$ for each pair (p,m) and we denote this field as $\mathbb{F}_{p^m}$ or $\mathbb{F}_q$ which is called as *Galois field* and also denoted as $GF(q)$. When $q = p^m$ is a prime power, the field $\mathbb{F}_{p^m}$ can be obtain by taking the set $\mathbb{F}_p[X]$ of all the polynomials in X with coefficients in $\mathbb{F}_q$ modulo any single irreducible polynomial of degree m.

Elliptic curves over finite fields are of special interest in implementation of cryptographic algorithms. There is no other known faster algorithm than $O(\sqrt{n})$ for computing discrete logarithms on the elliptic curve which is represented by group of points. In other words elliptic curves provide theoretical maximum possible level of security in public key cryptographic applications.

2.3 Elliptic curve Arithmetic

Elliptic curves group is a good choice for implementation of pairing-based cryptographic algorithms because of the simplicity of implementation as explained in [27]. An elliptic curve is the set of points satisfying an equation in two variables with degree two in one of the variables and three in the other.

Definition 2. Let $\mathbb{F}$ is a field with characteristics not equal to 2 or 3. An elliptical curve E over $\mathbb{F}$ is defined as set of solutions of the Weierstrass equation:

$$E : Y^2Z = X^3 + aXZ^2 + bZ^3 \tag{2.1}$$

where $a, b \in \mathbb{F}$.

The discriminant of E is defined as:

$$\triangle = 4a^3 + 27b^2$$

An elliptic curve is defined as "smooth" if there is no point at which the curve has two or more distinct tangent lines and $\triangle \neq 0$ is the condition to ensure that. In this thesis we will assume that the chosen curve is "smooth".

In Equation 2.1, if we set $Z = 0$ the result will be $X^3 = 0$ which has solution as $[0:1:0]$. For elliptic curve this point is called as the point at infinity and denoted as ∞.

We consider simplified Weierstrass equation with $Z \neq 0$ which is defined as:

$$y^2 = x^3 + ax + b \tag{2.2}$$

So, the elliptic curve can be defined as set of solutions to the equation 2.2 and also the point at infinity. We can also define elliptic curve as set of all $\mathbb{K}$-rational points on E where $\mathbb{K}$ is an extension of $\mathbb{F}$ as:

$$E(\mathbb{K}) : \{(x,y) \in \mathbb{K} \times \mathbb{K} : y^2 = x^3 + ax + b\} \cup \{\infty\}$$

Elliptic Curve Discrete Logarithm Problem (ECDLP) is defined as: given points P and Q in the group, find a number k such that $Pk = Q$ which is called scalar multiplication for elliptic curve. With known P and Q, one must guess at least the square root of the number of points on average to fine the value of k. If P and Q are two point on the elliptic curve, the elliptic curve point addition (ECADD) is defined as $R = P + Q$ and elliptic curve point doubling is defined as $Q = 2P$.

2.4 The Group Low

Let E be an elliptic curve defined over the field K. Chord-and-tangent rule is being used to add two points in $E(K)$ to give a third point in the same $E(K)$. This addition operation with set of points $E(K)$ and ∞ forms an abelian group to construct elliptic curve cryptographic systems. Let P and Q be two distinct points on curve E, then the group low is described as:

1. *Identity*. $P + \infty = \infty + P = P$ for all $P \in E(K)$.

2. *Negatives*. If $P = (x,y) \in E(K)$, then $(x,y) + (x,-y) = \infty$. The point $(x,-y)$ is called negative of P and denoted as $-P$ which is also a point on $E(K)$. Also $-\infty = \infty$.

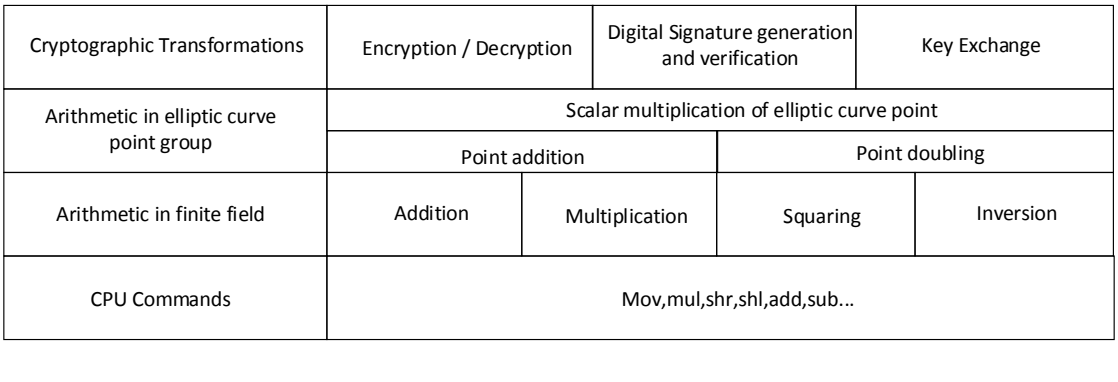

Figure 2.1: Operation flow of Elliptic Curve based cryptosystem.

3. *Point addition*. Let $P = (x_1, y_1) \in E(K)$ and $Q = (x_2, y_2) \in E(K)$, where $P \neq \pm Q$. Then $P + Q = (x_3, y_3)$ where

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ and } y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1.$$

4. *Point doubling*. Let $P = (x_1, y_1) \in E(K)$, where $P \neq -P$. Then $2P = (x_3, y_3)$,where

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ and } y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1.$$

Figure 2.1 shows the basic elliptic curve based cryptographic systems. The top layer shows the applications which uses elliptic curve point groups operations as point addition and point doubling. Elliptic curve uses arithmetic operations in finite field and the main operations are addition, multiplication, squaring and inversion. These arithmetic operations can be implemented in many ways and languages such as, C or Assembly. Lowest level uses CPU commands or instruction such as, MOV, MUL, SHR, SHL etc. To optimize the highest application layer, lower layers should be optimized using different algorithms or implementation languages.

2.5 The Tate Pairing

The Tate pairing is an example of bilinear pairings defined on the group of points over an elliptic curve for a finite field. Let E is and elliptical curve over the finite field $\mathbb{F}_q$ and the order of E is hn where n is a prime such that its not equal to the characteristics of $\mathbb{F}_q$ and h and n are co-prime. We define μ_n as $\mu_n = u \in \bar{\mathbb{F}}_q : u^n = 1$ which is a group of $nth - root$ of unity.

Definition 3. Definition of the field $F = \mathbb{F}_q(\mu_n)$ as the extension of the field $\mathbb{F}_q$ generated by the nth roots of unity. If k is the degree of this extension then k is called the embedding degree of E with respect to n. In other words embedded degree is the smallest value of integer k such that $n \mid q^k - 1$.

We can define Tate pairing $e(P,Q)$ as

$$e(P,Q) = f_P(D_Q)^{\frac{q^k-1}{n}} = \left(\frac{f_P(Q+R)}{f_P(R)}\right)^{\frac{q^k-1}{n}}$$

It can be observed that the value of Tate pairing does not depend on either function f_P or D_Q, so it is well defined and it is bilinear and non-degenerate [28]. It is shown in [28] that for $k > 1$, D_Q can be replaced with Q which redefines Tate pairing as $e(P,Q) = f_P(Q)^{\frac{q^k-1}{n}}$ where f_P is called Miller function and we can use Miller algorithm to calculate this function.

2.6 Miller Algorithm

Miller's algorithm [8] is the most efficient algorithm to evaluate at a certain point a function assiciated with prinipal divisor. In a general method, any function $f_{n,P}$ with the divisor $n(P) - (iP) - (i-1)(\infty)$ is called the Miller function. Miller function is the key component of Tate pairing computation. The key idea of Miller's algorithm is to construct the function $f_{n,P}$ using doubling and addition.

Miller algorithm construct f_n using double-and-add approach and $f_1 = 1$. The function f_n has divisor as $n(P) - ([n]P) - (n-1)\infty = n(P) - n(\infty)$. As explained in Algorithm 2.1, Miller algorithm uses loop (step 2 to step 9) which is called Miller Loop. In order to compute Tate pairing, Miller algorithm computes the value of $f_n(Q)$ instead of computing all the values for each step as f_i which makes it easy to compute. The step-10 of the algorithm is called final exponentiation which is another important function to decide computation timing of pairings.

Algorithm 2.1 Miller's Algorithm for the O-Ate pairing [8].

Inputs: Points $Q, P \in E[n]$ and integer
$n = (n_{l-1}, n_{l-2}, \cdots, n_1, n_0)_2 \in \mathbb{N}$ and $n_{l-1} = 1$
Output: $f_{n,Q}(P)$.
1: $T \leftarrow Q,\ f \leftarrow 1$
2: **for** $i = l-2$ **downto** 0 **do**
3: $\quad f \leftarrow f^2 \cdot l_{T,T}(P)$
4: $\quad T \leftarrow 2T$
5: $\quad$ **if** $l_i \neq 0$
6: $\quad\quad f \leftarrow f \cdot l_{T,P}(P)$
7: $\quad\quad T \leftarrow T + Q$
8: $\quad$ **end if**
9: **end for**
10: $f \leftarrow f^{\frac{q^k-1}{n}}$
11: **return** f

2.7 Applications

The bilinear pairings such as Tate pairing or Weil pairings on elliptical curves have been very efficient in cryptographic applications. Bilinear pairings has been used for implementing Several ID-based cryptosystems. Certificate-based public key infrastructure (PKI) can be replaced by the ID-based public key cryptosystem especially when moderate security and efficient key-management are required. These are some basic application based on bilinear pairing:

2.7.1 Tripartite Diffie-Hellman

Joux [3], use the pairings in the development of the three-party, one-round key-exchange protocol. This protocol is a three-party version of Diffie-Hellman and its security is based on the following assumption:

Definition 4. Let e be a bilinear pairings $e : G_1 \times G_1 \rightarrow G_2$with $P \in G_1$. Given P, aP, bP, cP, it is computationally infeasible to compute $e(P,P)^{abc}$.

These are steps for protocol:

1. The three parties agree on a common point $P \in G_1$.
2. Each party chooses a secret integer a, b, c and broadcasts aP, bP, cP .
3. Bilinearity of the pairings allows each party to compute a common secret key. Indeed,

$$e(P,P)^{abc} = e(aP,bP)^c = e(aP,cP)^b = e(bP,cP)^a.$$

2.7.2 Co-GDH Signature Scheme on Elliptic Curves

Lets G_1, G_2 are subgroups of the elliptic curve $E/\mathbb{F}_q$, and assume there is a mapping which can be used to place the message M as a point on the elliptic curve E. In other words, we can say M is being represented as a point on elliptic curve G_1. Further discussion can be found in [29].

Private Key: Alice randomly selects an integer $x \xleftarrow{R} \mathbb{Z}_q$. x is secret key as $x \in \mathbb{Z}_q$

Public Key: Compute public key is $V = xQ$ where Q generates G_2.

Signature: Let $M \in G_1$ be the message and compute $\sigma \leftarrow xM \in E(\mathbb{Z}_q)$ where $h \in G_1$. The x-coordinate of σ is the signature S on M so $S \in \mathbb{F}_q$

Verification: Given a public key $V \in G_2$, a message $M \in G_1$ and signature $s \in \mathbb{F}_q$, find $a, y \in \mathbb{F}_q$ such that $\sigma = (s,y)$ is a point which has order p in $E(\mathbb{F}_q)$.

If the signature is valid then we would have

$$e(\sigma,Q) = e(M,V) \text{ or } e(\sigma,Q)^{-1} = e(M,V)$$

as required for verification. This signature is efficient as it requires only one point on elliptic curve which is half of the size of DSA signature.

2.7.3 BLS Short Signatures on Elliptic curves

Signature schemes are another use of pairings. Most discrete logarithm signature schemes are variants of the ElGamal scheme. Boneh, Lynn and Shacham presented a signature scheme which uses one group element as the signature and groups elements are represented as the same number of bits as an integer modulo n which is called as BLS short signature scheme [30].

Let e be a bilinear pairings as $e : G_1 \times G_1 \to G_2$ where $G_1 =< P >$ be the prime order n subgroup generated by point $P \in E(\mathbb{F}_q)$ then also $G_1 \in E(\mathbb{F}_{q^k})$ where k is the embedding degree of Q and G_2 is a prime order subgroup $G_2 \in E(\mathbb{F}_{q^k})$ with linear independent points of the ones in group G_1. Let's H be a cryptographic hash function $H : \{0,1\}^* \to G_1$. The general BLS algorithm works as follows:

Private Key: Alice randomly selects an integer $x \xleftarrow{R} \mathbb{Z}_q$. x is secret key as $x \in \mathbb{Z}_q$

Public Key: Compute $A = xP$ where $A \in G_2$.

Signature: Let $m \in \{0,1\}^*$ be the message and compute $h \leftarrow H(m)$ where $h \in G_1$. Then $S = xh$ is the signature.

Verification: Compute $h = H(m)$. Then verify that $e(P,S) = e(A,h)$.

If the signature is valid then we would have

$$e(P,S) = e(P,aH(m)) = e(aP,H(m)) = e(A,h)$$

as required for verification.

2.8 Tower Extension Field Arithmetic

Cryptosystems based on elliptic curve can be so generic that can work for different point size on curves such as, 128-bit, 164-bit and 192-bit. When using pairings based protocols, its necessary to perform arithmetic in higher fields such as, F_{q^k} for moderate value of k. It is important to represent the field in such a way that arithmetic can be performed in efficient way. One of the most efficient way is to use a tower of extension field [31] which explains that higher level computations can be calculated as a function of lower level computations so that efficient implementation of lower level algorithms can impact the performance of algorithms in higher fields.

Miller Algorithm is being executed using arithmetic in $F_{q^{12}}$ field during the accumulation step of the algorithm. Extension field arithmetic are very important to improve the performance of pairing. $\mathbb{F}_{q^k}$ should be represented with the tower of extensions with the use of irreducible binomials as explained in [32] and can be expressed as:

$$\begin{aligned}
\mathbb{F}_{q^2} &= \mathbb{F}_q[i]/(i^2-\beta), \textit{where } \beta = -1 \\
\mathbb{F}_{q^4} &= \mathbb{F}_{q^2}[s]/(s^2-\xi), \textit{where } \xi = i+1 \\
\mathbb{F}_{q^6} &= \mathbb{F}_{q^2}[u]/(u^3-\xi), \textit{where } \xi = i+2 \\
\mathbb{F}_{q^{12}} &= \mathbb{F}_{q^4}[t]/(t^2-s) \textit{ or } \mathbb{F}_{q^6}[w]/(w^2-u)
\end{aligned}$$

The conversion from one towering $\mathbb{F}_{q^2} \rightarrow \mathbb{F}_{q^6} \rightarrow \mathbb{F}_{q^{12}}$ to another $\mathbb{F}_{q^2} \rightarrow \mathbb{F}_{q^4} \rightarrow \mathbb{F}_{q^{12}}$ is simply possible by permuting the order of coefficients. The choice of $p \equiv 3 (\mathrm{mod}\, 4)$ accelerates arithmetic in F_{q^2} , since multiplications by $\beta = -1$ can be computed simply by substation.

2.9 Barreto-Naehrig Elliptic Curves

Pairings based cryptography requires pairing-friendly curves [33] and these curves are parameterised by the factor embedding degree k. Pairing definitions show that randomly chosen elliptic curve might be suitable for implementing pairing-based protocols. But if the embedding degree of the curve E is very large, it is not possible to implement field arithmetic in the size of F_{q^k}. For the implementation of elliptic curve the embedding degree, with respect to n, should be small enough so that curve arithmetic in extension field should be easy to implement yet large enough that Discrete Logarithmic Problem is intractable in $\mathbb{F}_{q^k}$. Since the larger subgroups provide high level of security so the elliptic curve E should have a large prime order subgroup for arithmetic implementation of the curve. Elliptic curves with suitably low embedding degree are very rare. Balasubramaniam and Koblitz [34] showed that one can expect $k \approx q$ for a randomly selected prime-order elliptic curve over a randomly selected prime-order field and also the probability that $k \leq \log^2 q$ is vanishing small.

There are many methods to generate elliptic curve in literature. David Freeman [33] showed the comprehensive review of the different curves. Barreto and Narhrig [13] devised a method to generate elliptic curve which supports pairings over prime field with prime order $k = 12$ which is called as Barreto-Narhrig or BN-curve. We have used BN-curves for our implementations as BN-curve is suitable to achieve high security and efficiency of cryptographic algorithms. BN-curves enable all kind of pairing-based cryptographic schemes and protocols (including short signatures) [35]. In this work we are focusing on implementation of pairings on BN-curve which has prime order and every point in the curve has order n. The equation of BN-curve is $E : y^2 = x^3 + b$, with $b \neq 0$. The trace of the curve, the characteristic of F_q and the curve order are parameterised as:

$$\begin{aligned} t(x) &= 6x^2 + 1 \\ n(x) &= 36x^4 + 36x^3 + 18x^2 + 6x + 1 \\ q(x) &= 36x^4 + 36x^3 + 24x^2 + 6x + 1 \end{aligned}$$

Pairings on BN-curves are computed over points in $E(\mathbb{F}_{q^{12}})$ because BN-curves have embedding degree as 12 so the pairings on a BN curve over a 256-bit prime field $\mathbb{F}_q$ takes its values in the field $\mathbb{F}_{q^{12}}$ with size $256 \times 12 = 3072$. To construct a pairing-friendly elliptic curve using BN method, Choose randomly an integer x of appropriate size such that both polynomials $q(x)$ and $n(x)$ evaluate to prime numbers and let $q = q(x)$. Let $b \in \mathbb{F}_q^*$ such that

$b+1$ is a quadratic residue. Furthermore, $P = (1, \sqrt{b+1})$ is a point on curve E that can be used as generator for $E(\mathbb{F}_q)$. As mentioned in [36], BN-curves are ideal for implementation of optimized variant of Tate pairing which is referenced as O-Ate pairing.

2.10 The Ate Pairing

Pairings can be computed in polynomial time using Miller's algorithm as explained in 2.6. Many other techniques have been suggested for optimizing the computation of pairings. Among those, one of the most elegant technique is to shorten the iteration loop in Miller's algorithm to compute pairings efficiently. The Ate Pairing is referred as optimized version of Tate Pairing in which Miller loop is designed to be shorter than that of used in the Tate pairing. We derive the Ate Pairing using [10] and further explained Optimal-Ate pairing.

To minimize the number of addition steps in Miller's algorithm, we choose n to have a low Hamming weight. In Miller loop, the computation of $nP = \infty$ is being done using double-and-add method which can be simplified choosing P such that it's coordinates lie in a sub-field of $\mathbb{F}_{q^k}$. As mentioned in above section, we can choose $P = (1, \sqrt{b+1})$ as the generator of the subgroup. In Ate pairing, the parameters are restricted to Frobenius eignespace. We will choose the first parameter $Q \in G_2$ and second parameter $P \in G_1$.

Lemma 5. *[27]* $f_{ab,Q} = f_{a,Q}^{b} \cdot f_{b,aQ}$ *for all* $a, b \in \mathbb{Z}$.

This leads to following lemma [10].

Lemma 6. $e(Q,P)^m = f_{mn,Q}(P)^{\frac{q^k-1}{n}}$ *where* $e(Q,P)$ *is the Tate Pairing with* $Q \in G_2$, $P \in G_1$ *and* $m \in \mathbb{Z}$.

which defines the pairings on the curve. The following fact defines the pairings in simpler way:

Fact 7. *[10]* $f_{a,\Pi q(Q)}(P) = f_{a,Q}(P)^q$ *for all* $a \in \mathbb{Z}$ *and* $Q \in G_2$.

Using above fact we can get $e(Q,P)^m = f_{\lambda,Q}(P)^{\frac{q^k-1}{n} k q^{k-1}}$.

Since n and q are prime, we have that $n \nmid q$ and $n \nmid k$. Thus we can define:

$$a(Q,P) = e(Q,P)^{m((k^{-1}q^{-(k-1)}) \bmod n)} = f_{\lambda,Q}(P)^{\frac{q^k-1}{n}}.$$

The above expression represent the Ate Pairing [10]. $m((k^{-1}q^{-(k-1)}) \bmod n)$ is relatively prime to n which makes the above expression bilinear and non-degenerate. For BN-curves,

$q(x) \cong 6x^2 (\mathrm{mod}\, n(x))$ so we take $\lambda = 6x^2$. This reduces the Miller loop length from $\log(36x^4 + 36x^2 + 18x^2 + 6x + 1)$ to $\log(6x^2)$.

There is further possibility to optimize the Miller loop as sown in [10] which is called Optimal Ate or O-Ate pairing.

2.11 The Optimal Ate-Pairing

The Optimal Ate or O-Ate pairing is an improved version of Ate pairing. Let's consider the m^{th} power of Tate Pairing and $\sigma = mn$ and suppose $\sigma = \sum_{i=0}^{l} c_i q^i$ is the base-q expansion of σ. If $s_i = \sum_{j=i}^{l} c_j q^j$, we have:

$$e(Q,P)^m = (\prod_{i=0}^{l} f_{q^i,Q}^{c_i}(P)^{\frac{q^k-1}{n}})(\prod_{i=0}^{l} f_{c_i,Q}^{q^i}(P) \prod_{i=0}^{l-1} \frac{l_{[c_i q^i]Q, s_{i+1}Q}(P)}{v_{s_i,Q}(P)})^{\frac{q^k-1}{n}}$$

The left side of above expression is bilinear pairing. The right side of expression is a product of powes of Ate pairing so it is also bilinear. The factors of second set should also be a bilinear pairing. As explained in [10] let $\sigma = mn$ with $n \nmid m$ then:

$a_O : G_2 \times G_1 \to \mu_n$ given by:

$$(Q,P) \mapsto (\prod_{i=0}^{l} f_{c_i,Q}^{q^i}(P) \prod_{i=0}^{l-1} \frac{l_{[c_i q^i]Q, s_{i+1}Q}(P)}{v_{s_i,Q}(P)})^{\frac{q^k-1}{n}}$$

defines a bilinear pairings called the O-Ate pairing. Furthermore if $mkq^{-1} \not\equiv \frac{q^k-1}{r} \sum_{i=0}^{l} ic_i q^{i-1} \mathrm{mod}\, n$ then the pairing is non-degenerate.

The O-Ate Pairings on BN-Curves

From BN-curve definition:

$$\begin{aligned} Q(x) &= 36x^4 + 36x^3 + 24x^2 + 6x + 1 \\ N(x) &= 36x^4 + 36x^3 + 18x^2 + 6x + 1 \end{aligned}$$

We can also represent $Q(x)$ as:

$$\begin{aligned} Q(x) &= 6x^2 \bmod N(x), \\ Q(x)^2 &= 36x^3-18x^2-6x-1 \bmod N(x), \\ Q(x)^3 &= 36x^3-24x^2-12x-3 \bmod N(x). \end{aligned}$$

Therefore,

$$6x+2+Q(x)-Q(x)^2+Q(x)^3 = 0 \bmod N(x). \tag{2.3}$$

Lets $\sigma = 6x+2+Q(x)-Q(x)^2+Q(x)^3$ which gives the following expression for O-Ate pairing on BN-curves:

$$(Q,P) \mapsto f_{6x+2,Q}(P) f_{1,Q}^{q}(P) f_{-1,Q}^{q^2}(P) f_{1,Q}^{q^3}(P) g(P).$$

We know $f_{1,Q} = f_{-1,Q} = 1$, so we can ignore these in above expression. The expression for $g(P)$ is given by:

$$g(P) = l_{[6x+2]Q,[q-q^2+q^3]Q}(P) l_{[q]Q,[-q^2+q^3]Q}(P) l_{[-q^2]Q,[q^3]Q}(P).$$

The expression of $g(P)$ can be simplified further using following lemma:

Lemma 8. $g(P)^{\frac{q^k-1}{n}} = h(P)^{\frac{q^k-1}{n}}$ *where* $h(P) = l_{[6x+2]Q,qQ}(P) l_{[6x+2]Q+qQ,-q^2Q}(P)$.

By the above lemma, $g(P)$ can be replaced by $h(P)$ when computing the O-Ate pairing on BN-curves. Finally (Q,P) can be represented as:

$$(Q,P) \mapsto f_{6x+2,Q}(P) \cdot h(P). \tag{2.4}$$

The above expression is the expression for O-Ate pairing on BN-curves. In this work we have used the pairings equation as 2.4 for efficient implementation.

2.12 Final Exponentiation

Exponentiation is the final step in Miller algorithm [8] in which it is required f to be raised to the exponent $\frac{q^k-1}{n}$. BN-curves have embedding degree as $k = 12$. We can write $k = 2d$ and implement $\mathbb{F}_{q^k}$ as a quadratic extension of $\mathbb{F}_{q^d}$ where $d = 6$. Then, exponent can be represent as

$$\frac{q^k-1}{n} = \frac{q^{2d}-1}{n} = \frac{(q^d-1)(q^d+1)}{n}.$$

k was chosen to be minimal such that $n|q^k-1$, we see that $n \nmid q^d-1$ and n is a prime, $n \mid q^d+1$. So, we split the exponentiation in two parts as exponentiation by q^d-1 and then $\frac{q^d+1}{n}$.

If $a \in \mathbb{F}_{q^k}$ can be represented as $a = \alpha + \beta s$ where $\alpha, \beta \in \mathbb{F}_{q^d}$ and s is an adjoined square root. The Frobenius endomorphism can be represented as

$$(\alpha+\beta s)^{q^d} = \alpha - \beta s \tag{2.5}$$

Equation 2.5 gives us ability to compute f^{q^d-1} in an easy way. The next step would be computing $(f.h)^{\frac{q^d+1}{n}}$ and represent as

$$\frac{q^d+1}{n} = (q^2+1)\frac{q^4-q^2+1}{n}$$

Computing $f^{\frac{q^4-q^2+1}{n}}$ relies on computing O-Ate pairing to some suitable power will still give a bi-linear pairing. To make the computation easy, we will consider multiple of $\frac{q^4-q^2+1}{n}$ to compute the value. Grewal et al. [12] present the implementation can be performed as

$$f \to f^x \to f^{2x} \to f^{4x} \to f^{6x} \to f^{6x^2} \to f^{12x^2} \to f^{12x^3} \tag{2.6}$$

Then final exponentiation can be represented as

$$af^{6x^2} fb^p a^{p^2} (bf^{-1})^{p^3}. \tag{2.7}$$

which costs 6 multiplication and 6 Frobenius operations.

2.13 Coordinate Systems

Points on an elliptic curve can be represented using different coordinate systems. For elliptic curve based systems, the computational costs of elliptic curve addition (ECADD) and elliptic curve doubling (ECDBL) are determined by the coordinate systems. Many studies about coordinate systems for elliptic curves implementation are available in the literature to find a way to accelerate the computation of ECADD and ECDBL. The most popular coordinate system among these are Jacobian coordinates, Affine coordinates and Homogeneous or Projective coordinates. Gurleen Grewal [[12], [36]] present the formulas for point addition and point

doubling for all the three coordinates.

Projective coordinates provides easy way to represent the points on elliptic curves as it represents the ratios of the numbers instead of numbers. The projective coordinate system allow the carry denominator to the third coordinate as:

$$\left(\frac{x}{z}, \frac{y}{z}, 1\right) \longmapsto (x, y, z)$$

For example 5 can be represented as either $5 = 5/1$ or as $4 = 8/2$, which could be represented as $(5,5,1)$. The infinity point is being represented as third coordinate as 0 as dividing anything with zero indicate infinity. Affine coordinates can be transform to projective coordinates and represented as:

$$[X : Y : Z] \longmapsto [X/Z : Y/Z : 1] \longmapsto \{X/Z : Y/Z\}$$

and $\{x, y\} \mapsto [X : Y : 1]$.

The best coordinate system for efficient implementation of pairings depends on computational environment e.g., CPU architecture. As mentioned in [36] if the crossover inversion to multiplication (I/M) ratio is greater than 10.8, projective coordinates are gives better computation timing results for pairing. In this work the results of projective coordinates are better than affine coordinates. In general, actual and crossover I/M ratios gets smaller as field size increases because of the inversion formula for higher extensions. As a result, as the degree of the extension field used for the bulk of the arithmetic in a pairings computation increases, the case for using affine coordinates gets stronger.

2.14 Curve Arithmetic

For the remainder of this thesis, let m, s, a, i and r denote the times for multiplication, squaring, addition, inversion and modular reduction in $\mathbb{F}_q$, respectively. Let $\tilde{m}, \tilde{s}, \tilde{a}, \tilde{i}$ and $\tilde{r}$ denote times for multiplication, squaring, addition, inversion and modular reduction in F_{q^2} , respectively.

ARM is widely being used for embedded systems and its performance characteristic is different from PC which is the main factor that ARM platform optimization techniques are different from PC optimization techniques. For ARM processor based platforms, the ratio of cost of field inversions to field multiplications and the ratio of the cost of field multiplications to field additions is generally lower than on the PC platform. Therefore, coordinate system and choice of formulas can be optimal for one platform but not perform much speedup for another platform.

The authors of the original implementation [[12], [36]] examined the implementation of Jacobian, Affine and Homogeneous coordinates systems and it is shown that homogeneous coordinates are most efficient for the given implementation. Explicit formulas and main computation costs are being presented by the author for different coordinate systems. For details operation counts and implementations, one can refer [[36], [37]].

In the next chapter we discuss the basics of prime field arithmetic and implementations of basic algorithms such as, addition, multiplication and reduction.

Chapter 3

Prime Field Arithmetic

In this chapter, we discuss the arithmetic over prime fields. For pairing computations all of the higher level arithmetic operations are done over prime fields.

3.1 Addition and Subtraction

Addition and substation are most basic algorithms for any prime field algorithm. An assignment of the form "$(\varepsilon, z) \leftarrow w$" for an integer w is understood as:

$$z \leftarrow w \bmod 2^W, \textit{ and}$$

$$\varepsilon \leftarrow 0 \textit{ if } w \in [0, 2^W), \textit{ otherwise } \varepsilon \leftarrow 1.$$

where W is the word size of machine. If w is defined as $w = x + y + \varepsilon'$ for $x, y \in [0, 2^W)$ and $\varepsilon' \in \{0, 1\}$, then $w = \varepsilon 2^W + z$ and ε is called the carry bit for single word addition. Modular Addition for multi-word numbers is defined as $((x + y) \bmod q)$ and subtraction is defined as $((x - y) \bmod q)$.

Algorihm 3.1 explains addition in prime field. As shown in step-1, we add first word of the inputs and save the carry. *For* loop in Step-2 go over all the words, perform the addition on current words and also adds the carry of previous addition. Finally if the carry flag is one or the addition result is greater than prime (q), step-5 perform the reduction step.

Similarly subtraction algorithm works as explained in Algorithm 3.2. In step-1, first we perform subtraction of first word and save the carry which is being forwarded at each step in the *for* loop. Finally in step-5, if the carry flag is one, we add the prime (q) and return the output.

Algorithm 3.1 Addition in $\mathbb{F}_q$ field[1].

Inputs: Modulus q, and integers $A = (a_{n-1}, ..., a_1, a_0)$ and $B = (b_{n-1}, ..., b_1, b_0)$. $A, B \in [0, q-1]$
Output: $C = (A + B)\ mod\ q$.
1: $(\varepsilon, c_0) \leftarrow a_0 + b_0$
2: **for** $i = 1$ **upto** $n-1$ **do**
3: $\quad (\varepsilon, c_i) \leftarrow a_i + b_i + \varepsilon$
4: **if** $\varepsilon = 1$**or** $C \geq q$
5: $\quad C \leftarrow C - q$
6: **Return(C)**

Algorithm 3.2 Subtraction in $\mathbb{F}_q$ field.[1].

Inputs: Modulus q, and integers $A = (a_{n-1}, ..., a_1, a_0)$ and $B = (b_{n-1}, ..., b_1, b_0)$. $A, B \in [0, q-1]$
Output: $C = (A + B) \bmod q$.
1: $(\varepsilon, c_0) \leftarrow a_0 - b_0$
2: **for** $i = 1$ **upto** $n-1$ **do**
3: $\quad (\varepsilon, c_i) \leftarrow a_i - b_i - \varepsilon$
4: **if** $\varepsilon = 1$
5: $\quad C \leftarrow C + q$
6: **Return(C)**

Algorithm 3.3 Schoolbook Multiplication Method 3.3.

Inputs: Two n-digit arguments $A = (a_{n-1}, ..., a_1, a_0)$ and $B = (b_{n-1}, ..., b_1, b_0)$.
Output: $P = A.B = (p_{2n-1}, ..., p_1, p_0)$.
1: $P \leftarrow 0$
2: **for** $i = 0$ **upto** $n-1$ **do**
3: $\quad s \leftarrow 0$
4: $\quad$ **for** $j = 0$ **upto** $n-1$ **do**
5: $\quad\quad (s, c) \leftarrow a_j \times b_i + p_{i+j} + s$
6: $\quad\quad p_{i+j} \leftarrow c$
7: $\quad$ **end for**
8: $\quad p_{n+i} \leftarrow s$
9: **end for**

3.2 Finite Field Multipliers

Pairings based cryptosystems are computation-intensive algorithms especially when execution is taking place on embedded processors with low resources. The basic reason of computation rich algorithms are the operand size of underlying computations, e.g., finite field multiplication, exponentiation. Among these, multiplication is important building block of elliptical curve cryptosystems and needs careful optimization in pairing-based cryptographic algorithms. Finite field multiplication in pairings algorithms are defined as $c = a.b \bmod q$ and faster implementation results in improving the performance of pairings algorithms especially on embedded platforms. In this section we will analyze four multipliers as Schoolbook method [26], Comba [38], Montgomery [23], Karatsuba [25].

3.2.1 Schoolbook Method

The schoolbook method for finite field multiplication is also known as operand scanning method. Schoolbook multiplier can be represented as in Equation 3.1 where we have input arguments as $A = A_H \cdot 2^{n/2} + A_L$ and $B = B_H \cdot 2^{n/2} + B_L$ and the output $C = A.B$.

$$C = A_H . B_H . 2^n + (A_H . B_L + A_L . B_H) 2^{\frac{n}{2}} + A_L . B_L \tag{3.1}$$

As explained in Algorithm 3.3, it consist of two *for* loops and each of them iterate over the digits of input arguments of $n - bit$. In each iteration of outer *for* loop at step-2, the digit b_i of second operand is multiplied with all the digits of first argument and n-bit results is accumulated according to their weight. After completion of two loops the weighted output is

Algorithm 3.4 Comba Multiplication Method [38].

Inputs: Two n-digit arguments $A = (a_{n-1}, ..., a_1, a_0)$ and $B = (b_{n-1}, ..., b_1, b_0)$.
Output: $P = A.B = (p_{2n-1}, ..., p_1, p_0)$.
1: $(s, c, t) \leftarrow 0$
2: **for** $i = 0$ **upto** $n-1$ **do**
3: **for** $j = 0$ **upto** i **do**
4: $(s, c, t) \leftarrow (s, c, t) + a_j \times b_{i-j}$
5: **end for**
6: $p_i \leftarrow t$
7: $t \leftarrow c, c \leftarrow s, s \leftarrow 0$
8: **end for**
9: **for** $i = n$ **upto** $2n-2$ **do**
10: **for** $j = i-n+1$ **upto** $n-1$ **do**
11: $(s, c, t) \leftarrow (s, c, t) + a_j \times b_{i-j}$
12: **end for**
13: $p_i \leftarrow t$
14: $t \leftarrow c, c \leftarrow s, s \leftarrow 0$
15: **end for**
16: $p_{2n-1} \leftarrow t$

non-reduced result of multiplication.

3.2.2 Comba's Method

Comba [38] first describe one alternative method for finite field multiplication which is called Comba's method. Comba's method is also known as product scanning method. Algorithm 3.4 illustrates comba algorithm to multiply two n-bit numbers and typically faster than schoolbook method. In step-2, the outer *for* loop is iterating through each element of output p_i and inner loop implements the algorithm as much simpler as schoolbook multiplier. The $i-th$ digit of output p_i of $P = A.B$ is the accumulation of all inner loop products $a_j \times b_{i-j}$ where $0 \leq j \leq i$ as shown in step-4 and step-11. The carry is added eventually to the digit from the calculation of previous digit inside the loop. Algorithm 3.4 uses three $w-bit$ registers (s, c, t) for the storage of sum of $2w-bit$ and operations at step-7 and step-14 are the right shift of the registers (s, c, t).

For this work, we have implemented 638-bit comba multiplication using 32-bit ARM assembly to compare the performance improvement using assembly.

Algorithm 3.5 SOS Montgomery Multiplier [39].

Inputs: Two n-digit arguments $A = (a_{n-1}, ..., a_1, a_0)$ and $B = (b_{n-1}, ..., b_1, b_0)$.
Output: $P = A.B = (p_{2n-1}, ..., p_1, p_0)$.
1: **for** $i = 0$ **upto** $n-1$ **do**
2: $C \leftarrow 0$
3: **for** $j = 0$ **upto** $n-1$ **do**
4: $(C, S) \leftarrow P_{i+j} + a_j \times b_i + C$
5: $P_{i+j} \leftarrow S$
6: **end for**
7: $P_{i+n} \leftarrow C$
8: **end for**
9: **Return P**

3.2.3 Montgomery Method

Montgomery multiplication algorithm is typically faster than both schoolbook and comba multiplication methods. As described in [23], there are five different versions of Montgomery multiplier as:

- Separated Operand Scanning (SOS)
- Coarsely Integrated Operand Scanning (CIOS)
- Finely Integrated Operand Scanning (FIOS)
- Finely Integrated Product Scanning (FIPS)
- Coarsely Integrated Hybrid Scanning (CIHS)

Separated Operand Scanning (SOS) is most usable for pairings when we use lazy reduction technique. Coarsely Integrated Operand Scanning (CIOS) method is improvement over SOS which gives modular multiplication result of input arguments. We use SOS multiplier for NEON implementation. SOS multiplier algorithm is defined in Algorithm 3.5 which shows two *for* loops, in step-1 and step-3, iterate through the input arguments and combine the multiply of each word with previous carry in step-4. Carry is being handled each time after inner loop is being finished as shown in step-7.

The reduction method is performed using $u \leftarrow (t + m \cdot n)/r$, where $m = t \cdot n' \bmod r$ and $\gcd(n, r) = 1$. Since reduction process is word by word, we can use $n'_0 = n' \bmod 2^w$ instead of n'. Coarsely Integrated Operand Scanning(CIOS) method integrate the multiplication and

Algorithm 3.6 CIOS Montgomery Multiplier [39].

Inputs: Two n-digit arguments $A = (a_{s-1}, ..., a_1, a_0)$ and $B = (b_{s-1}, ..., b_1, b_0)$.
Output: $P = A.B = (p_{2s-1}, ..., p_1, p_0)$.
1: **for** $i = 0$ **upto** $s-1$ **do**
2: $C \leftarrow 0$
3: **for** $j = 0$ **upto** $s-1$ **do**
4: $(C,S) \leftarrow P_j + a_j \times b_i + C$
5: $P_{i+j} \leftarrow S$
6: **end for**
7: $(C,S) \leftarrow P_s + C$
8: $P_s \leftarrow S$
9: $P_{s+1} \leftarrow C$
10: $C \leftarrow 0$
11: $m \leftarrow P_0 \times n'_0$ **mod** W
12: **for** $j = 0$ **upto** $s-1$ **do**
13: $(C,S) \leftarrow P_j + m \times n_j + C$
14: $P_j \leftarrow S$
15: $(C,S) \leftarrow P_s + C$
16: $P_s \leftarrow S$
17: $P_{s+1} \leftarrow P_{s+1} + C$
18: **for** $j = 0$ **upto** s **do**
19: $P_j \leftarrow P_{j+1}$
20: **end for**
21: **end for**
21: **end for**
22: **Return P**

reduction steps in same algorithm which is improvement on SOS method which use reduction method after the multiplication. Instead of computing the entire product $a \cdot b$, then reducing, CIOS multiplication perform the product and reduction word by word. As shown in Algorithm 3.6, step-1 and step-3 shows two loops over the input arguments and perform word vise multiplication and addition of carry at step-4. Once the inner loop performs the multiplication another for loop at step-12 performs the reduction step using m calculated at step-11 as the value of m in the $i-th$ iteration depends only on the value of P_i. Once the three loop ends it returns the reduced output at step-22.

Montgomery multiplier is faster than other multiplier for NEON implementations of 254-bit. In this work, we have implemented pairings using Montgomery multiplier and pairings results are being compared with pairing implementations using other multipliers.

Algorithm 3.7 Montgomery Product $MonPro(\bar{a},\bar{b})$ [23].

Inputs: prime n, n', $r = 2^k$ and $\bar{a},\bar{b} \in \mathbb{F}_q$
Output: $\bar{c} = MonPro(\bar{a}\cdot\bar{b})$.
1: $t \leftarrow \bar{a}\cdot\bar{b}$
2: $u \leftarrow (t + (t \cdot n' \bmod r)\cdot n)/r$
3: **if** $u > n$ **then**
4: **return** $u - n$
5: **else**
6: **return** u
7: **end if**

3.2.4 Karatsuba Method

Karatsuba Multiplication algorithm reduces the size of $m-bit$ multiplication to three multiplication of $\frac{m}{2}$–bit but it increases the cost of additions. These half size multiplication can be done using schoolbook or comba multiplier. For Karatsuba multiplier each input is of size m-bit and w is the word size of machine. Each operand for multiplier is written as $A = (A[n-1],...,A[2],A[1],A[0])$ and $B = (B[n-1],...,B[2],B[1],B[0])$ and output is $C = (C[2n-1],...,C[2],C[1],C[0])$, where $n = [m/w]$. We can represent the input arguments as $A = A_H \cdot 2^{n/2} + A_L$ and $B = B_H \cdot 2^{n/2} + B_L$ and the output $C = A.B$ can be represented as Equation 3.2. Karatsuba multiplier can be represent in recursive way and its complexity is $\theta(n^{log_2 3})$. There are two typical ways to describe the karatsuba multiplication algorithm as additive Karatsuba and substrative Karatsuba. Additive Karatsuba multiplier is defined as:

$$C = A.B = A_H.B_H.2^n + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{\frac{n}{2}} + A_L.B_L \tag{3.2}$$

and the subtractive Karatsuba multiplier is defined is

$$C = A.B = A_H.B_H.2^n + [A_H.B_H + A_L.B_L - |A_H - A_L|.|B_H\text{-}B_L|] \cdot 2^{\frac{n}{2}} + A_L.B_L \tag{3.3}$$

3.3 Reduction

The performance of elliptic curve schemes depends heavily on finite field multiplication, so the reduction techniques are performance factor for computations. The most basic reduction algorithm is division algorithm but its not efficient if the size of integer is large as division is

expensive computation. There are faster methods available reduction which includes Barrett Reduction [40], Montgomery reduction [41] and Special Moduli such as Mersenne primes. For pairings there are no special moduli primes and hence we use Montgomery reduction as it is a lot efficient than Barrett when implemented in assembly.

Montgomery reduction technique has replaced the classic heavy reduction technique with less-expensive operations. Montgomery reduction algorithm perform the transformation of the data and calculates Montgomery product. Let the modulus n be a m-bit integer, i.e., $2^{k-1} \leq n < 2^k$ and let $r = 2^k$ such that $\gcd(r,n) = \gcd(2^k,n) = 1$ and this requirement is satisfied when n is odd. Montgomery multiplier uses the n-residue of an integer $a < n$ as $\bar{a} = a \cdot r \pmod{n}$. The Montgomery reduction introduce a much faster multiplication routine which computes the multiplication of the two integers whose n-residues are known. Lets suppose $\bar{a}$ and $\bar{b}$ are two n-residues, then Montgomery product is defined as the n-residue

$$\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \tag{3.4}$$

where r^{-1} is the inverse of r modulo n with the property $r^{-1} \cdot r = 1 \pmod{n}$. The result c in 3.4 is the n-residue of the product of a and b such that $c = a \cdot b \pmod{n}$, since

$$\begin{aligned} \bar{c} &= \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \pmod{n} \\ &= c \cdot r \pmod{n} \end{aligned}$$

Montgomery reduction uses another quantity, n', which has the property as

$$r \cdot r^{-1} - n \cdot n' = 1$$

The integers r^{-1} and n' can both be computed by Extended Euclidean algorithm [42]. The computation of $MonPro(\bar{a},\bar{b})$ is explained in algorithm 3.7.

Montgomery reduction step is calculating u as shown in step-2 of Algorithm 3.7. Step3 to step-7 compare u with n and return the reduced result accordingly. This method is very efficient when there are many multiplications are performed for given input, such as modular exponentiation.

Next chapter explains the implementation methods using NEON engine on different ARM platforms.

Chapter 4

NEON based ARM Architectures

ARM introduced single instruction multiple data (SIMD) extension for its processor after ARMv6 series. ARM introduced NEON as supportive coprocessor that is included in ARM Cotex-A8, Cortex-A9 and Cortex-A53 etc. The co-processor contains an Arithmetic Logical Unit (ALU), shift unit and floating point addition and multiply unit which woks for SIMD instructions. This SIMD processing unit is called $NEON$ or $NEON-Engine$. NEON engine is an architecture extension for the processors which supports groups of instructions that processes vectors stored in 64-bit or 128-bit vector registers for both signed and unsigned values. NEON support special SIMD instructions on vector of elements of the same data type and each instruction perform the same operation in all the vector lanes.

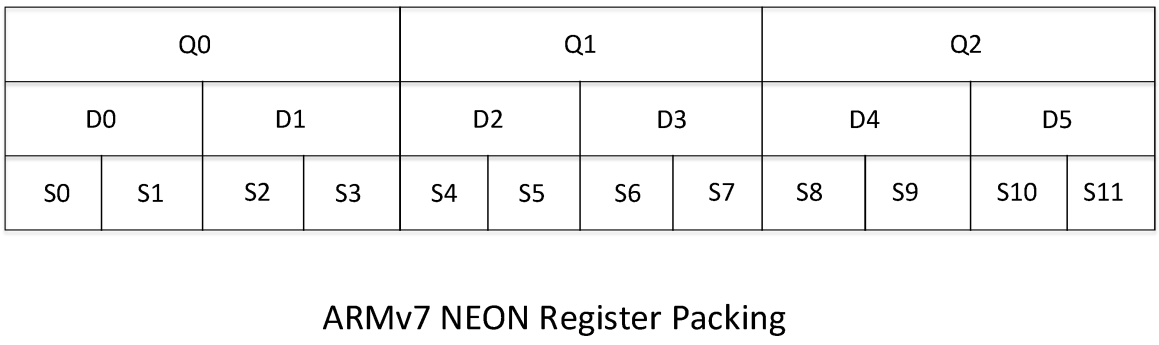

Figure 4.1: Graphic interpretation of ARMv7 NEON Register Packing.

Table 4.1: ARMv7 NEON instruction options.

Parameters	meaning	Definition
<mod>	modifiers	Q: This uses saturating arithmetic, such as $VQABS$.
		H: It shows the shifting right by one place, such as $VHADD$.
		D: This instruction doubles the result, such as $VQDMULL$.
		R: This perform the rounding on the result such as $VRHADD$.
<op>	Operation	Such as ADD, MUL, SUB.
<shape>	Shape	Long(L) : Iinputs are double-word vector operands, result is a quad-word vector.
		Wide(W) : Inputs are double-word vector and a quad-word vector, result is a quad-word vector.
		Narrow(N): Inputs are quad-word vector operands, result is a double-word vector.
<cond>	Condition	used with IT instruction
<.dt>	Data type	Such as $s8$, $u8$, $f32$ etc.
<src1>	Source Operand 1	It can be one of the available vector register.
<src2>	Source Operand 2	It can be one of the available vector register.

4.1 ARMv7 Architecture

ARMv7 architecture has 13, 32-bit general purpose register which are represented as R0-R12. NEON engine in ARMv7 uses special vector registers which is separate register file than general registers and represented as 64-bit D or double-word and 128-bit Q or quad-word. Figure 4.1 shows the register packing for ARMv7. For ARMv7 or lower version of ARM processors there are 16 128-bit registers as Q0-Q15 or 32 64-bit registers as D0-D31. Q0 register is corresponding to D0-D1 and Q1 register is corresponding to D2-D3 etc. NEON uses instructions to load/store and process the data in these registers. Neon instructions has capabilities to perform memory access, data-copy to and from NEON to general purpose registers. NEON can also perform data type conversion and data procession in D or Q registers.

There are many general instructions are available to process the vector in NEON such as addition, multiplication, shift, compare and selection, shuffles (ZIP, UZIP) etc. General instruction set of ARMv7 starts with letter “V” and suffix of instruction indicates the size of data vector to use. The general format of instructions is as:

$$V\{<mod>\}<op>\{<shape>\}\{<cond>\}\{.<dt>\}\{<dest>\},src1,src2$$

where $\{<>\}$ represents an optional parameter and other parameters are:

Table 4.2: Basic difference in ARMv7 and ARMv8 assembly instructions.

Instructions Type	Architecture	
	A32	A64
Arithmetic instructions	*ADD Rd, Rn, #9*	*ADD Wd, Wn, #9*
	ADDS Rd, Rm, LSL#2	*ADDS Wd, Wn, LSL #2*
	MUL Rd, Rn, Rm	*MUL Wd, Wn, Wm*
	MOV Rd, #imm	*MOV Wd, #imm*
Load/Store	*PUSH r0 – r1*	*STP x0, x1, [sp, #16]*
	POP r0 – r1	*LDP x0, x1, [sp],#16*
	LDMIA r0!, r1, r2	*LDP x0, x1, [x0], #16*
	STMIA r0!, r1, r2	*LDP x0, x1, [x0], #16*
Subroutine return	*MOV PC, LR*	*RET*
	POP PC	
	BX LR	
Exception return	*SUBS PC, LR, #4* *MOVS PC, LR*	*ERET*

For example:

VADD.I8 $D0$, $D1$, $D2$ instruction adds two 64-bit vector registers and store the results in a 64-bit register.

VMULL.S16 $Q1$, $D4$, $D5$ instruction multiply two 64-bit registers and store the results in a 128-bit vector register.

4.2 ARMv8 Architecture

ARMv8 works as 64 bit architecture which introduced new instruction set (aarch64). ARMv8 architecture has 31 general purpose registers ($X0 - X30$) which are 64-bit accessible at all the times and lower half of each register can be represented as $W0 - W30$ can be representedas as shown in Figure 4.2. The basic differnece in assembly instructions for ARMv7 and ARMv8 are shown in Table 4.2. ARMv8 architecture also having separate NEON register file which is differnt from ARMv7 register file. NEON extension register in ARMv8 are distinct from ARM core register set and contains 32×128 bit wide vector registers or 32×64 bit wide vectors which hold lower 64–bit of each $128 - bit$ register. ARMv8 architecture supports instruction set as for AArch32 using NEON intrinsic, which is equivalent to the ARMv7 and for AArch64 which supports same NEON registers. Figure 4.3 shows the vector register packing for ARMv8. Vn represents the 128-bit NEON vector register, Dn can be used to access the lower 64-bits of Vn and Sn represents the lower 32-bits of Dn.

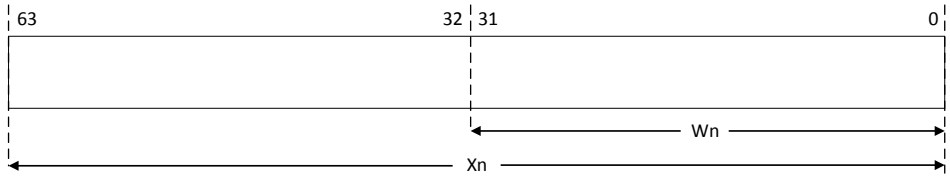

Figure 4.2: Graphic interpretation of ARMv8 scalar Register Packing.

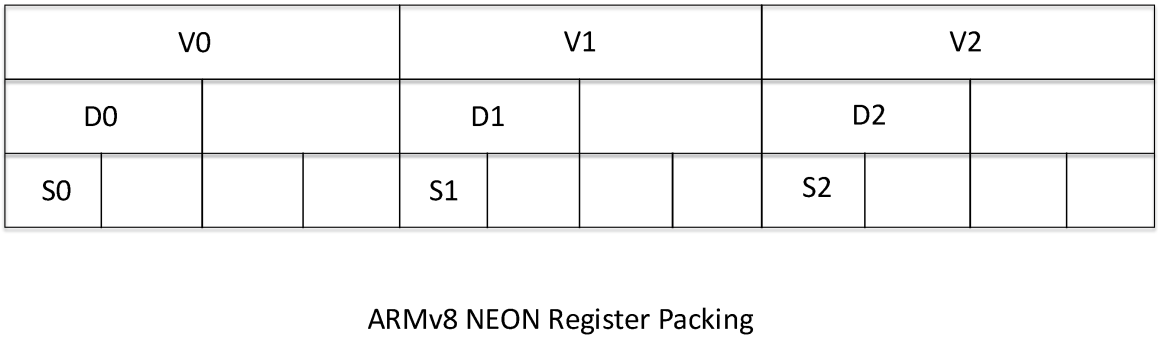

Figure 4.3: Graphic interpretation of ARMv8 Register Packing.

Table 4.3: ARMv8 vector shape and Name convention.

Shape(bits×lanes)	$8b \times 8$	$8b \times 16$	$16b \times 4$	$16b \times 8$	$32b \times 2$	$32b \times 4$	$64b \times 1$	$64b \times 2$
Name	$V_n.8B$	$V_n.16B$	$V_n.16H$	$V_n.8H$	$V_n.2S$	$V_n.4S$	$V_n.1D$	$V_n.2D$

Table 4.4: ARMv8 NEON instruction options.

Parameters	meaning	Definition
<prefix>	prefix	S/U/F/P represents signed/unsigned/bool data type
<op>	Operation	Such as *ADD*, *SUB*.
<suffix>	Suffix	P is "pairwise" operations, such as *ADDP*.
		V is the new reduction operations, such as *FMAXV*.
		2 is new widening/narrowing. Such as *ADDHN*2, *SADDL*2.
<T>	Data Type	Such as 8B/16B/4H/8H/2S/4S/2D.
		B represents byte (8-bit).
		H represents half-word (16-bit).
		S represents word (32-bit).
		D represents a double-word (64-bit).

Table 4.3 shows the vector registers used in ARMv8. These vectors can be 128-bits wide with two or more elements or 64-bits wide with one or more elements.

The NEON instruction sets of ARMv8 AArch64 architecture is different from ARMv7 architecture and represented as:

$$\{<prefix>\}<op>\{<suffix>\}Vd.<T>,Vn.<T>,Vm.<T>$$

where $\{<>\}$ represents and optional parameter and other parameters are shown in Table 4.4.

For example:

$UADDLP\ V0.8H$, $V0.16B$ instruction adds Unsigned long pairwise.

$FADD\ V0.4S$, $V1.4S$, $V2.4S$ instruction adds two vectors and store in third.

4.3 ARMv7/v8 NEON programming basics

GNU GCC provides vectorizing options for C code to generate NEON code. GCC compiler must trade-off to generate the executable code and it uses number of resources, such as registers, stack and heap space, compilation time, number of instructions, debugging strategies and number of cycles per instruction in optimal way. But Compiler generated code can compromise on resources as per the system state while running the code. To improve the performance of NEON based systems we can write the code using NEON intrinsic or NEON assembly

```
#include <arm_neon.h>
void add_float_c(uint32_t * dst, uint32_t * src1, uint32_t * src2, uint32_t count)
{
        uint32_t i;
        for (i = 0; i < count; i++)
                dst[i] = src1[i] + src2[i];
}
```

Figure 4.4: Addition of unsigned integer (uint32_t) array using C. Assumed that number of words in input are multiple of 4.

methods which can use the NEON engine directly. The basic steps for NEON computations are load, compute and store.

4.3.1 NEON intrinsic method

NEON intrinsic language provides methods for C like functions as interface to NEON operations which can be used to call NEON instructions as C-like code. The compiler can generate relevant NEON instructions based object file and then executable to run on either an ARMv7-A or ARMv8-A platform. The C code shown in Figure 4.4 presents the addition for an array of elements count as 4. The *for* loop adds the individual elements of the input arrays and store the result in corresponding element of output array.

NEON intrinsic code of the addition of array is shown in Figure 4.5. Most of the calls in intrinsic NEON code converted to neon assembly instructions using compiler. All the intrinsic methods, such as vld1_u32, are defined in "neon.h". In the given example, inside the *for* loop first individual elements are being load in varialbes and then we add them and finally store in the output register using vst1_u32() method. We have to set the compiler flags as "-mfloat-abi=hard -mfpu=neon" to use floating-point unit (FPU) implementation in the NEON SIMD architecture extension. We should also use compiler option " -ffast-math -O3" as strict conformance to the floating-point standard (IEEE 754) should be avoided since NEON may not implement it entirely. Compiler can check the flag as *__aarch64__* for aarch64 architecture specific instructions.

4.3.2 NEON Assembly

NEON assembly can be written in inline assembly using "asm volatile ();" or in separate file as file.s. NEON Assembly instructions are different for ARMv7 and ARMv8 as mentioned in section above. Following codes show the NEON assembly code for addition example of the arrays for ARMv7 and ARMv8 in Figure 4.6 and 4.7 respectively. As shown in Figure 4.6,

```
#include <arm_neon.h>
void add_float_neon1(uint32_t * dst, uint32_t * src1, uint32_t * src2, uint32_t count)
{
	int i;
	for (i = 0; i < count; i += 4)
	{
		uint32x4_t in1, in2, out;
		in1 = vld1_u32(src1);
		src1 += 4;
		in2 = vld1_u32(src2);
		src2 += 4;
		out = vaddq_u32(in1, in2);
		vst1_u32(dst, out);
		dst += 4;
		// The following way shows how to use AArch64 specific NEON instructions.
#if defined (__aarch64__)
		uint32_t tmp = vaddvq_u32(in1);
#endif
	}
}
```

Figure 4.5: Addition of unsigned integer (uint32_t) array using Intrinsic NEON. Assumed that count is multiple of 4.

```
.text
.syntax unified
.align 4
.global add_float_neon2
.type add_float_neon2, %function
.thumb
.thumb_func
add_float_neon2:
.L_loop:
        vld1.32 {q0}, [r1]!
        vld1.32 {q1}, [r2]!
        vadd.u32 q0, q0, q1
        subs r3, r3, #4
        vst1.32 {q0}, [r0]!
        bgt .L_loop
        bx lr
```

Figure 4.6: Assembly code in .S file for ARMv7.

NEON assembly code uses less instructions to perform the same computations. vld1.32 loads 128-bits from input argument and store in a vector register then vadd.u32 instruction adds two 128-bit vectors and store in third register. Finally the result is being stored in output register using vst1.32 instruction.

Figure 4.7 shows the NEON assembly code for ARMv8 architecture. ld1 loads 128-bits from input argument and store in a vector register then uqadd instruction adds two 128-bit vectors and store in same register. Finally the result is being stored in output register using st1 instruction.

```
.text
.align 4
.global add_float_neon2
.type add_float_neon2, %function
add_float_neon2:
.L_loop:
        ld1{v0.4s}, [x1], #16
        ld1 {v1.4s}, [x2], #16
        uqadd v0.4s, v0.4s, v1.4s
        subs x3, x3, #4
        st1 {v0.4s}, [x0], #16
        bgt .L_loop
        ret
```

Figure 4.7: Assembly code in .S file for ARMv8.

Chapter 5

Field Level Optimizations[1]

There is only one NEON based implementation available for pairings in literature. Ana Helena Sánchez et al. [19] tried to use NEON engine to optimize timing for pairing computations on ARM processor based architecture. Ana Helena Sánchez et al. implementation is fastest for 256-bit but there is no implementation for 446-bit and 638-bit security level using NEON engine.

Finite Field multiplier is the most important computation for pairing algorithms. There are many implementations of different scalar multipliers using NEON in literature which can be used to make pairing computation faster on low resource based architectures. Seo and Liu [22] presented modified version of Montgomery multiplier using NEON which is faster than original implementation of Montgomery multiplier. Revisited Montgomery multiplier implementation is effective for 256-bit (8 words), 512-bit (16 words), 1024-bit (32 words) and so on as it uses 4 words in processing simultaniously. This implementaton is not effective for 446-bit (14 words) and 638-bit (20 words) fields and for this reason we use NEON implementation of Schoolbook method.

Seo and Liu [43] presented another multiplier additive Karatsuba Multiplier using ARM-NEON which is an improved version of original Karatsuba multiplier. Additive Karatsuba multiplier is presented as efficient for higher field size multiplications. In the given implementation the Karatsuba method is being used as two level multiplier and montgomery multiplier is being used as the $\frac{n}{2}$–bit multiplications. 446-bit and 638-bit security levels uses odd numbers of byte multiplication so montgomery multiplier for $\frac{n}{2}$–bit multiplication would not be efficient. Comba multiplier is very flexible for variable size multiplication which makes it better choice to implement additive Kratsuba multiplier. Our implementation uses Comba multiplier for $\frac{n}{2}$–bit multiplication and compare the implementation result with other multipliers present

[1] NEON implementations of differnt algorithms can be found here.

in literature.

In this chapter, we present previous efforts made in optimization of the pairing based protocols and our optimization in basic algorithms such as, multiplication, inversion, square.

5.1 Challenges in implementing pairing for higher fields

The concept of Lazy reduction [16] is conveniently implementaed with irreducible binomials in [37]. Aranha et al. [37] proposed efiicient basic computation algorithms e.g., inversion, squaring etc., using lazy reduction and tower as $\mathbb{F}_q \rightarrow \mathbb{F}_{q^2} \rightarrow \mathbb{F}_{q^6} \rightarrow \mathbb{F}_{q^{12}}$. According to the Miller loop implementation explained in [12], for point arithmetic and line evaluation, reductions can be delayed from underlying $\mathbb{F}_{q^2}$field during squaring and multiplication. However, the upper layer reductions should only be delayed in cases where the technique leads for fewer reductions. In our implementations of lazy reduction most of the reductions are being delayed till $\mathbb{F}_{q^2}$ level. $\mathbb{F}_{q^2}$ reductions are two $\mathbb{F}_q$ field reductions and it is efficient to implement $\mathbb{F}_{q^2}$ reduction using NEON. We have implemented parallel reduction technique using NEON engine in $\mathbb{F}_{q^2}$ field which is being used as basic unit for $\mathbb{F}_{q^6}$ and $\mathbb{F}_{q^{12}}$reductions and we have used assembly implementation of Montgomery reduction from relic toolkit [44].

Single precision operations are defined as operands occupying $n = \lceil \lceil \log_2 q \rceil / w \rceil$ words, where w is the word size of machine and double precision operations are defined as with operands of $2n$ words. Pairing computations uses both single precision and double precision operations accordingly in tower computations. Lazy reduction technique is effective for upper layer algorithm performance, but there are some penalties for the use of lazy reduction technique. With lazy reduction technique, reduction is being delayed till lower layers which replace single precision operations in higher field with double precision operations. For higher security levels, when prime field size is larger than the word size and there are less number of registers to hold the data (e.g., ARM has 12 general purpose registers) longer computations loading data from memory could slowdown the overall performance. This is the reason that single and double precision algorithms should be optimized for systems with low resources. However, this disadvantage can be minimized up-to some extend using NEON engine as extra SIMD registers can be used for parallel load/store and computations which leads to improve performance of basic algorithms.

We have implemented Schoolbook and Karatsuba multiplier using NEON and compare their performance for ARM based architecture also improved higher layer algorithms. The next subsection describe them one by one.

Algorithm 5.1 Parallel multiplication (mul_N) using NEON [19]

Inputs: $a = (a_0, a_1)$,$b = (b_0, b_1)$,$c = (c_0, c_1)$ and $d = (d_0, d_1)$
$a, b, c, d \in \mathbb{F}_q$ **Output:** $M = a.b$ and $N = c.d$

1: $M \leftarrow 0$,$N \leftarrow 0$
2: **for** $i = 0 \rightarrow 1$ **do**
3: $T_1 \leftarrow 0$, $T_2 \leftarrow 0$
4: **for** $j = 0 \rightarrow 1$**do**
5: $(T_1, C_1) \leftarrow M_{i+i} + a_j.b_i + T_1$, $(T_2, C_2) \leftarrow N_{i+i} + c_j.d_i + T_2$
6: $M_{i+j} = C_1$, $N_{i+j} = C_2$
7: **end for**
8: $M_{i+n} = T_1$, $N_{i+n} = T_2$
9: **return** (M, N)
10: **end for**

5.2 Proposed NEON implementation of Schoolbook multiplier.

For this work, we have implemented Schoolbook multiplier using NEON for BN-446 and BN-638 curves. The algorithm uses NEON implementation of Montgomery multiplier as described in [19] and described in Algorithm 5.1 for half size multiplication. Algorithm 5.1, represnts the base multiplier. The input arguments are four numbers a, b, c, d and outputs are $M = a.b$ and $N = c.d$ as the product of two input arguments. We are referencing algorithm 5.1 as mul_N in rest of the paper and the steps are as follows:

- The step-1 of Algorithm 5.1 initialize the output variable as zero.
- Two loops as shown in step 2 and step-4 of Algorithm 5.1 iterate over the two input arguments. T_1 and $T_2$are variables to hold temporary values.
- Step-5 performs the parallel multiplication and assign the values for $C_1$and C_2as carries for each step to further values of M and N. After for loop of step-4 ends, T and $T_2$are assigned to second half of the output values M and N.
- Finally, at the end of both for loops the output will be products of 4 numbers, stored in two numbers.

The implementation of Schoolbook multiplier algorithm is shown in Figure 5.1. It is being implemented in three steps. Step-1 and step-2 of algorithm uses parallel Montgomery multiplication as explained in Algorithm 5.1 which multiplication of half of the original bits in

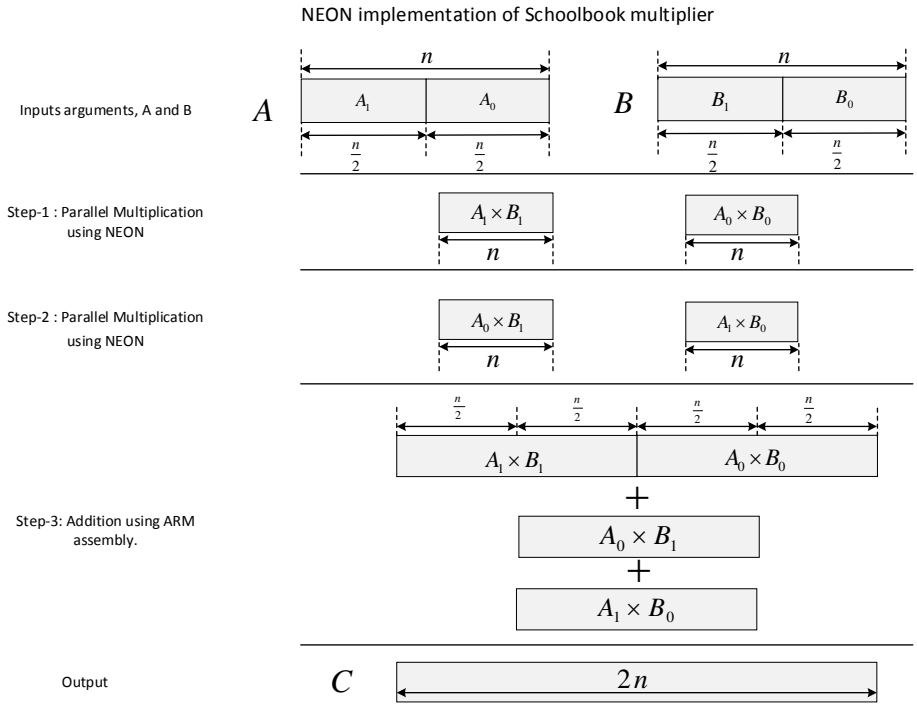

Figure 5.1: Graphic interpretation of Schoolbook Multiply Algorithm using NEON.

numbers using NEON. Step-3 is three $n-bit$ number addition using ARM assembly. NEON is not effective for additions because of carry handling so the step-3 is being implemented in ARM assembly. Finally the output C is $2n$-bit long number.

We also have used Coarsely Integrated Operand Scanning (CIOS) for parallel F_{q2} field multiplication as explained in [19] and using its reference as $mulRed_N$ in rest of the paper.

5.3 Karatsuba multiplier using NEON

Karatsuba multiplier is defined in Equation 3.2. In this work, we have implemented the Additive version of Karatsuba multiplier. As described in Algorithm 5.3, the inputs for Additive Karatsuba algorithm are n–bit numbers $a = (a_0, a_1), a \in \mathbb{F}_q$ and $b = (b_0, b_1), b \in \mathbb{F}_q$ and output is $2n$–bit number $C = a.b$. The steps are defined as:

- Step-1 calculates the parallel multiplication of (a_0, b_0) and (a_1, b_1) and store the result in M and N respectively.

Algorithm 5.2 Schoolbook Multiplier using NEON

Inputs: $a = (a_0, a_1), a \in \mathbb{F}_q$
$b = (b_0, b_1), b \in \mathbb{F}_q$
Output: $C = a.b$
1: $(M_0, N_0) \leftarrow mul_N(a_0, b_0, a_1, b_1)$
2: $(M_1, N_1) \leftarrow mul_N(a_0, b_1, a_1, b_0)$
3: $C = N_0.2^n + M_0$
4: $C = C + M_1.2^{\frac{n}{2}}$
5: $C = C + N_1.2^{\frac{n}{2}}$
6: **return** C

Algorithm 5.3 Karatsuba Multiplier using NEON

Inputs: $a = (a_0, a_1), a \in \mathbb{F}_q$
$b = (b_0, b_1), b \in \mathbb{F}_q$
Output: $C = a.b$
1: $(M, N) \leftarrow mul_N(a_0, b_0, a_1, b_1)$
2: $(A_C, A_S) = a_0 + a_1$
3: $(B_C, B_S) = b_0 + b_1$
4: $S = A_S.B_S$
5: $S = S + (AND(COM(A_C), B_S)).2^{\frac{n}{2}}$
6: $S = S + (AND(COM(B_C), A_S)).2^{\frac{n}{2}}$
7: $S = S + (AND(A_C, B_C)).2^{\frac{n}{2}}$
8: $C = N.2^n + (S - M - N)2^{\frac{n}{2}} + M$
9: **return** C

- Step-2 and step-3 calculate the sum (A) of the $\frac{n}{2}$–bit of each numbers and store the carry and sum in separate variables.

- Step-4 calculates the product (S) of result of each step-2 and step-3.

- Step-5 and step-6 performs the addition of result of step-4 with carry handler. COM function is the 2's compliment of the input. $AND(COM(A_C), B_S)$, $AND(COM(B_C), A_S)$ and $(AND(A_C, B_C))$ are being added to adjust the carry from previous steps.

- Finally step-6 calculates the quantity $S - M - N$ and adds up all the results of previous step to calculate C and return.

Algorithm 5.4 Squaring over F_{q^2} [19]

Inputs: $M = m_0 + m_1 i$; $m_0, m_1 \in \mathbb{F}_q$;
Output: $N = M^2 \in F_{q^2}$
1: $N_0 \leftarrow m_0 - m_1$
2: $t \leftarrow m_0 + m_1$
3: $(n_1, n_0) \leftarrow mul_N(m_0, m_1, N_0, t)$
4: $n_1 \leftarrow 2n_1$
5: $n_0 \leftarrow n_0 - n_1$
6: **return** $N = n_0 + n_1 i$

5.4 Reduction using NEON

Modular multiplication is the one of the most important base field arithmetic in pairing computations. Montgomery multiplication method is effective for reduction step. Algorithm 3.7 explains the classic reduction method and we have modified it using NEON engine as explained in [19] for different field size. The first integer product of the algorithm $t = a \cdot b$, is being computed using Separated Operand Scanning (SOS) method as explained in Algorithm 3.5 which is followed by calculation of u such that $u = (t + m \cdot q)/r$, where $m = t \cdot q' \pmod r$. We are using $r = 2^{wn}$ where w is the word size of the architecture and $n = [(\lfloor \log_2 q \rfloor + 1)/w]$. Since both the steps are being performed word-by-word, we can perform $m = t \cdot q' \bmod r$ by replacing q' by $q'_0 = q' \bmod 2^w$. For F_q field reduction we are using assembly implementation of Montgomery reduction used in [44].

Appendix-2 shows the pseudo code of reduction of 256-bit numbers to reduce it to 128-bit number using NEON. We can use the algorithm to calculate perform in $\mathbb{F}_{q^2}$ field or two parallel $\mathbb{F}_q$ as the input for the algorithm are two separate 256-bit words. We use, rdc_N as the reference of reduction algorithm using NEON.

5.5 $\mathbb{F}_{q^2}$ and $\mathbb{F}_{q^6}$ field multiplication without reduction

Aranha et al. [37] proposed the algorithm for higher field multiplication for $\mathbb{F}_{q^2}$ and $\mathbb{F}_{q^6}$ without reduction. Reduced multiplication does the reduction after multiplication as per used by upper layer algorithms. Algorithm 5.6 explains the use of NEON based parallel multiplication in the algorithm which result in faster single multiplication at $\mathbb{F}_{q^2}$ level. The steps for $\mathbb{F}_{q^2}$ multiplication algorithm is explained as:

- Step-1 is using the duel multiplication of F_q field using the NEON based function mul_N

and results are being saved in temporary variables.

- Step-2 combines elements for inputs and save in temporary variables (t_0, t_1).
- Step-3, does multiplication of temporary variables of step-2.
- Step-4 and step-5, gives the T_3 which is one factor of the output.
- Step-6 calculates the resulting second factor of output. Step-7 returns the resulting multiplication.

Three $\mathbb{F}_q$ multiplications used in the F_{q^2} multiplication algorithm [37] is being replaced with two parallel $\mathbb{F}_{q^2}$multiplication and one F_q multiplication. The new implementation is resulting the improvement in final pairings performance on ARM platform.

We used simultaneous multiplication of two F_{q^2} number multiplication as shown in Algorithm 5.5 which replaces the serial multiplication of two $\mathbb{F}_{q^2}$ numbers. The input to the algorithm are four numbers $a, b, c, d \in F_{q^2}$ and outputs are $M = a.b \in F_{q^2}$ and $N = c.d \in F_{q^2}$. The steps for algorithm are as follows:

- Step-1 to step-4 adds up the components for all the input arguments and assign to temporary variables.
- Step-5 uses mul_N to multiply results of step-1 with step-2 and step-3 with step-4 and assign to new variables $(x_0 \, and \, y_0)$.
- Step-6 and step-7 multiplies first and second elements of the arguments and assign to temp variables (x_1, y_1) and (x_2, y_2).
- Step-8 to step-13, calculate the value of $m_0 \leftarrow x_1 - y_1$, $m_1 \leftarrow x_0 - x_1 - y_1$, $n_0 \leftarrow x_2 - y_2$ and $n_1 \leftarrow y_0 - y_1 - y_2$. Finally, function return output value as $M \leftarrow m_0 + m_1 i$ and $N \leftarrow n_0 + n_1 i$.

We further improved the multiplication in $\mathbb{F}_{q^6}$ field without reduction using NEON implementation of lower level multipliers. Algorithm 5.7 explain the improved implementation of $\mathbb{F}_{q^6}$ multiplier for BN254 curve. We can observe that six $\mathbb{F}_{q^2}$ multiplications used in the algorithm [37] is being replaced with three parallel $\mathbb{F}_{q^2}$ multiplication. The new implementation is resulting the improvement in final pairing performance on ARM platform.

Algorithm 5.5 Simultaneous multiplication of two numbers ($muldual_N$) in $\mathbb{F}_{q^2}$.

Inputs: $a = a_0 + a_1 i;\ a_0, a_1 \in \mathbb{F}_q$,$b = b_0 + b_1 i;\ b_0, b_1 \in \mathbb{F}_q$
$c = c_0 + c_1 i;\ c_0, c_1 \in \mathbb{F}_q$,$d = d_0 + d_1 i;\ d_0, d_1 \in \mathbb{F}_q$
Output: $M = a.b \in \mathbb{F}_{q^2}$, $N = c.d \in \mathbb{F}_{q^2}$
1: $S_1 \leftarrow a_0 + a_1$
2: $T_1 \leftarrow b_0 + b_1$
3: $S_2 \leftarrow c_0 + c_1$
4: $T_2 \leftarrow d_0 + d_1$
5: $(x_0, y_0) \leftarrow mul_N(S_1, T_1, S_2, T_2)$
6: $(x_1, y_1) \leftarrow mul_N(a_0, b_0, a_1, b_1)$
7: $(x_2, y_2) \leftarrow mul_N(c_0, d_0, c_1, d_1)$
8: $x_0 \leftarrow x_0 - x_1$
9: $y_0 \leftarrow y_0 - y_1$
10: $m_0 \leftarrow x_1 - y_1$
11: $m_1 \leftarrow x_0 - y_1$
12: $n_0 \leftarrow x_2 - y_2$
13: $n_1 \leftarrow y_0 - y_2$
14: **return** $M = m_0 + m_1 i; N = n_0 + n_1 i;$

5.6 Squaring

Squaring is another important computation in the pairing computations on elliptic curves. As we can save computation cycles by optimizing squaring which is same as multiplication of same arguments. With lazy reduction technique efficient squaring method provide speedup to the higher level of tower computations. We have modified the F_{q^2} squaring as explained in [12] using mul_N as shown in Algorithm 5.4. The input of the algorithm is $M = M_0 + M_1 i$ where $M_0, M_1 \in \mathbb{F}_q$ and the output is $N = M^2 \in F_{q^2}$. The output is having two components

Algorithm 5.6 Multiplication in $\mathbb{F}_{q^2}$ without reduction

Inputs: $M = m_0 + m_1 i$ *and* $N = n_0 + n_1 i \in \mathbb{F}_{q^2}$;
Output: $P = M.N \in \mathbb{F}_{q^2}$
1: $(T_0, T_1) \leftarrow mul_N(m_0, n_0, m_1, n_1)$
2: $t_0 \leftarrow m_0 + m_1$, $t_1 \leftarrow n_0 + n_1$
3: $T_2 \leftarrow t_0 \times t_1$
4: $T_3 \leftarrow T_0 + T_1$
5: $T_3 \leftarrow T_2 - T_3$
6: $T_4 \leftarrow T_0 \ominus T_1$
7: **return** $P = T_4 + T_3 i$

Algorithm 5.7 Multiplication in $\mathbb{F}_{q^6}$ without reduction using NEON.

Inputs: $a = a_0 + a_1 v + a_2 v^2 \in \mathbb{F}_{q^6}$
and $b = b_0 + b_1 v + b_2 v^2 \in \mathbb{F}_{q^6}$
Output: $c = a \cdot b = c_0 + c_1 v + c_2 v^2 \in \mathbb{F}_{q^6}$
1: $t_0 \leftarrow a_1 +^2 a_2,\ t_1 \leftarrow b_1 +^2 b_2$
2: $(T_1, T_3) \leftarrow muldual_N(a_1, b_1, t_0, t_1)$
3: $t_0 \leftarrow a_0 +^2 a_1,\ t_1 \leftarrow b_0 +^2 b_1$
4: $(T_2, T_5) \leftarrow muldual_N(a_2, b_2, t_0, t_1)$
5: $T_4 \leftarrow T_1 +^2 T_2$
6: $T_{3,0} \leftarrow T_{3,0} \ominus T_{4,0}$
7: $T_{3,1} \leftarrow T_{3,1} - T_{4,1}$
8: $T_{4,0} \leftarrow T_{3,0} \ominus T_{3,1}, T_{4,1} \leftarrow T_{3,0} \oplus T_{3,1} (\equiv T_4 \leftarrow \xi \cdot T_3)$
9: $t_0 \leftarrow a_0 +^2 a_2, t_1 \leftarrow b_0 +^2 b_2$
10: $(T_0, T_6) \leftarrow muldual_N(a_0, b_0, t_0, t_1)$
11: $T_7 \leftarrow T_4 \oplus^2 T_0$
12: $T_4 \leftarrow T_0 +^2 T_1$
13: $T_{5,0} \leftarrow T_{5,0} \ominus T_{5,0}$
14: $T_{5,1} \leftarrow T_{5,1} - T_{4,1}$
15: $T_{4,0} \leftarrow T_{2,0} \ominus T_{2,1}$
16: $T_{4,1} \leftarrow T_{2,0} + T_{2,1} (steps\ 14 - 15 \equiv T_4 \leftarrow \xi \cdot T_2)$
17: $T_7 \leftarrow T_5 \oplus^2 T_4$
18: $T_4 \leftarrow T_0 +^2 T_2$
19: $T_{6,0} \leftarrow T_{6,0} \ominus T_{4,0}$
20: $T_{6,1} \leftarrow T_{6,1} - T_{4,1}$
21: $T_{8,0} \leftarrow T_{6,0} \oplus T_{1,0}$
22: $T_{8,1} \leftarrow T_{6,1} + T_{1,1}$
23: return $c = (T_7 + T_7 v + T_8 v^2)$

as $N = N_0 + N_1 i;\ N_0, N_1 \in \mathbb{F}_q$ and still follow the lazy reduction method.

- Step-1 assign substation of input components to the first element of output.
- Step-2 assigns the addition of input components to a temporary variable.
- Step-3 uses mul_N function to get multiplication of outputs of step-1 and step-2 also the components of input argument.
- Step-4 assigns the second element of output (N_1) as doubling element of step-3 multiplication of components of input arguments $M_0\, and\, M_1$.
- Step-5 assigns the first element of output as difference of outputs of step-3.

- Step-6 return the result of algorithm.

Algorithm 5.8 Inversion over $\mathbb{F}_{q^2}$ employing lazy reduction technique using NEON.

Inputs: $a = a_0 + a_1 i;\ a_0, a_1 \in \mathbb{F}_q$;
β is a quadratic non-residue over $\mathbb{F}_q$
Output: $c = a^{-1} \in \mathbb{F}_{q^2}$
1: $(T_0, T_1) \leftarrow mul_N(a_0, a_0, a_1, a_1)$
2: $T_1 \leftarrow -\beta \cdot T_1$
3: $T_0 \leftarrow T_0 + T_1$
4: $t_0 \leftarrow T_0 \bmod q$
5: $t_0 \leftarrow t_0^{-1} \bmod q$
6: $(c_0, c_1) \leftarrow mulRed_N(a_0, t_0, a_1, t_0)$
7: $c_1 \leftarrow -c_1$
8: **return** $c = c_0 + c_1 i$

5.7 Inversion

We have modified the $\mathbb{F}_{q^2}$inversion algorithm as explained in [12] using NEON multipliers as shown in Algorithm 5.8. The input of the algorithm is $a = a_0 + a_1 i;\ a_0, a_1 \in \mathbb{F}_q$ and the output is $c = a^{-1} \in \mathbb{F}_{q^2}$. The output is having two components as $c = c_0 + c_1 i;\ c_0, c_1 \in \mathbb{F}_q$ and still follow the lazy reduction method. The steps for Algorithm 5.8 are explained as:

- Step-1 computes multiplications of two components of input argument and assign to temp variables T_0 and T_1.
- Step-2 multiplies the quadratic non-residue over $\mathbb{F}_q$ with T_1.
- Step-3 assign addition of srep-1 results to T_0.
- Step-4 assign another temporary variable as mod of T_0 and step-5 takes the modulus-inversion of variable and assign back to it.
- Step-6 uses function $mulRed_N$, which is parallel CIOS multiplication of arguments using NEON.
- Step-7 negate the second component of output value and step-8 return the inverse value of input in $\mathbb{F}_{q^2}$.

Above algorithms shows the improvement of computation timing as using NEON based algorithms gives simultaneous execution on data. The next chapter shows the faster timing results of different cryptographic algorithm and finally the pairings after using NEON instruction for different level computations.

Chapter 6

Results and comparison

This section present the operation counts of Miller loop for differnt security levels and comparision of differnt implementations of multipliers for differnt field sizes. Finally, we present the faster timing results of pairing computations.

6.1 Miller Loop Operations

We provide here detailed operation counts for our algorithms on the BN-254, BN-446, and BN-638 curves used in Acar et. al [18] and defined in [45]. Table 6.1 provides the operation counts for all component operations. Numbers for BN-446 and BN-638 are the same except where indicated.

For BN-254, using the techniques described above, the projective pairing Miller loop executes one negation in $\mathbb{F}_q$, one first doubling with line evaluation, 63 point doublings with line evaluations, 6 point additions with line evaluations, one q-power Frobenius in $E'(\mathbb{F}_{q^2})$, one q^2-power Frobenius in $E'(\mathbb{F}_{q^2})$, 66 sparse multiplications, 63 squarings in $\mathbb{F}_{q^2}$, 1 negation in $E'(\mathbb{F}_{q^2})$, 2 sparser (i.e. sparse-sparse) multiplications [37], and 1 multiplication in $\mathbb{F}_{q^{12}}$. Using Table 6.1, we compute the total number of operations required in the Miller loop using homogeneous projective coordinates to be

Table 6.1: Operation counts for 254-bit, 446-bit, and 638-bit prime fields

$E'(\mathbb{F}_{q^2})$ **Arithmetic**	254-bit	446-bit/638-bit
Doubl/Eval (Projective) ARM	$2\tilde{m}_u + 7\tilde{s}_u + 8\tilde{r} + 25\tilde{a} + 4m$	$2\tilde{m}_u + 7\tilde{s}_u + 8\tilde{r} + 34\tilde{a} + a + 4m$
Doubl/Eval (Projective) x-86 64	$3\tilde{m}_u + 6\tilde{s}_u + 8\tilde{r} + 21\tilde{a} + 4m$	$3\tilde{m}_u + 6\tilde{s}_u + 8\tilde{r} + 30\tilde{a} + a + 4m$
Doubl/Eval (Affine)	$\tilde{i} + 3\tilde{m}_u + 2\tilde{s}_u + 5\tilde{r} + 7\tilde{a} + 2m$	$\tilde{i} + 3\tilde{m}_u + 2\tilde{s}_u + 5\tilde{r} + 7\tilde{a} + 2m$
Add./Eval (Projective)	$11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 10\tilde{a} + 4m$	$11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 10\tilde{a} + 4m$
Add/Eval (Affine)	$\tilde{i} + 3\tilde{m}_u + \tilde{s}_u + 4\tilde{r} + 6\tilde{a} + 2m$	$\tilde{i} + 2\tilde{m}_u + \tilde{s}_u + 3\tilde{r} + 6\tilde{a} + 2m$
First doubl./Eval	$3\tilde{m}_u + 4\tilde{s}_u + 7\tilde{r} + 14\tilde{a} + 4m$	$3\tilde{m}_u + 4\tilde{s}_u + 7\tilde{r} + 23\tilde{a} + a + 4m$
q-power Frobenius	$2\tilde{m} + 2a$	$8\tilde{m} + 2a$
q^2- power Frobenius	$4m$	$16\tilde{m} + 4a$
$\mathbb{F}_{q^2}$ **Arithmetic**	254-bit	446-bit/638-bit
Add/Subtr./Nega.	$\tilde{a} = 2a$	$\tilde{a} = 2a$
Multiplication	$\tilde{m} = \tilde{m}_u + \tilde{r} = 3m_u + 2r + 8a$	$\tilde{m} = \tilde{m}_u + \tilde{r} = 3m_u + 2r + 10a$
Squaring	$\tilde{s} = \tilde{s}_u + \tilde{r} = 2m_u + 2r + 3a$	$\tilde{s} = \tilde{s}_u + \tilde{r} = 2m_u + 2r + 5a$
Multiplication by β	$m_b = a$	$m_b = 2a$
Multiplication by ξ	$m_\xi = 2a$	$m_\xi = 3a$
Inversion	$\tilde{i} = i + 2m_u + 2s_u + 3r + 3a$	$\tilde{i} = i + 2m_u + 2s_u + 3r + 5a$
$\mathbb{F}_{q^{12}}$ **Arithmetic**	254-bit	446-bit/638-bit
Multiplication	$18\tilde{m}_u + 110\tilde{a} + 6\tilde{r}$	$18\tilde{m}_u + 117\tilde{a} + 6\tilde{r}$
Sparse Multiplication	$13\tilde{m}_u + 6\tilde{r} + 48\tilde{a}$	$13\tilde{m}_u + 6\tilde{r} + 54\tilde{a}$
Sparser Multiplication	$6\tilde{m}_u + 6\tilde{r} + 13\tilde{a}$	$6\tilde{m}_u + 6\tilde{r} + 14\tilde{a}$
Affine Sparse Multiplication	$10\tilde{m}_u + 6\tilde{r} + 47\tilde{a} + 6m_u + a$	$10\tilde{m}_u + 53\tilde{a} + 6\tilde{r} + 6m_u + a$
Squaring	$12\tilde{m}_u + 6\tilde{r} + 73\tilde{a}$	$12\tilde{m}_u + 6\tilde{r} + 78\tilde{a}$
Cyclotomic Squaring	$9\tilde{s}_u + 46\tilde{a} + 6\tilde{r}$	$9\tilde{s}_u + 49\tilde{a} + a + 6\tilde{r}$
Simult. Decompression	$9\tilde{m} + 6\tilde{s} + 22\tilde{a} + \tilde{i}$	$9\tilde{m} + 6\tilde{s} + 24\tilde{a} + \tilde{i}$ (BN-446) $16\tilde{m} + 9\tilde{s} + 35\tilde{a} + \tilde{i}$ (BN-638)
q-power Frobenius	$5\tilde{m} + 6a$	$5\tilde{m} + 6a$
q^2-power Frobenius	$10m + 2\tilde{a}$	$10m + 2\tilde{a}$
Exponentiation. by x	$45\tilde{m}_u + 378\tilde{s}_u + 275\tilde{r} + 2164\tilde{a} + \tilde{i}$	$45\tilde{m}_u + 666\tilde{s}_u + 467\tilde{r}_u + 3943\tilde{a} + \tilde{i}$ (BN-446) $70\tilde{m} + 948\tilde{s} + 675\tilde{r} + 5606\tilde{a} + 158a + \tilde{i}$ (BN-638)
Inversion	$25\tilde{m}_u + 9\tilde{s}_u + 16\tilde{r} + 121\tilde{a} + \tilde{i}$	$25\tilde{m}_u + 9\tilde{s}_u + 18\tilde{r} + 138\tilde{a} + \tilde{i}$
Compressed Squaring	$6\tilde{s}_u + 31\tilde{a} + 4\tilde{r}$	$6\tilde{s}_u + 33\tilde{a} + a + 4\tilde{r}$

$$
\begin{aligned}
\text{ML254P} = {} & a + 3\tilde{m}_u + 7\tilde{r} + 14\tilde{a} + 4m + \\
& 63(2\tilde{m}_u + 7\tilde{s}_u + 8\tilde{r} + 25\tilde{a} + 4m) + \\
& 6(11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 10\tilde{a} + 4m) + \\
& 2\tilde{m} + 2a + 4m + 66(\tilde{m}_u + 6\tilde{r} + 48\tilde{a}) + \\
& 63(12\tilde{m}_u + 6\tilde{r} + 73\tilde{a}) + \tilde{a} + \\
& 2(6\tilde{m}_u + 6\tilde{r} + 13\tilde{a}) + 18\tilde{m}_u + 110\tilde{a} + 6\tilde{r} \\
& = 1841\tilde{m}_u + 457\tilde{s}_u + 1371\tilde{r} + 9516\tilde{a} + 284m + 3a.
\end{aligned}
$$

For the curve BN-446, the Miller loop executes one negation in $\mathbb{F}_q$ to precompute $\overline{y_P}$, one first doubling with line evaluation, 111 point doublings with line evaluations, 6 point additions with line evaluations, 2 q-power Frobenius in $E'(\mathbb{F}_{q^2})$, 114 sparse multiplications, 111 squarings in $\mathbb{F}_{q^2}$, 1 negation in $E'(\mathbb{F}_{q^2})$, 2 sparser multiplications, and 1 multiplication in $\mathbb{F}_{q^{12}}$. Thus, the cost of the Miller loop on an ARM processor using projective coordinates is:

$$\begin{aligned}
\text{ML446P} &= a+3\tilde{m}_u+4\tilde{s}_u+7\tilde{r}+23\tilde{a}+a+4m+\\
&111(2\tilde{m}_u+7\tilde{s}_u+8\tilde{r}+34\tilde{a}+a+4m)+\\
&6(11\tilde{m}_u+2\tilde{s}_u+11\tilde{r}+11\tilde{a}+4m)+\\
&2(8\tilde{m}+2a)+114(13\tilde{m}_u+6\tilde{r}+54\tilde{a})+\\
&111(12\tilde{m}_u+6\tilde{r}+78\tilde{a})+\tilde{a}+\\
&+2(6\tilde{m}_u+6\tilde{r}+14\tilde{a})+18\tilde{m}_u+117\tilde{a}+6\tilde{r}\\
&= 3151\tilde{m}_u+793\tilde{s}_u+2345\tilde{r}+18601\tilde{a}+472m+117a
\end{aligned}$$

Finally, for the curve BN-638, we use the NAF Miller algorithm. Hence, the Miller loop executes one negation in $\mathbb{F}_q$ to precompute $\overline{y_P}$, one first doubling with line evaluation, 160 point doublings with line evaluations, 8 point additions with line evaluations, 2 q-power Frobenius in $E'(\mathbb{F}_{q^2})$, 167 sparse multiplications, 160 squarings in $\mathbb{F}_{q^2}$, 2 negations in $E'(\mathbb{F}_{q^2})$, 2 sparser multiplications, and 1 multiplication in $\mathbb{F}_{q^{12}}$. Thus, the cost of the Miller loop is:

$$\begin{aligned}
\text{ML638P} &= a+3\tilde{m}_u+4\tilde{s}_u+7\tilde{r}+23\tilde{a}+a+4m+\\
&160(2\tilde{m}_u+7\tilde{s}_u+8\tilde{r}+34\tilde{a}+a+4m)+\\
&8(11\tilde{m}_u+2\tilde{s}_u+11\tilde{r}+11\tilde{a}+4m)+\\
&2(8\tilde{m}+2a)+167(13\tilde{m}_u+6\tilde{r}+54\tilde{a})\\
&160(12\tilde{m}_u+6\tilde{r}+78\tilde{a})+2\tilde{a}+\\
&2(6\tilde{m}_u+6\tilde{r}+14\tilde{a})+18\tilde{m}_u+117\tilde{a}+6\tilde{r}\\
&= 4548\tilde{m}_u+1140\tilde{s}_u+3557\tilde{r}+27206\tilde{a}+676m+166a
\end{aligned}$$

6.2 Implementation Timings

The computational cost of the addition, multiplication, squaring and inversion operations over F_q and F_{q^2} are represented by (a,m,s,i) and $(\tilde{a},\tilde{m},\tilde{s},\tilde{i})$, respectively. Table 6.2 present the timing results for previous efforts using affine and projective coordinates.

Table 6.2: Timings for affine and projective pairing on different ARM processors and comparisons with prior literature. Times for the Miller loop (ML) in each row reflect those of the faster pairing.

NVidia Tegra 2 (ARM v7) Cortex-A9 at 1.0 GHz[18]														
Field Size	Language	Operation Timing [cc]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML ($\times 10^3$)	FE ($\times 10^3$)	O-A(a) ($\times 10^3$)	O-A(p) ($\times 10^3$)
254-bit	ASM	670	1720	-	18,350	10.7	1,420	8,180	5,200	26,610	-	24,690	51,010	55,190
446-bit		1,170	4,010	-	35,850	8.9	2,370	17,240	10,840	54,230	-	86,750	184,280	195,560
638-bit		1,710	8,220	-	56,090	6.8	3,480	31,810	20,550	535,420	-	413,370	649,850	768,060
Galaxy Nexus (ARM v7) TI OMAP 4460 Cortex-A9 at 1.2 GHz [12]														
Field Size	Language	Operation Timing [cc]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML ($\times 10^3$)	FE ($\times 10^3$)	O-A(a) ($\times 10^3$)	O-A(p) ($\times 10^3$)
254-bit	ASM	60	1,116	660	11,304	10.1	120	2,952	2,484	16,548	7,376	4,509	12,687	11,886
	C	84	1,176	636	11,544	9.8	156	3,372	2,532	16,860	8,230	5,258	14,206	13,489
446-bit		144	2,832	1,524	27,696	9.8	264	7,548	6,204	38,724	30,950	16,502	47,863	47,452
638-bit		228	5,844	3,660	46,140	7.9	540	14,640	12,468	68,136	78,837	40,389	119,227	119,359
Arndale Board (ARM v7) Cortex-A15 at 1.7 GHz [SAC ext.]														
Field Size	Language	Operation Timing [cc]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML ($\times 10^3$)	FE ($\times 10^3$)	O-A(a) ($\times 10^3$)	O-A(p) ($\times 10^3$)
254-bit	N	51	697	340	8,653	12.4	85	1,377	782	10,302	3,964	2,351	6,538	6,089
	ASM	50	663	338	8,657	13.1	85	1,785	1,564	11,220	4,996	2,961	8,260	7,886
	C	51	765	340	8,887	11.6	102	2,091	1,567	11,628	5,499	3,291	9,130	9,001
446-bit	ASM	81	1,581	934	19,193	11.29	164	4,539	3,655	25,313	10,924	10,205	30,664	30,069
	C	81	1,734	935	19,363	11.39	165	5,015	4,063	25,908	11,788	10,891	32,823	31,898
638-bit	C	117	3570	1,700	31,671	8.87	221	9,707	7,871	44,217	20,673	27,018	81,239	79,843
Arndale Board (ARM v7) Cortex-A15 at 1.7 GHz [19]														
Field Size	Language	Operation Timing [cc]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML ($\times 10^3$)	FE ($\times 10^3$)	O-A(a) ($\times 10^3$)	O-A(p) ($\times 10^3$)
254-bit	N	-	-	-	-	-	140	1,360	860	29,010	3,388	2,353	-	5,838

Table 6.3: Timing Comparision for Multiplier of diffrerent fields

Arndale Board (ARM v7) Cortex-A15 at 1.7 GHz						
Field Size	Operation Timing (*cc*)					
	GMP	Comba (ASM)	Revisited Montgomery(N)	Schoolbook(N)	Karatsuba (N)	Karatsuba (ASM)
254-bit	355	249	193	324	435	448

6.2.1 Timing comparison for different multipliers

Pairing algorithm supports basic multiplier at the lowest level computation which gives flexibility to use and analyze different multipliers present in literature. In this section we are presenting performance of different multipliers which can be used to implement lower level computations. For 256-bit multiplier, we are comparing the libgmp based multiplier, assembly implementation of Comba multiplier 3.4 , Hwajeong Seo's SOS multiplier implementation [22], NEON engine based implementation of Schoolbook multiplier 5.2 and NEON based Karatsuba multiplier 5.3. We have also implemented Karatsuba multiplier with assembly multiplier for $\frac{n}{2} - bit$ multiplication in the algorithm 5.3.

- Table 6.3 presents the timing comparison of different multipliers on Arndale developement board for 254-bit field size. We can observe the performance of NEON implementation of Revisited Montgomery multiplier [22] is the best among the multipliers. Karatsuba multiplier is having the least performance timing. The reason of Karatsuba multiplier's least performance is that NEON implementation provide parallel implementation to the $\frac{n}{2} - bit$ multiplication but the third step of multiplier needs assembly implementation to combine the results of two $\frac{n}{2} - bit$ multipliers as explained in Algorithm 5.3.

- Table 6.4 shows the comparison for different multipliers in 446-bit and 638-bit fields. Our NEON implementation of Schoolbook multiplier 5.3 and assembly implementation of Comba multiplier preform faster than GMP library for 446-bit and 638-bit. The comparision shows that Schoolbook(N) multiplier performance is fastest for both of the fields.

- Table 6.5 presents the comparison for schoolbook multiplier for assembly and NEON engine based implementation for 256-bit field for ARMv8 architecture.

Table 6.4: Timings Comparison for Multiplier of 446-bit and 638-bit field size.

Arndale Board (ARM v7) Cortex-A15 at 1.7 GHz				
Field Size	Operation Timing (*cc*)			
	GMP	Comba (ASM)	Schoolbook(N)	Karatsuba (N)
446-bit	860	689 [46]	596	822
638-bit	1625	1349	1115	1382

Table 6.5: Timings comparison for Multiplier for ARMv8 architecture.

Linaro HiKey - 96 Boards (ARM v8) Cortex-A53 at 1.2 GHz			
Field Size	Operation Timing (*cc*)		
	GMP	Schoolbook(N)	Schoolbook (ASM)
256-bit	252	205	324

Table 6.6: Timings for Schoolbook Multipliers on Different platforms.

Timing results of Schoolbook Multiplier			
Field Size	Language	Operation Timing (*cc*)	
		Arndale	Linaro
256-bit	N	324	205
446-bit	N	596	n/a
638-bit	N	1115	n/a

6.2.2 Timing comparison for NEON based Karatsuba and Schoolbook multipliers

Table 6.6 presents the comparison of timings for NEON based schoolbook multiplier for ARMv7 (Arndale and Jetson) and ARMv8 (Liaro HiKey) for 256-bit field size multiplier. Table 6.6 also compares the timing results for BN446 and BN638 curves multiplication for ARMv7. The results shows the multipliers performs better on Jetson TK-1 architecture for all the fields in comparison with Arndale.

In this thesis, we have implemented the Karatsuba multiplier using NEON engine as explained in Algorithm 5.3. Table 6.7 represents the timing comparison of NEON based Karatsuba multiplier for different ARMv7 architectures. The results shows the performance of Karatsuba multiplier is almost same for both of the boards.

Table 6.7: Timings for Karatsuba Multiplier for different platform

Timings for Karatsuba Multiplier			
Field	Language	Operation Timing (*cc*)	
Size		Arndale	Jetson
256-bit	N	435	409
446-bit	N	822	827
638-bit	N	1382	1370

6.2.3 Timing results for pairings

We run our library on Arndale Board which is ARMv7 Cortex-A15 with 1.7 GHz processor and Linaro HiKey-96 which is ARMv8 Cortex-A53 with 1.2GHz processor. Our software is based on version 0.2.3 of the RELIC toolkit [44], modified to include our optimizations and GMP 6.0.0 is being used as back-end. For each platform, we used the standard operating system (Linaro OS) and development environment that ships with the device, namely Debian Squeeze (native C compiler).

We present the results of our trials in Table 6.8 which shows the improved results of different operations for BN254, BN446 and BN638 curves. Here are different observations from the results being presented in the table. N_C, N_S and N_K rows shows the results for timing of pairings on BN254 curve with Comba (ASM), Schoolbook(NEON) and Karatsuba (NEON) multiplier respectively in $\mathbb{F}_q$ field and NEON optimization applied in higher fields ($\mathbb{F}_{q^2}$ and $\mathbb{F}_{q^6}$).

- N_M rows in the table 6.8 shows the timing results for the pairings using Hwajeong's latest implementation [22]. NEON optimization are being applied at higher layers of tower structure which improves pairing performance. The pairing results on projective coordinate for revisited Montgomery multiplier its the recorded fastest pairing timing for BN254 as $5,470 \times 10^3$.

- A_C rows represents the timing results of pairing with assembly implementation of Comba multiplier and without the NEON optimization in higher fields for different security levels. We can observe from the results that NEON implementations are faster than assembly implementations for all the security levels which proves that NEON engine is supportive for pairings on all security levels.

- In this work, we have implemented the Comba multiplier for 638-bit prime field as explained in Algorithm 3.4 using assembly language for ARMv7 boards. The table 6.8 shows the assembly implementations (A_c row for 638-bit) are faster than C implemen-

Table 6.8: Timings for affine and projective pairings on different ARM processors and comparisons with prior literature. N_M,N_S,N_K, N_C represent the F_q multiplier used as Revisited Montgomery(NEON), Schoolbook(NEON), Karatsuba(NEON), and Comba(ASM) with NEON optimization in higher fields and A_C represents Comba(ASM) multiplier in F_q field without NEON optimization in higher fields.

Arndale Board (ARM v7) Cortex-A15 at 1.7 GHz														
Field Size	Implementation	Operation Timing (*cc*)												
		a	m	r	i	I/M	$\bar{a}$	$\bar{m}$	$\bar{s}$	$\tilde{i}$	ML ($\times 10^3$)	FE ($\times 10^3$)	O-A(a) ($\times 10^3$)	O-A(p) ($\times 10^3$)
254-bit	N_M	42	502	298	9,021	17.91	82	1,261	735	10,811	3,820	2,064	6,125	5,470
	N_C	46	560	298	8,801	15.63	86	1,331	744	11,330	3,808	2,072	6,146	5,511
	N_S	42	637	300	9,141	14.30	80	1,372	741	10,739	3,935	2,080	6,253	5,528
	N_K	43	736	298	9,244	12.51	80	1,492	736	11,039	3,984	2,083	6,301	5,908
	A_C	51	554	299	8,958	16.16	88	1,567	1,217	11,110	4,407	2,514	7,181	6,872
	C	52	787	340	8,867	11.41	102	2,105	1,561	11,634	5,500	3,292	9,123	9,002
446-bit	N_S	80	1,461	877	19,210	13.41	156	3,787	2,274	23,576	9,000	7,663	24,302	23,593
	A_C	80	1,593	918	19,200	12.05	162	4,546	3,656	25,328	10,923	10,203	30,665	30,123
	C	80	1,739	936	19,370	11.13	166	5,030	4,071	25,923	11,789	10,893	32,825	31,889
638-bit	N_S	95	2,774	1,726	31,452	11.33	192	7,640	5,623	42,833	17,613	24,000	70,403	65,012
	A_C	117	3,050	1,716	31,482	10.33	219	8,376	6,800	43,586	19,157	24,984	74,253	72,279
	C	118	3,575	1,700	31,487	8.86	222	9,414	7,678	44,233	20,673	26,018	80,243	78,843
Linaro HiKey - 96 Boards (ARM v8) Cortex-A53 at 1.2 GHz														
Field Size	Implementation	Operation Timing (*cc*)												
		a	m	r	i	I/M	$\bar{a}$	$\bar{m}$	$\bar{s}$	$\tilde{i}$	ML ($\times 10^3$)	FE ($\times 10^3$)	O-A(a) ($\times 10^3$)	O-A(p) ($\times 10^3$)
254-bit	N_S	36	360	216	9,672	28.60	72	1,354	1,164	11,844	4,310	2,505	7,066	6,765
	A_C	36	552	192	9,720	17.62	84	1,788	1,440	12,180	4,995	3,108	8,288	8,065
	C	36	480	204	9,696	20.2	84	1,500	1,176	12,084	4,646	2,889	7,833	7,563
446-bit	A_C	48	1512	456	18,624	12.31	84	4,716	3,696	25,728	11,233	11,233	32,472	32,330
	C	48	1,452	504	18,456	12.71	96	4,008	3,288	24,108	9,373	8,920	26,235	25,315
638-bit	A_C	72	3667	940	28,624	7.80	124	10,501	7,984	42,810	22,391	28,574	86,734	85,132
	C	72	2,340	960	28,632	12.23	144	7,248	5,052	38,328	15,609	20,311	61,383	61,206

tation of 638-bit pairing computations and NEON implementation further improve the resulting timings.

- The table 6.8 presents all the results for BN446 and BN638 for Arndale in rows (N_S) which is fastest for each fields. The fastest timing results are corresponding to Schoolbook(N) multiplier in $\mathbb{F}_q$ field and NEON optimizations applied using parallel SOS multiplier in $\mathbb{F}_{q^2}$.

- Lauter et al. used affine coordinates for pairings in [47] for different security levels. Later Acer el al. [18] explained that extension field inversion to multiplication ratios makes affine coordinates better choice for pairing computation than homogeneous coordinates. Later Grewal [12] presented inversion to multiplication ratio of blow 10-12 at different security levels and showed the timing for C and Assembly code implementation for affine and projectile coordinates using different security levels. As shown in Table 6.8, I/M ratio is higher than 10 for most of the computations results which makes projective coordinates as efficient to implement.

- We also run our library on Linaro HiKey-96 board which is ARMv8 Cortex-A53 architecture with clock speed as 1.2GHz. The C-language based results are shown in Table6.8. We have implemented Comba multiplier as explained in 3.4 using ARMv8 assembly for BN254 and BN446. The results are being represented in the table. The assembly implementations could not improve the timings for pairings on ARMv8 board. The reason of the slow implementation results is the extra stack transitions in pure assembly for ARMv8. ARMv8 architecture doesn't support burst transaction to ad from the stack e.g., PUSH/POP which are available in ARMv7. We have to transfer the current register state on stack using LDP/STP commands which can be executed on a register couple as $LDP\ X1, X2, addr$ and result in slow implementation.

- We have also tried to run the BN254 curve NEON based pairing implementation on ARMv8 Linaro Jetson TK-1 board. The results are being shown in Table 6.8. The timing results of BN-254 curve based computations is faster as 10% of its C-implementation. But all the assembly and NEON assembly codes have to be rewritten for the further improvement as future work

6.2.4 Speed records for pairings computations on different security levels

Our results show the speed records of pairing on different curves compared to previously avaialble results. As shown in table 6.8, pairings with revisited Montgomery [22] field multiplier and NEON optimization in higher fields ($\mathbb{F}_{q^2}$ and $\mathbb{F}_{q^6}$) gives the record fastest timing of $5,470 \times 10^3$ clock cycles which is 6% faster than the previous fastest result for same field [19].

This work also shows the speed record of pairing computations for BN446 and BN638 curves as $23,593 \times 10^3$ clock cycles and $65,012 \times 10^3$ respectively which are 45% and 50% improved timings over previous fastest known timings[12] and [18].

Chapter 7

Conclusion and Future Work

We have modified the cryptographic library used in [12] which is based on (RELIC, an Efficient Library for Cryptography) RELIC [44], a platform independent library for cryptographic algorithms. We used NEON engine to improve timings for ARM processor based embedded systems which gives higher speed to Optimal-Ate pairing on BN254, BN446 and BN638 curves. The main changes made to the library is modular multiplication using NEON engine and modification in $\mathbb{F}_{q^2}$ and $\mathbb{F}_{q^6}$ field algorithms which result in improvement of timings results for pairing computations. Our pairing computations are fastest among the implementations present in the literature for all three security levels. We present the results of Optimal-Ate pairing for both Affine and Projective coordinates and pairing using projective coordinate are faster than affine for all the mentioned curves. We can conclude from the experimental results that NEON can offer performance improvement for pairing based cryptographic algorithms irrespective of security level.

As ARM introduced 64-bit architecture in ARMv8, the future optimization of this work would be pairing computations on ARMv8 architecture. ARMv8 architecture is using new NEON instructions set and also provide crypto extension to the computations which could be halpful to further optimize the basic algorithms and pairing computations.

References

[1] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., 2003.

[2] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, pp. 644–654, Sept. 2006.

[3] A. Joux, "A one round protocol for tripartite diffie-hellman," in *Proceedings of the 4th International Symposium on Algorithmic Number Theory*, ANTS-IV, (London, UK, UK), pp. 385–394, Springer-Verlag, 2000.

[4] D. Boneh and M. Franklin, "Identity-based encryption from the Weil pairing," *SIAM J. of Computing*, vol. 32, no. 3, pp. 586–615, 2003. extended abstract in Crypto'01.

[5] R. Dutta, R. Barua, and P. Sarkar, "Pairing-based cryptography: A survey."

[6] B. Waters, "Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization," in *Proceedings of the 14th International Conference on Practice and Theory in Public Key Cryptography Conference on Public Key Cryptography*, PKC'11, pp. 53–70, 2011.

[7] R. Schoof, *Elliptic curves over finite fields and the computation of square roots mod p*. Report. Department of Mathematics. University of Amsterdam, Department, Univ., 1983.

[8] V. S. Miller, "The Weil pairing, and its efficient calculation," *J. Cryptology*, vol. 17, no. 4, pp. 235–261, 2004.

[9] F. Hess, N. P. Smart, and F. Vercauteren, "The eta pairing revisited.," *IACR Cryptology ePrint Archive*, vol. 2006, p. 110, 2006.

[10] F. Vercauteren, "Optimal pairings," *IEEE Transactions on Information Theory*, vol. 56, no. 1, pp. 455–461, 2010.

[11] D. J. Bernstein and T. Lange, "Performance evaluation of a new coordinate system for elliptic curves," 2007.

[12] G. Grewal, R. Azarderakhsh, P. Longa, S. Hu, and D. Jao, "Efficient Implementation of Bilinear Pairings on ARM Processors," in *Selected Areas in Cryptography* (L. R. Knudsen and H. Wu, eds.), vol. 7707 of *Lecture Notes in Computer Science*, pp. 149–165, 2012.

[13] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *Selected Areas in Cryptography* (B. Preneel and S. E. Tavares, eds.), vol. 3897 of *Lecture Notes in Computer Science*, pp. 319–331, Springer, 2005.

[14] 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*, pp. 21–39, 2010.

[15] D. F. Aranha, P. S. L. M. Barreto, P. Longa, and J. E. Ricardini, "The realm of the pairings," in *20th International Conference on Selected Areas in Cryptography - SAC 2013* (T. Lange, K. E. Lauter, and P. Lisonek, eds.), vol. 8282, pp. 3–25, Springer, 2013.

[16] A. J. Devegili, M. Scott, and R. Dahab, "Implementing cryptographic pairings over barreto-naehrig curves," in *Pairing-Based Cryptography - Pairing 2007, First International Conference, Tokyo, Japan, July 2-4, 2007, Proceedings* (T. Takagi, T. Okamoto, E. Okamoto, and T. Okamoto, eds.), vol. 4575 of *Lecture Notes in Computer Science*, pp. 197–207, Springer, 2007.

[17] D. F. Aranha, L. Fuentes-Castañeda, E. Knapp, A. Menezes, and F. Rodríguez-Henríquez, "Implementing pairings at the 192-bit security level," in *Pairing-Based Cryptography - Pairing 2012 - 5th International Conference, Cologne, Germany, May 16-18, 2012, Revised Selected Papers* (M. Abdalla and T. Lange, eds.), vol. 7708 of *Lecture Notes in Computer Science*, pp. 177–195, Springer, 2012.

[18] T. Acar, K. Lauter, M. Naehrig, and D. Shumow, "Affine Pairings on ARM," in *Pairing-Based Cryptography - Pairing 2012 - 5th International Conference, Cologne, Germany, May 16-18, 2012, Revised Selected Papers* (M. Abdalla and T. Lange, eds.), vol. 7708 of *Lecture Notes in Computer Science*, pp. 203–209, 2012.

[19] A. H. Sánchez and F. Rodríguez-Henríquez, "NEON Implementation of an Attribute-Based Encryption Scheme," in *Applied Cryptography and Network Security - 11th In-*

ternational Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings (M. J. J. Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, eds.), vol. 7954 of *Lecture Notes in Computer Science*, pp. 322–338, 2013.

[20] A. Joux and V. Vitse, "Elliptic curve discrete logarithm problem over small degree extension fields - application to the static diffie-hellman problem on $e(\mathbb{F}_{q^{5}})$," *J. Cryptology*, vol. 26, no. 1, pp. 119–143, 2013.

[21] D. J. Bernstein and P. Schwabe, "NEON crypto," in *14th International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2012* (E. Prouff and P. Schaumont, eds.), vol. 7428, pp. 320–339, Springer, 2012.

[22] H. Seo, Z. Liu, J. Großschädl, J. Choi, and H. Kim, "Montgomery modular multiplication on ARM-NEON revisited," *IACR Cryptology ePrint Archive*, vol. 2014, p. 760, 2014.

[23] c. K. Koç, T. Acar, and B. S. Kaliski, Jr., "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, June 1996.

[24] D. J. Bernstein, C. Chuengsatiansup, and T. Lange, "Curve41417: Karatsuba revisited," in *CHES'14*, pp. 316–334, 2014.

[25] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics Doklady*, vol. 7, p. 595, Jan. 1963.

[26] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 1996.

[27] D. Jao, "Elliptic curve cryptography." http://goo.gl/wwN5aJ, 2010.

[28] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott, "Efficient algorithms for pairing-based cryptosystems," in *Proceedings of the 22Nd Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '02, (London, UK, UK), pp. 354–368, Springer-Verlag, 2002.

[29] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Proceedings of the 22Nd International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'03, (Berlin, Heidelberg), pp. 416–432, Springer-Verlag, 2003.

[30] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '01, (London, UK, UK), pp. 514–532, Springer-Verlag, 2001.

[31] N. Benger and M. Scott, "Constructing tower extensions of finite fields for implementation of pairing-based cryptography," in *Arithmetic of Finite Fields* (M. Hasan and T. Helleseth, eds.), vol. 6087 of *Lecture Notes in Computer Science*, pp. 180–195, Springer Berlin Heidelberg, 2010.

[32] IEEE Std. 1363-3/D1, ""Draft Standard for Identity-based public key cryptography using pairings"," January 2008.

[33] D. Freeman, M. Scott, and E. Teske, "A taxonomy of pairing-friendly elliptic curves," *J. Cryptology*, 2010.

[34] R. Balasubramanian and N. Koblitz, "The improbability that an elliptic curve has subexponential discrete log problem under the menezes - okamoto - vanstone algorithm," *J. Cryptology*, 1998.

[35] S. D. Galbraith, X. Lin, and M. Scott, "Endomorphisms for faster elliptic curve cryptography on a large class of curves," in *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, (Berlin, Heidelberg), pp. 518–535, Springer-Verlag, 2009.

[36] G. Grewal, "Efficient pairings on various platforms.." https://uwspace.uwaterloo.ca/handle/10012/6722, 2012.

[37] 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 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings* (K. G. Paterson, ed.), vol. 6632 of *Lecture Notes in Computer Science*, pp. 48–68, Springer, 2011.

[38] P. G. Comba, "Exponentiation cryptosystems on the ibm pc," *IBM Syst. J.*, vol. 29, pp. 526–538, Oct. 1990.

[39] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 1996.

[40] P. Barrett, "Implementing the rivest shamir and adleman public keyencryption algorithm on a standard digital signal processor," vol. 263 of *Lecture Notes in Computer Science*, pp. 311–323, 1987.

[41] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[42] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[43] H. Seo, Z. Liu, J. Großschädl, and H. Kim, "Efficient arithmetic on ARM-NEON and its application for high-speed RSA implementation," *IACR Cryptology ePrint Archive*, 2015.

[44] D. F. Aranha and C. P. L. Gouvêa, "RELIC is an Efficient LIbrary for Cryptography." http://code.google.com/p/relic-toolkit.

[45] C. C. F. P. Geovandro, M. A. S. Jr., M. Naehrig, and P. S. L. M. Barreto, "A family of implementation-friendly BN elliptic curves," *Journal of Systems and Software*, vol. 84, no. 8, pp. 1319–1326, 2011.

[46] D. Fishbein, "Machine-level software optimization of cryptographic protocols.." https://uwspace.uwaterloo.ca/handle/10012/8400, 2014.

[47] K. Lauter, P. L. Montgomery, and M. Naehrig, "An analysis of affine coordinates for pairing computation," in *Pairing* (M. Joye, A. Miyaji, and A. Otsuka, eds.), vol. 6487 of *Lecture Notes in Computer Science*, pp. 1–20, Springer, 2010.

Appendix A

Code for 128-bit SOS multiplier

```
void MUL_128_NEON(uint32_t *C, uint32_t *C1, uint32_t *A,
                                uint32_t *B, uint32_t *A1, uint32_t *B1)
    {
        uint32x2_t b[4],a;
        uint64x2_t temp_mul[5], mulCarry, low;
        uint32x2_t val;
        uint32_t temp_B[]={B[0],B1[0],B[1],B1[1],B[2],B1[2],B[3],B1[3]};
        uint32_t temp_A[]={A[0],A1[0],A[1],A1[1],A[2],A1[2],A[3],A1[3]};
        uint64_t low_digit[]={0x00000000FFFFFFFFUL,0x00000000FFFFFFFFUL};
        b[0]=vld1_u32(temp_B);
        b[1]=vld1_u32(&temp_B[2]);
        b[2]=vld1_u32(&temp_B[4]);
        b[3]=vld1_u32(&temp_B[6]);
        low=vld1q_u64(low_digit);
    //First Iteration
        a=vld1_u32(temp_A);
        temp_mul[0]=vmull_u32(a,b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_u32(mulCarry,a,b[1]);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        temp_mul[2]=vmlal_u32(mulCarry,a,b[2]);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
        temp_mul[3]=vmlal_u32(mulCarry,a,b[3]);
        temp_mul[4]=vshrq_n_u64(temp_mul[3],32);
```

```
        val=vmovn_u64(temp_mul[0]);
        C[0]=vget_lane_u32(val,0);
        C1[0]=vget_lane_u32(val,1);
    //Second Iteration
        a=vld1_u32(&temp_A[2]);
        temp_mul[0]=vandq_u64(temp_mul[0],low);
        temp_mul[1]=vandq_u64(temp_mul[1],low);
        temp_mul[2]=vandq_u64(temp_mul[2],low);
        temp_mul[3]=vandq_u64(temp_mul[3],low);
        temp_mul[0]=vmlal_u32(temp_mul[1],a,b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_u32(temp_mul[2],a,b[1]);
        temp_mul[1]=vaddq_u64(temp_mul[1],mulCarry);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        temp_mul[2]=vmlal_u32(temp_mul[3],a,b[2]);
        temp_mul[2]=vaddq_u64(temp_mul[2],mulCarry);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
        temp_mul[3]=vmlal_u32(temp_mul[4],a,b[3]);
        temp_mul[3]=vaddq_u64(temp_mul[3],mulCarry);
        temp_mul[4]=vshrq_n_u64(temp_mul[3],32);
        val=vmovn_u64(temp_mul[0]);
        C[1]=vget_lane_u32(val,0);
        C1[1]=vget_lane_u32(val,1);
    //third Iteration
        a=vld1_u32(&temp_A[4]);
        temp_mul[0]=vandq_u64(temp_mul[0],low);
        temp_mul[1]=vandq_u64(temp_mul[1],low);
        temp_mul[2]=vandq_u64(temp_mul[2],low);
        temp_mul[3]=vandq_u64(temp_mul[3],low);
        temp_mul[0]=vmlal_u32(temp_mul[1],a,b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_u32(temp_mul[2],a,b[1]);
        temp_mul[1]=vaddq_u64(temp_mul[1],mulCarry);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        temp_mul[2]=vmlal_u32(temp_mul[3],a,b[2]);
```

```
        temp_mul[2]=vaddq_u64(temp_mul[2],mulCarry);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
        temp_mul[3]=vmlal_u32(temp_mul[4],a,b[3]);
        temp_mul[3]=vaddq_u64(temp_mul[3],mulCarry);
        temp_mul[4]=vshrq_n_u64(temp_mul[3],32);
        val=vmovn_u64(temp_mul[0]);
        C[2]=vget_lane_u32(val,0);
        C1[2]=vget_lane_u32(val,1);
    //Fourth Iteration
        a=vld1_u32(&temp_A[6]);
        temp_mul[0]=vandq_u64(temp_mul[0],low);
        temp_mul[1]=vandq_u64(temp_mul[1],low);
        temp_mul[2]=vandq_u64(temp_mul[2],low);
        temp_mul[3]=vandq_u64(temp_mul[3],low);
        temp_mul[0]=vmlal_u32(temp_mul[1],a,b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_u32(temp_mul[2],a,b[1]);
        temp_mul[1]=vaddq_u64(temp_mul[1],mulCarry);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        temp_mul[2]=vmlal_u32(temp_mul[3],a,b[2]);
        temp_mul[2]=vaddq_u64(temp_mul[2],mulCarry);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
        temp_mul[3]=vmlal_u32(temp_mul[4],a,b[3]);
        temp_mul[3]=vaddq_u64(temp_mul[3],mulCarry);
        temp_mul[4]=vshrq_n_u64(temp_mul[3],32);
        val=vmovn_u64(temp_mul[0]);
        C[3]=vget_lane_u32(val,0);
        C1[3]=vget_lane_u32(val,1);

        b[0]=vmovn_u64(temp_mul[1]);
        b[1]=vmovn_u64(temp_mul[2]);
        b[2]=vmovn_u64(temp_mul[3]);
        b[3]=vmovn_u64(temp_mul[4]);

        C[4]=vget_lane_u32(b[0],0);
```

```
        C1[4]=vget_lane_u32(b[0],1);
        C[5]=vget_lane_u32(b[1],0);
        C1[5]=vget_lane_u32(b[1],1);
        C[6]=vget_lane_u32(b[2],0);
        C1[6]=vget_lane_u32(b[2],1);
        C[7]=vget_lane_u32(b[3],0);
        C1[7]=vget_lane_u32(b[3],1);
}
```

Appendix B

Pseudo Code: 256 Reduction using NEON

```
void RED_256_NEON(uint32_t *T,uint32_t *T2, uint32_t *t, uint32_t *t2)
    {
        uint64x2_t temp_mul[5], mulCarry;
        uint32x2_t b[5];
        uint32x2_t val;
        uint32_t temp_B[]={t[0],t2[0],t[1],t2[1],t[2],t2[2],t[3],t2[3]};
        uint32_t temp_A[]={t[4],t2[4],t[5],t2[5],t[6],t2[6],t[7],t2[7]};
        b[0]=vld1_u32(temp_B);
        b[1]=vld1_u32(&temp_B[2]);
        b[2]=vld1_u32(&temp_B[4]);
        b[3]=vld1_u32(&temp_B[6]);
        b[4]=vld1_u32(temp_A);

    //First Iteration
        mulCarry=vmull_n_u32(b[0],pi[0]);
        val=vmovn_u64(mulCarry);
        temp_mul[0]=vmull_n_u32(val,p[0]);
        temp_mul[0]=vaddw_u32(temp_mul[0],b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_n_u32(mulCarry,val,p[1]);
        temp_mul[1]=vaddw_u32(temp_mul[1],b[1]);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        b[0]=vmovn_u64(temp_mul[1]);
        temp_mul[2]=vmlal_n_u32(mulCarry,val,p[2]);
```

```
        temp_mul[2]=vaddw_u32(temp_mul[2],b[2]);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
        b[1]=vmovn_u64(temp_mul[2]);
        temp_mul[3]=vmlal_n_u32(mulCarry,val,p[3]);
        temp_mul[3]=vaddw_u32(temp_mul[3],b[3]);
        b[2]=vmovn_u64(temp_mul[3]);
        temp_mul[4]=vshrq_n_u64(temp_mul[3],32);
        temp_mul[4]=vaddw_u32(temp_mul[4],b[4]);
        b[3]=vmovn_u64(temp_mul[4]);
        temp_mul[4]=vshrq_n_u64(temp_mul[4],32);
        b[4]=vld1_u32(&temp_A[2]);
    //Second Iteration
        mulCarry=vmull_n_u32(b[0],pi[0]);
        val=vmovn_u64(mulCarry);
        temp_mul[0]=vmull_n_u32(val,p[0]);
        temp_mul[0]=vaddw_u32(temp_mul[0],b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_n_u32(mulCarry,val,p[1]);
        temp_mul[1]=vaddw_u32(temp_mul[1],b[1]);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        b[0]=vmovn_u64(temp_mul[1]);
        temp_mul[2]=vmlal_n_u32(mulCarry,val,p[2]);
        temp_mul[2]=vaddw_u32(temp_mul[2],b[2]);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
        b[1]=vmovn_u64(temp_mul[2]);
        temp_mul[3]=vmlal_n_u32(mulCarry,val,p[3]);
        temp_mul[3]=vaddw_u32(temp_mul[3],b[3]);
        mulCarry=vshrq_n_u64(temp_mul[3],32);
        b[2]=vmovn_u64(temp_mul[3]);
        temp_mul[4]=vaddq_u64(temp_mul[4],mulCarry);
        temp_mul[4]=vaddw_u32(temp_mul[4],b[4]);
        b[3]=vmovn_u64(temp_mul[4]);
        temp_mul[4]=vshrq_n_u64(temp_mul[4],32);
        b[4]=vld1_u32(&temp_A[4]);
    //Third Iteration
```

```
        mulCarry=vmull_n_u32(b[0],pi[0]);
        val=vmovn_u64(mulCarry);
        temp_mul[0]=vmull_n_u32(val,p[0]);
        temp_mul[0]=vaddw_u32(temp_mul[0],b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_n_u32(mulCarry,val,p[1]);
        temp_mul[1]=vaddw_u32(temp_mul[1],b[1]);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        b[0]=vmovn_u64(temp_mul[1]);
        temp_mul[2]=vmlal_n_u32(mulCarry,val,p[2]);
        temp_mul[2]=vaddw_u32(temp_mul[2],b[2]);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
        b[1]=vmovn_u64(temp_mul[2]);
        temp_mul[3]=vmlal_n_u32(mulCarry,val,p[3]);
        temp_mul[3]=vaddw_u32(temp_mul[3],b[3]);
        mulCarry=vshrq_n_u64(temp_mul[3],32);
        b[2]=vmovn_u64(temp_mul[3]);
        temp_mul[4]=vaddq_u64(temp_mul[4],mulCarry);
        temp_mul[4]=vaddw_u32(temp_mul[4],b[4]);
        b[3]=vmovn_u64(temp_mul[4]);
        temp_mul[4]=vshrq_n_u64(temp_mul[4],32);
        b[8]=vld1_u32(&temp_A[4]);
    //Fourth Iteration
        mulCarry=vmull_n_u32(b[0],pi[0]);
        val=vmovn_u64(mulCarry);
        temp_mul[0]=vmull_n_u32(val,p[0]);
        temp_mul[0]=vaddw_u32(temp_mul[0],b[0]);
        mulCarry=vshrq_n_u64(temp_mul[0],32);
        temp_mul[1]=vmlal_n_u32(mulCarry,val,p[1]);
        temp_mul[1]=vaddw_u32(temp_mul[1],b[1]);
        mulCarry=vshrq_n_u64(temp_mul[1],32);
        b[0]=vmovn_u64(temp_mul[1]);
        temp_mul[2]=vmlal_n_u32(mulCarry,val,p[2]);
        temp_mul[2]=vaddw_u32(temp_mul[2],b[2]);
        mulCarry=vshrq_n_u64(temp_mul[2],32);
```

```
b[1]=vmovn_u64(temp_mul[2]);
temp_mul[3]=vmlal_n_u32(mulCarry,val,p[3]);
temp_mul[3]=vaddw_u32(temp_mul[3],b[3]);
mulCarry=vshrq_n_u64(temp_mul[3],32);
b[2]=vmovn_u64(temp_mul[3]);
temp_mul[4]=vaddq_u64(temp_mul[4],mulCarry);
temp_mul[4]=vaddw_u32(temp_mul[4],b[4]);
b[3]=vmovn_u64(temp_mul[4]);

T[0]=vget_lane_u32(b[0],0);
T2[0]=vget_lane_u32(b[0],1);
T[1]=vget_lane_u32(b[1],0);
T2[1]=vget_lane_u32(b[1],1);
T[2]=vget_lane_u32(b[2],0);
T2[2]=vget_lane_u32(b[2],1);
T[3]=vget_lane_u32(b[3],0);
T2[3]=vget_lane_u32(b[3],1);

while(T > q){T ← T − q;}
while(T2 > q) {T2 ← T2 − q;}
}
```