Machine-Level Software Optimization of Cryptographic Protocols

by

Dieter Fishbein

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This work explores two methods for practical cryptography on mobile devices. The first method is a quantum-resistant key-exchange protocol proposed by Jao et al.. As the use of mobile devices increases, the deployment of practical cryptographic protocols designed for use on these devices is of increasing importance. Furthermore, we are faced with the possible development of a large-scale quantum computer in the near future and must take steps to prepare for this possibility. We describe the key-exchange protocol of Jao et al. and discuss their original implementation. We then describe our modifications to their scheme that make it suitable for use in mobile devices. Our code is between 18–26% faster (depending on the security level). The second is an highly optimized implementation of Miller's algorithm that efficiently computes the Optimal Ate pairing over Barreto-Naehrig curves proposed by Grewal et al.. We give an introduction to cryptographic pairings and describe the Tate pairing and its variants. We then proceed to describe Grewal et al.'s implementation of Miller's algorithm, along with their optimizations. We describe our use of hand-optimized assembly code to increase the performance of their implementation. For the Optimal Ate pairing over the BN-446 curve, our code is between 7–8% faster depending on whether the pairing uses affine or projective coordinates.

Acknowledgements

First, I wish to thank my supervisor, David Jao, for his invaluable guidance throughout my masters program. Thank you also to the authors of [22,26] for providing me with figures for use in this thesis. Thank you to my readers Alfred Menezes and Edlyn Teske for their constructive feedback. Finally, I wish to thank my parents for their constant support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

It is expected that the use of mobile devices, such as smartphones and tablets, will become further widespread in the coming years. As their use increases, more people are using these devices for increasingly sensitive applications such as corporate email, online banking and for the storage of confidential information. As such, the deployment of practical cryptographic protocols for use on mobile devices is of the utmost importance.

The goal of this work is to explore two methods for practical cryptography on mobile devices. The first method is a quantum-resistant key-exchange protocol proposed by Jao et al. [17, 26]. Since we are faced with the possible development of a large-scale quantum computer in the near future, it is prudent for us to focus our efforts on the deployment of classical protocols that are resistant to attacks from such technology. The second is a highly optimized implementation of Miller's algorithm that efficiently computes the Optimal Ate pairing proposed by Grewal et al. [21, 22].

De Feo, Jao, and Plût [17,26] have proposed a Diffie-Hellman type key-exchange scheme based on computing isogenies between supersingular elliptic curves. The proposed scheme is believed to be quantum-resistant, and the fastest known attacks are exponential time. In this work, we present a practical implementation of the key-exchange protocol suitable for use in mobile (and non-mobile) devices. Our implementation is primarily written in C with hand-optimized assembly designed for use with either ARMv7 or x86-64 processors. It uses precomputed public parameters, with all the time-consuming computations offloaded from the device. Compared to the original implementation of [17], our code is between 18–26% faster (depending on the security level), and on iOS and Android devices we measured running times around 0.5–1 second for a round of key exchange at the (quantum) 80-bit security level.

The development of methods to efficiently compute cryptographic pairings is also of great importance. In the past decade, researchers have found a range of applications for the use of pairings in cryptography such as in key establishment and short signatures. These protocols require the efficient computation of the pairing in use. Grewal et al. [21, 22] have proposed a highly optimized implementation of Miller's algorithm to compute the Optimal Ate pairing over Barreto-Naehrig curves, designed for use on the ARMv7 processor. Their implementation is written in C and assembly language and, at the time of publication, is over three times faster that previously reported pairing implementations on ARM processors. Using similar techniques we used to increase the performance of the key-exchange, we introduced hand-optimized assembly code to increase the performance of their implementation. For the Optimal Ate pairing over the BN-446 curve, our code is between 7–8% faster depending on whether the pairing uses affine or projective coordinates.

The remainder of this work is organized as follows. In Chapter 2 we introduce the key-exchange protocol of Jao et al. and discuss their original implementation. In Chapter 3, we introduce cryptographic pairings and discuss Grewal et al.'s implementation of Miller's algorithm, along with their optimizations. In Chapter 4, we give a brief overview of assembly language and discuss our optimizations to both the key-exchange protocol and pairing computation. In Chapter 5 we present our conclusion and some opportunities for future research.

Chapter 2

A Quantum-Resistant Key-Exchange Protocol

2.1 Introduction

De Feo, Jao, and Plût [17, 26] have proposed a Diffie-Hellman type key-exchange scheme based on computing isogenies between supersingular elliptic curves. In this section we present their scheme. Following [17,26] , we discuss elliptic curves, their isogenies, the main key exchange protocol, low-level algorithmic details and specifics on the implementation.

2.2 Elliptic Curves & Isogenies

We define an *elliptic curve* over a field K as a projective nonsingular genus-1 algebraic curve E over K together with a distinguished base point ∞ of E defined over K. When the characteristic of K does not equal 2 or 3, which is always the case in this work, one can write E in the form

$$E : y^2 = x^3 + ax + b.$$

Points on an elliptic curve form a group with an efficiently computable group law, with identity element ∞. An elliptic curve E is determined up to isomorphism by its *j-invariant*, defined by

$$j(E) = 1728\frac{4a^3}{4a^3 + 27b^2}.$$

For any positive integer n, the *n-torsion group* $E[n]$ is defined to be the set of all points P in E defined over the algebraic closure $\overline{K}$ of K such that n times P is the identity:

$$E[n] = \{P \in E(\overline{K}) : nP = \infty\}.$$

As a group, $E[n]$ has $\mathbb{Z}$-rank equal to 2 provided that the characteristic of K does not divide n, and thus when viewed as a module over $\mathbb{Z}/n\mathbb{Z}$ it admits a basis of two elements.

An isogeny

$$\phi\colon E \to E'$$

is defined to be an algebraic map satisfying the property that ϕ is a group homomorphism. The *degree* of ϕ, denoted $\deg \phi$, is its degree as an algebraic map. An isogeny is *separable* if it is separable as an algebraic map.

We are interested in separable isogenies defined over finite fields. Assume E and E' are elliptic curves defined over a finite field $\mathbb{F}_q$. In this case, isogenies are determined up to isomorphism by their kernels. Any finite subgroup H of E induces an isogeny $E \to E/H$; conversely, for any isogeny ϕ, the group $\ker \phi$ is a finite subgroup of E. Finite subgroups of E in turn can be specified by identifying a set of generators. Given such a set of generators, the corresponding isogeny can be computed by using Vélu's formulas [41]. Additionally, every isogeny of degree greater than 1 can be factored into a composition of isogenies of prime degree over $\mathbb{F}_q$ [16].

Two curves E and E' are said to be isogenous over $\mathbb{F}_q$ if there exists an isogeny $\phi : E \to E'$ defined over $\mathbb{F}_q$. A theorem of Tate states that E and E' are isogenous over $\mathbb{F}_q$ if and only if the number points on both curves are the same [40]. Let ϕ have degree ℓ. Then ϕ has a dual isogeny $\hat{\phi}$ [39] such that $\phi \circ \hat{\phi} = [\ell]$. The property of being isogenous over $\mathbb{F}_q$ is an equivalence relation on the set of $\mathbb{F}_q$-isomorphism classes of elliptic curves defined over $\mathbb{F}_q$. Thus, we define an isogeny class to be an equivalence class under this equivalence relation.

The key-exchange scheme uses isogenies between *supersingular* elliptic curves. An elliptic curve is supersingular if its endomorphism ring (defined as the ring of all isogenies from a curve to itself, under the operations of pointwise addition and functional composition) has $\mathbb{Z}$-rank equal to 4. An elliptic curve is *ordinary* if its endomorphishm ring does not have $\mathbb{Z}$-rank equal to 4 (in this case its endomorphism risk will have $\mathbb{Z}$-rank equal to 1 or 2). Curves in the same isogeny class are either all supersingular or all ordinary.

2.3 Key Exchange Protocol

Fix a prime p of the form $\ell_A^a \ell_B^b \cdot f \pm 1$ where ℓ_A and ℓ_B are small primes, a and b are positive integers, and f is some (typically very small) cofactor. Let E be a supersingular elliptic curve defined over $\mathbb{F}_q = \mathbb{F}_{p^2}$. Fix a basis $\{P_A, Q_A\}$ of $E[\ell_A^a]$ over $\mathbb{Z}/\ell_A^a\mathbb{Z}$ and a basis $\{P_B, Q_B\}$ of $E[\ell_B^b]$ over $\mathbb{Z}/\ell_B^b\mathbb{Z}$. All of these parameters are public.

The idea of this protocol is a variation *à la* Diffie-Hellman of the commutative diagram

$$\begin{array}{ccc} E & \xrightarrow{\phi} & E/\langle P \rangle \\ {\scriptstyle\psi}\downarrow & & \downarrow \\ E/\langle Q \rangle & \to & E/\langle P, Q \rangle \end{array} \tag{2.3.1}$$

where ϕ and ψ are random walks in the graphs of isogenies of degree ℓ_A and ℓ_B respectively. The security of the key exchange is based on the difficulty of finding a path connecting two specified vertices in a graph of supersingular isogenies. We refer the reader to [17, 26] for a detailed discussion on the security of the protocol.

The key exchange protocol proceeds as follows. Alice chooses two secret, random elements $m_A, n_A \in_R \mathbb{Z}/\ell_A^a\mathbb{Z}$, not both divisible by ℓ_A, and computes an isogeny $\phi_A \colon E \to E_A$ with kernel $K_A := \langle [m_A]P_A + [n_A]Q_A \rangle$. Alice computes the image $\{\phi_A(P_B), \phi_A(Q_B)\} \subset E_A$ of the basis $\{P_B, Q_B\}$ for $E[\ell_B^b]$ under her secret isogeny ϕ_A. She sends these points to Bob together with E_A. Similarly, Bob selects secret, random elements $m_B, n_B \in_R \mathbb{Z}/\ell_B^b\mathbb{Z}$, not both divisible by ℓ_B and computes an isogeny $\phi_B \colon E \to E_B$ having kernel $K_B := \langle [m_B]P_B + [n_B]Q_B \rangle$. Bob then computes $\{\phi_B(P_A), \phi_B(Q_A)\}$ and sends the values to Alice along with E_B. With this information, Alice computes an isogeny $\phi'_A \colon E_B \to E_{AB}$ having kernel equal to $\{[m_A]\phi_B(P_A), [n_A]\phi_B(Q_A)\}$. Bob proceeds *mutatis mutandis*. Alice and Bob can then use the common j-invariant of

$$E_{AB} = \phi'_B(\phi_A(E)) = \phi'_A(\phi_B(E)) = E/\{[m_A]P_A + [n_A]Q_A, [m_B]P_B + [n_B]Q_B\}$$

as their shared secret key. For further details, we refer the reader to [17, 26].

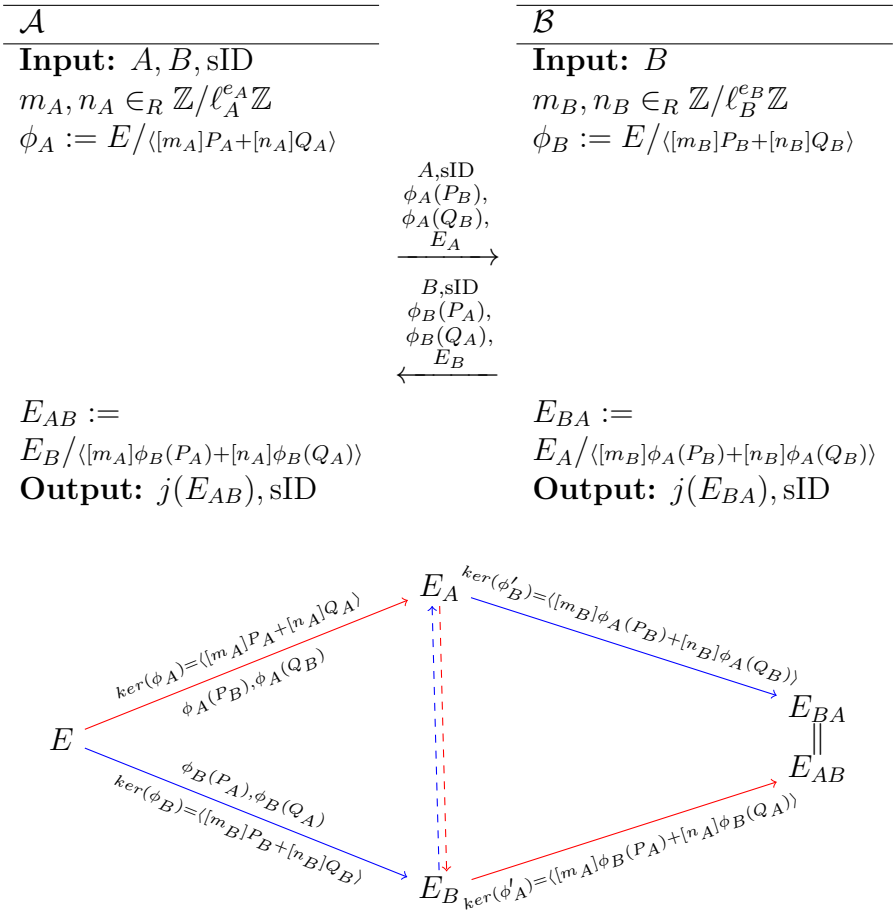

Figure 2.1: Key-exchange protocol using isogenies on supersingular curves.

2.4 Algorithmics

2.4.1 Parameter Generation

Jao et al.'s original implementation of the key exchange works for any value of ℓ_A and ℓ_B. When $\ell_A = 2$ and $\ell_B = 3$, the original implementation performs most of the key exchange protocol in C (as opposed to Cython) making it more efficient. However, even in this case, parts of the key-exchange protocol are done in Cython. Our modifications only concern

themselves with the case $\ell_A = 2$ and $\ell_B = 3$ and we assume those parameter values for the remainder of the chapter. For any fixed choice of a and b one can choose random values of f until one of $p = \ell_A^a \ell_B^b \cdot f + 1$ or $p = \ell_A^a \ell_B^b \cdot f - 1$ is prime. An effective version of the prime number theorem in arithmetic progressions by Lagarias and Odlyzko [29] guarantees that the density of such primes is sufficient.

Fixing a prime $p = \ell_A^a \ell_B^b \cdot f \pm 1$ we now need a supersingular curve E_0. A result by Broker [15] says that it is computationally easy to find a supersingular curve E over $\mathbb{F}_{p^2}$ with cardinality $(p \mp 1)^2 = (\ell_A^a \ell_B^b \cdot f)^2$. Once can either chose $E_0 = E$ or construct the isogeny graph consisting of all supersingular curves defined over $\mathbb{F}_{p^2}$ and choose E_0 via random walks on said isogeny graph. Using either method, we obtain E_0 with group structure $(\mathbb{Z}/(p \mp 1)\mathbb{Z})^2$. To obtain a basis for the torsion group $E_0[\ell_A^a]$, choose a random point $P \in_R E_0(\mathbb{F}_{p^2})$ and set $P' = (\ell_B^b \cdot f)^2 P$ so that P' has order dividing ℓ_A^a. One checks whether P' has order exactly equal to ℓ_A^a by multiplying P' by powers of ℓ_A. If this check succeeds (which it will with high probability) then we set $P_A = P'$. We choose a second point of order ℓ_A^a, Q_A, in the same way. One must check that P_A and Q_A are independent and this is done by computing the Weil pairing $e(P_A, Q_A)$ in $E_0[\ell_A^a]$ and checking that the result has order ℓ_A^a via repeated multiplications of ℓ_A. If this fails we can simply chose another point Q_A and try again.

2.4.2 Key Exchange

The key exchange is performed in two rounds and in each round Alice and Bob proceed as follows:

1. Compute $\langle R \rangle = \langle [m]P + [n]Q \rangle$ for points P, Q;
2. Compute the isogeny $\phi : E \to E/\langle R \rangle$ for a supersingular curve E;
3. In only the first round, compute $\phi(R)$ and $\phi(S)$ for some points R, S;

where E, P, Q, R and S depend on both the round and the player. We now discuss how to implement these three steps.

There are many classical techniques for computing $\langle [m]P + [n]Q \rangle$. One first observes that we need only a single generator. We can compute $[m]P + [n]Q$ naively by repeatedly adding copies of P and Q together. Note that at least one of m or n will be invertible modulo the order of the group and so without loss of generality we can assume m is invertible. In this case, $R' = P + [m^{-1}n]Q$ is another generator. Though computing R' via

Algorithm 2.1 Three-point ladder to compute $P + [t]Q$ [26].

Require: t, P, Q;
Set $A = 0, B = Q, C = P$;
Compute $Q - P$;
for i decreasing **from** $|t|$ **to** 1 **do**
  Let t_i be the i-th bit of t;
  if $t_i = 0$ **then**
    $B = \text{dadd}(A, B, Q), \quad C = \text{dadd}(A, C, P), \quad A = 2A;$
  else
    $A = \text{dadd}(A, B, Q), \quad C = \text{dadd}(B, C, Q - P), \quad B = 2B$
  end if
end for
Ensure: $C = P + [t]Q$.

a standard double and add approach is much more efficient that computing $[m]P + [n]Q$ naively, it is vulnerable to simple power analysis (SPA) [28]. The choice of model for elliptic curves in this implementation is the Montgomery curve (see Section 2.4.4). In [35], Jao et al. proposed Algorithm 2.1 which both efficiently computes R' on Montgomery curves and is not vulnerable to SPA. At each iteration of the algorithm, registers A, B and C contain values $[x]Q$, $[x + 1]Q$ and $P + [x]Q$ respectively, where x is equal to a certain number of the leftmost bits of $m^{-1}n$. When the algorithm terminates, C contains the required result. The function dadd(A,B,C) is *differential addition*. It computes the sum $A + B$ knowing $C = A - B$. In Montgomery coordinates there is no distinction between the affine points (x, y) and $(x, -y)$. Thus, Q and $-Q$ have the same coordinates. For example, in the first iteration of Algorithm 2.1, $A - B = 0 - Q = -Q = Q$, as required. Montgomery gave explicit formulas for differential addition on certain curves (now called Montgomery curves). Such curves have very efficient differential addition so as to make the ladder in Algorithm 2.1 nearly as efficient as a naive double and add approach, while also being resistant to SPA attacks. The only (potentially) useful piece of information leaked by Algorithm 2.1 is the size of t in bits.

2.4.3 Computing Isogenies

Computing the isogenies in the protocol can be accomplished via an iterative process. Given an elliptic curve E and a point R of order ℓ^e, we compute $\phi\colon E \to E/\langle R\rangle$ by decomposing ϕ into a chain of degree ℓ isogenies, $\phi = \phi_{e-1} \circ \cdots \circ \phi_0$, as follows. Set $E_0 = E$

and $R_0 = R$, and define

$$E_{i+1} = E_i/\langle \ell^{e-i-1} R_i \rangle, \qquad \phi_i \colon E_i \to E_{i+1}, \qquad R_{i+1} = \phi_i(R_i).$$

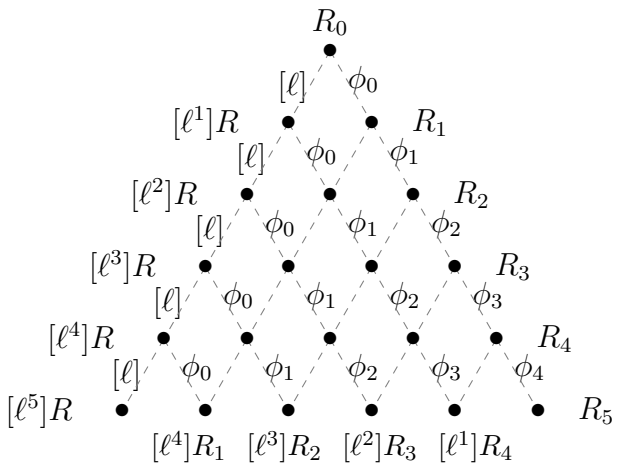

Figure 2.2: Diagram of isogeny computation.

Figure 2.2 shows the computational structure of computing isogenies for $c = 6$. The bold dots represent points on E. Points on the same left diagonal belong to the same curve and points of the same height on the diagram represent points of the same order. Leftward dashed edges refer to multiplication by ℓ, while rightward dashed edges refer to evaluation of isogenies of degree ℓ. At the beginning of the algorithm, only R_0 is known. In order to compute ϕ, we must compute all the elements at the bottom row of Figure 2.2. Using $[\ell^{e-i-1}]R_i$ we can compute the kernel of ϕ_i via $O(\ell)$ point additions. We can then apply Vélu's formulas to compute ϕ_i and E_{i+1}. Since evaluating degree ℓ isogenies is generally twice as expensive as multiplications by ℓ, determining the best approach is a non-trivial combinatorial problem.

We will now formalize the picture in Figure 2.2 and then discuss an algorithm to optimally compute ϕ.

Definition 2.1. *Let T_n be the portion of the unit triangular equilateral lattice contained between the x-axis, the line $y = \sqrt{3}x$ and the line $y = -\sqrt{3}(x - n + 1)$. T_n is called the* discrete equilateral triangle *(DET) of side n.*

An *edge* is a segment of unit length directed towards the x-axis connecting two points in T_n. A *left edge* is an edge with positive slope. It is called a *right edge* otherwise. Directing the edges as such imparts a directed acyclic graph structure on T_n. We equip the points of T_n with the ordering $\to$ defined by $x \to y$ if and only if there exists a path in T_n from x to y. The *leaves* and *root* of T_n are the final and initial point(s) respectively. For any two points y, y' of T_n, there is at most one point x such that $x \to y$ and $x \to y'$. We write $x = y \wedge y'$. A *strategy* S is a sub-graph of T_n having a unique root. We call a strategy *full* if it contains all the leaves of T_n. In this case we must have that the root of S is the same as T_n.

One should first note that any full strategy yields a valid algorithm to compute the isogeny ϕ. One travels the graph in depth-first left-first oder. Each time the bottom (x-axis) is hit, one apply's Vélu's formulas before proceeding.

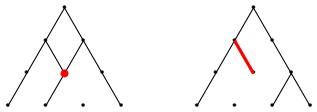

Figure 2.3: Two ill-formed strategies.

One should note that there are certain types of strategies which cannot be optimal. A strategy with more than one edge passing through a point is not optimal as one of those edges is clearly useless. Furthermore, a strategy that has a leaf different from the leaves of T_n is not optimal as that particular leaf will be immaterial to the isogeny computation. We define a *well-formed* strategy to be one that has neither of the preceding flaws. We call a strategy that is not well-formed, *ill-formed*.

Figure 2.4: The seven well-formed full strategies for $n = 4$. Notice that the three middle strategies share the same binary tree topology and the middle one is the canonical strategy.

We are interested in computing the "optimal" full strategy, according to some measure of computational effort. We first note that any well-formed strategy has a particular binary tree topology obtained by discarding the internal nodes of out-degree less than 2

and preserving the same connectivity structure. Conversely, let A be a binary tree with n leaves and such that no internal nodes have out-degree less than 2. We can canonically associate a strategy S on A as follows. If $n = 1$ then $S = T_1$. Otherwise, let S' be the strategy associate to the left branch of A, S'' the strategy associated to the right branch of A translated to the right by $|S'|$. Let r' and r'' be the roots of S' and S'' respectively. Then $r' \wedge r''$ is the root of T_n and our canonical strategy is defined as $S = rr' \cup rr'' \cup S' \cup S''$ where xx' denotes the edge between x and x'. This allows us to recursively build a strategy from an appropriate binary tree. Intuitively, one can view S as being visually the same as A.

Recall that in a strategy, left edges are multiplications by ℓ and right edges are ℓ-isogeny evaluations. The cost of each of these operations is dependent on the specific hardware and software platform used in implementation. Thus, in striving to achieve an optimal strategy, it makes sense for us to assign different weights to left and right edges.

Let (p, q) be a pair of positive real numbers where p represents the weight of a left edge and q represents the weight of a right edge. We call such a (p, q) a *measure*. For a set of edges S of T_n let (S) denote the sum of the weights of edges in S.

Jao et al. show that for any measure, the canonical strategy is minimal with respect to all other strategies sharing the same tree topology. We call a strategy *optimal* when it is minimal among all full strategies with n leaves with a respect to a given measure (p, q). Thus, for a given measure, the optimal strategy would, in theory, give the least computationally intensive way to compute a degree n isogeny.

Lemma 2.1 ([26], Lemma 4.5)**.** *Let S be an optimal strategy. Let S' and S'' be, respectively, its left and right branch. Then S' and S'' are optimal strategies.*

Let $C_{p,q}(n)$ be the cost of the optimal strategies with n leaves. Lemma 2.1 tells us

$$C_{p,q}(n) = \min_{i=1,\ldots,n-1} \big(C_{p,q}(i) + C_{p,q}(n-i) \; + \; (n-i)p + iq\big). \tag{2.4.1}$$

Intuitively, the cost of the optimal strategy S with n leaves is equal to the total cost of the optimal strategies on its left and right branch plus the cost of the the edges leading to those branches. This equality suggests a dynamic programming algorithm that for a given measure (p, q) computes $C(n)$ in $O(n^2)$ time. Additionally, determining the cost $C_{p,q}(i)$ for $i = 1, \ldots n$ will give us the optimal strategy for T_n. For example, assume we wish to determine the optimal strategy for T_3 under a given measure (p, q). One first determines $C_{p,q}(2)$ which is always equal to $p + q$ (we can assume $C_{p,q}(1) = 0$). The optimal strategy on two leaves appears in Figure 2.5.

Figure 2.5: Optimal strategy for two leaves.

Then one determines $C_{p,q}(3)$ which equals either $C_{p,q}(2) + 2p + q$ or $C_{p,q}(2) + p + 2q$. Determining $C_{p,q}(3)$ will tell us which additional edges should be added. $C_{p,q}(3)$ would be one of the two strategies in Figure 2.6. Thus by determining which of $(C_{p,q}(2) + 2p + q, C_{p,q}(2) + p + 2q)$ is minimal we determine the optimal strategy on T_3.

Figure 2.6: The two possible optimal strategies for three leaves superimposed on the optimal strategy for two leaves.

Since the optimal strategy can be computed off the device and loaded onto the device as a precomputed parameter, we need not be very concerned with efficiency of the algorithm.[1] A straightforward Python implementation computes the optimal strategies for $n = 1024$ in under one second [17, 26].

2.4.4 Choice of Models

All curves we are concerned with have group structure $(\mathbb{Z}/(p \mp 1)\mathbb{Z})^2$. This forces either the curve itself or its twist to have a point of order 4. In [10], Bernstein et al. prove that any elliptic curve having a point of order 4 is isomorphic to a twisted Edwards curve. They further prove that any twisted Edwards curve is isomorphic to a Montgomery curve. Since, over a quadratic extension field, an elliptic curve is isomorphic to its quadratic twist, we see that all curves we are concerned with will be isomorphic to a Montgomery curve.

To estimate efficiency we count the number of elementary operations in $\mathbb{F}_{p^2}$. We write I, M, S for the costs of one inversion, multiplication and squaring respectively. We make the

[1] Optimal strategies can be mathematically characterized, though the dynamic programming algorithm is satisfactory from an implementation perspective. See [17, 26].

assumption that $S \leq M \leq I$. We ignore additions, subtractions and comparisons as they are significantly faster than the operations we include in our estimate. We use the Explicit Formulas Database (EFD) [11] for operation counts on elliptic curves. However, unlike the EFD, we count multiplications by constants (other than small integers) as ordinary multiplications.

Montgomery curves [35] have equation

$$M_{B,A} : By^2 = x^3 + Ax^2 + x. \tag{2.4.2}$$

One can represent points on $M_{B,A}$ by $(X : Z)$ where $x = X/Z$. This is known as a *Kummer* representation. One refers to performing operations on a curve with such a representation as performing operations on the curve's *Kummer line*. Such a representation has the disadvantage of identifying P with $-P$ since the negative of a point only differs in the Y coordinate. However, Montgomery curves have very efficient arithmetic on their Kummer line. A point can be doubled using $3M + 2S$ or $2M + 2S$ when it is scaled to have Z-coordinate equal to 1. Since P is identified with $-P$, it is not possible to add two distinct points. However, if $P - Q$ is known, one can compute $P + Q$ via differential addition. One differential addition has a cost of $4M + 2S$ or $3M + 2S$ when $P - Q$ is scaled to have Z coordinate equal to 1. Through the use of a Montgomery ladder, along with doublings and differential additions, one can compute any scalar multiple of a point [35]. Since P and $-P$ generate the same subgroup of a group of points on an elliptic curve, and an isogeny can be uniquely identified with such a subgroup, we see that isogenies can be defined and evaluated correctly on the Kummer line.

Computing $\langle [m]P + [n]Q \rangle$

As discussed in Section 2.4.2, in order to compute $[m]P + [n]Q$, we will actually compute $P + [m^{-1}n]Q$ via Algorithm 2.1. We first compute $P - Q$ in projective coordinates by using the standard chord and tangent law. We then scale P, Q and $P - Q$ to have Z-coordinate equal to 1 and work on the Kummer line. At each iteration of the for loop in Algorithm 2.1 we perform one doubling and two differential additions at a total cost of $9M + 6S$ per iteration.

Isogenies of Montgomery Curves

In [17,26], Jao et al. give explicit formulas for isogenies of Montgomery curves and optimize the degree 2 and 3 case. We will not derive those formulas here but instead refer the reader

to the original paper for those details. Let E be the Montgomery curve defined in equation 2.4.2. It has a point of order two $P_2 = (0, 0)$ and a point of order four $P_4 = (1, \sqrt{(A+2)/B})$ such that $[2]P_4 = P_2$. Let X be the abscissa of P. We now describe the formulas for degree 2, 3 and 4 isogenies that are used in the software implementation. Isogenies of degree 2 are defined by

$$\begin{aligned} &\phi_2 : E \to F_2, \\ &(x, y) \mapsto \left(\frac{(x-1)^2}{x}, y\left(1 - \frac{1}{x^2}\right)\right), \\ &\text{and } F_2 \ : \ \frac{B}{2\sqrt{2+A}}y^2 = x^3 + \frac{A+6}{2\sqrt{2+A}}x^2 + x. \end{aligned} \tag{2.4.3}$$

Isogenies of degree 3 are defined by

$$\begin{aligned} &\phi_3 : E \to F_3, \\ &(x, y) \mapsto \left(\frac{x\left(x-\frac{1}{X}\right)^2}{(x-X)^2}X^2,\ y\frac{\left(x-\frac{1}{X}\right)\left(\left(x-\frac{1}{X}\right)(x-X)+2x\left(\frac{1}{X}-X\right)\right)}{(x-X)^3}X^2\right), \\ &\text{and } F_3 \ : \ BX^2y^2 = x^3 + \left(A + \frac{6}{X} - 6X\right)X^2x^2 + x. \end{aligned} \tag{2.4.4}$$

If we define

$$\begin{aligned} &\psi : F_2 \to G, \\ &G \ : \ \frac{B}{2-A}y^2 = x^3 - 2\frac{A+6}{2-A}x^2 + x, \\ &\text{and } (x, y) \mapsto \left(\frac{1}{2-A}\frac{(x+4)(x+(A+2))}{x}, \frac{y}{2-A}\left(1 - \frac{4(2+A)}{x^2}\right)\right). \end{aligned} \tag{2.4.5}$$

Then $\phi_4 := \psi \circ \phi$ is an isogeny of degree 4.

Jao et al.'s original implementation of the key-exchange is suitable when ℓ_A and ℓ_B are arbitrary small primes. However, our implementation is only suitable for $\ell_A = 2$ and $\ell_B = 3$. As in Section 2.4.3, isogenies of composite smooth degree are computed by composing isogenies of prime degree. In the degree 3^e case, this is done by simply composing degree 3 isogenies. Let $e \geq 3$. In the degree 2^e case, one needs knowledge of a point of order 8 in order to determine the curve F_2 [17,26]. Thus, one cannot simply chain together multiple isogenies of degree 2. This problem is solved by chaining together $e-2$ isogenies of degree 2 followed by an isogeny of degree 4. In [17,26], Jao et al. suggested that there may be a small speed advantage in using chains of degree 4-isogenies instead. From

Table 2.1: Comparative costs for multiplication and isogeny evaluation in projective Kummer coordinates, in number of multiplications and squarings, and assuming $S = 0.8M$.

ℓ	2	3	4
Isogeny	$2M + S$	$4M + 2S$	$6M + S$
	2.8	5.6	6.8
Multiplication	$3M + 2S$	$7M + 4S$	$6M + 4S$
	4.6	10.2	9.2

Table 2.2: Comparative costs of the optimal strategy for computing a degree 2^{514} ($\ell = 2, 4$) or 3^{323} ($\ell = 3$) isogeny, assuming $S = 0.8M$.

	optimal strategy		
ℓ	2	3	4
Isogenies	2741	1610	1166
Multiplications	1995	1151	921
Total cost	16852	20756	16402

Table 2.1, this certainly appears plausible. However, after further investigation, we found that using 4-isogenies seems to be slightly less efficient in practice ($< 1\%$ disadvantage compared to chains of 2-isogenies).

Table 2.1 compares the cost of isogeny evaluations and scalar multiplications for isogenies of degree 2, 3 and 4. We also report the cost obtain by setting $S = 0.8M$. This figure is roughly based on the fact that squaring in $\mathbb{F}_{p^2}$ requires 2 multiplications (as oppose to 3 for field multiplication). These figures assume certain expressions have been precomputed, and common subexpressions shared.

Table 2.2 lists the total cost of isogeny evaluations and scalar multiplications for an optimal strategy.

2.5 Implementation

The original implementation from [17, 26] uses a mixed C/Cython/Python/Sage architecture. Parameter generation is done in Sage, and the computation of the optimal strategy for computing isogenies is done in Python using the dynamic programming algorithm discussed in Section 2.4.3. Arithmetic in $\mathbb{F}_{p^2}$ is written using C, using GMP to support arithmetic modulo p. Elliptic curve arithmetic is implemented in Cython. In the special

case $\ell_A = 2$ and $\ell_B = 3$, the key exchange uses a combination of C and Cython, with the most critical parts done in C. The fact that elliptic curves are implemented using Cython prevents a pure C implementation. For all other values of ℓ_A and ℓ_B, the key exchange is done in Cython.

See Section 4.2 for a description of our contributions and results pertaining to isogeny-based key-exchange.

Chapter 3

An Efficient Method of Pairing Computation

3.1 Introduction to Cryptographic Pairings

The use of pairings in cryptography was initially proposed by Joux for use in the first three-party, one-round key agreement protocol that was secure against eavesdroppers [27]. This protocol was surprisingly simple and solved a problem that had been open for some time. This sparked researchers to begin investigating the use of pairings in other cryptographic contexts. Their use was further popularized by Boneh and Franklin as a result of their work on identity based encryption. Now, we see pairings in a variety of additional areas such as short signature schemes and attribute based encryption. Implementing these protocols successfully requires the efficient computation of the pairings involved. Much work has been done on the efficient computation of pairings on PCs [3,12,20,36]. However, relatively little work of this nature has been done on ARM-based platforms [2, 22, 25]. In this section we introduce pairings and discuss how to efficiently compute them. We then focus on recent work of Grewal et al. [22] which presents an implementation that efficiently computes the Optimal Ate pairing over Barreto-Naehrig curves.

3.1.1 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 3.1. *A* bilinear pairing e *is a function*

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

that satisfies the following additional properties:

*(1) (*bilinearity*) For all* P, $P' \in G_1$ *and* Q, $Q' \in G_2$,

$$e(P + P', Q) = e(P, Q)e(P', Q),$$
$$e(P, Q + Q') = e(P, Q)e(P, Q').$$

*(2) (*non-degeneracy*)* $e(P, P) \neq 1$.

Additionally, one wants e to be efficiently computable. From the two properties above, one can derive 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 $P = \infty$.

3.1.2 Applications

Tripartite Diffie-Hellman

The first use of pairings in cryptography was the development of the three-party, one-round key-exchange protocol by Joux in 2000 [27]. This protocol is very much a three-party version of Diffie-Hellman and its security is based on the following assumption:

Definition 3.2 (Bilinear Diffie-Hellman Assumption)**.** *Let* e *be a bilinear pairing* $e : G_1 \times G_1 \to G_2$ *with* $P \in G_1$*. Given* P, aP, bP, cP, *it is computationally infeasible to compute* $e(P, P)^{abc}$.

The protocol works as follows. Let e be a bilinear pairing $e : G_1 \times G_1 \to G_2$.

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 pairing (more precisely property (3) above) allows each party to compute a common secret key. Indeed,

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

BLS Short Signatures

An additional use of pairings is in signature schemes. Most discrete logarithm signature schemes are variants of the ElGamal scheme. In such schemes, signatures are usually comprised of a pair of integers modulo the order (n) of the underlying group (G). By using pairings, Boneh, Lynn and Shacham were able to develop a signature scheme that uses one group element as the signature and where group elements can be represented using roughly the same number of bits as an integer modulo n [14].

Let e be a bilinear pairing $e : G_1 \times G_1 \to G_2$ where $G_1 = \langle P \rangle$ has order n and H be a cryptographic hash function $H : \{0,1\}^* \to G_1 \setminus \{\infty\}$. The Boneh, Lynn, Shacham (BLS) short signature scheme is as follows:

Private Key: Alice randomly selects an integer $a \in_R [1, n-1]$.

Public Key: $A := aP$.

Signature: Let $m \in \{0,1\}^*$ be the message. Then $S = aH(m)$ is the signature.

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

If the signature is valid then we would have

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

as required for verification.

3.1.3 Divisors

All bilinear pairings that we know about are defined on elliptic curves. Furthermore, they all use the concepts of *divisors*. In this section, we give an overview of theory of divisors required for their study. This section is an adaptation of information found in [32]. Let E be an elliptic curve defined over a field K.

Definition 3.3. *A* divisor D *is a formal sum of points in* E*.*

$$D = \sum_{P \in E} n_P P,$$

where $n_P \in \mathbb{Z}$*. Furthermore, we only allow a finite number of the* n_P *to be non-zero.*

The *degree of* D, denoted $\deg(D)$, is defined to be $\deg(D) = \sum_{P \in E} n_P$. We note that $\deg(D)$ is an integer. The *order of* D *at* P, denoted $\mathrm{ord}_P(D)$, is defined to be $\mathrm{ord}_P(D) = n_P$.

We denote the set of divisors on E as $\mathbb{D}$. Note that $\mathbb{D}$ forms an abelian group under a natural addition rule,

$$\sum_{P \in E} n_P P + \sum_{P \in E} m_P P = \sum_{P \in E} (n_P + m_P) P.$$

Since only a finite number of n_P and m_P are non-zero, we see that only a finite number of $(n_P + m_P)$ will be non-zero. Hence, $\sum_{P \in E}(n_P + m_P)P$ satisfies the definition of a divisor.

Let $\overline{K}$ denote the algebraic closure of K. Let E be an elliptic curve and let $\overline{K}(E)$ denote the field of rational functions on E with coefficients in $\overline{K}$. Let $\overline{K}[E]$ denote the field of polynomial functions with coefficients in $\overline{K}$. Let R be a non-zero rational function in $\overline{K}(E)$. We wish to define the *order* of R at a point P denoted $(\mathrm{ord}_P R)$.

Definition 3.4. *Let* R *be a non-zero element of* $\overline{K}(E)$ *and let* $P \in E$*. If* $R(P) = 0$*, then* R *is said to have a* zero *at* P*. If* R *is not defined at* P *then* R *is said to have a* pole *at* P*.*

The zeros and poles of the rational functions on E roughly correspond to zeros and poles of functions in $\mathbb{C}(x)$, the field of rational functions over the complex numbers.

Theorem 3.1 ([32], Theorem 23)**.** *Let* $P \in E$*. Then there exists a rational function* $U \in \overline{K}(E)$ *with* $U(P) = 0$ *such that for each non-zero polynomial function* R *in* $\overline{K}[E]$*, there exists an integer* d *and function* S *in* $\overline{K}(E)$ *such that* $S(P) \neq 0, \infty$ *and* $R = U^d S$*. Also,* d *is only dependent on* R *and* P*. The function* U *is then called a* uniformizing parameter *for* P*.*

The existence of a uniformizing parameter is proved by explicitly constructing a uniformizing parameter for several cases depending on the characteristics of the point P.

Definition 3.5. *Let R be a non-zero polynomial in $\overline{K}[E]$ and $P \in E$. Let U be a uniformizing parameter for P. We can write $R = U^d S$ where S is a rational function on E with $S(P) \neq 0, \infty$. We define the* order of R at P *to be d, denoted* $\operatorname{ord}_P(R) = d$.

Consider $f(x) \in \mathbb{C}[x]$. In this setting, the order of f at a point $y \in \mathbb{C}$ is defined as the multiplicity of the zero at $f(y)$. If $f(y) \neq 0$, the order of f at y would be zero. If $f(y) = 0$, the we can rewrite f as $f(x) = (x-y)^d g(x)$ where $g(x)$ is a polynomial function in x and $g(y) \neq 0$. Then the order of f at y would be equal to d. Again, we see the connection between functions in $\overline{K}[E]$ and polynomials defined over $\mathbb{C}[x]$.

Now, consider $f(x) \in \mathbb{C}[x]$, a non-zero polynomial function of degree n. Since $\mathbb{C}$ is algebraically closed, we see that $f(x)$ splits completely into linear factors in $\mathbb{C}[x]$. Now, if we look at the order of $f(x)$ evaluated at each point in $\mathbb{C}$, we get, $\sum_{y \in \mathbb{C}} \operatorname{ord}_y f(x) = n$. If we consider a non-zero polynomial function $R \in \overline{K}[E]$, the only difference with the preceding analogy is that we must consider the point at infinity, ∞. One can show that $\operatorname{ord}_\infty R = -\deg(R)$. So intuitively it seems reasonable that $\sum_{P \in E} \operatorname{ord}_P(R) = 0$. This is in fact true, and also holds where R is a non-zero rational function in $\overline{K}(E)$. Furthermore, one can also show that the number of zeros and poles for a function in $\overline{K}(E)$ is finite.

Theorem 3.2 ([32], Theorem 29)**.** *Let R be a non-zero rational function in $\overline{K}(E)$. Then, R has a finite number of zeros and poles. Furthermore, $\sum_{P \in E} \operatorname{ord}_P(R) = 0$.*

Now, consider $\sum_{P \in E} (\operatorname{ord}_P R)P$ where R is a non-zero rational function in $\overline{K}(E)$. Proving that R has a finite number of zeros and poles is equivalent to proving that R has a finite number of points, P, at which $\operatorname{ord}_P R \neq 0$. Since we have $\sum_{P \in E} \operatorname{ord}_P(R) = 0$, we see that $\sum_{P \in E} (\operatorname{ord}_P R)P$ satisfies the definition of a divisor. In particular, it is a degree zero divisor.

Definition 3.6. *Let R be a non-zero rational function in $\overline{K}(E)$. We define the* divisor of R, *denoted* $\operatorname{div}(R)$, *as*

$$\operatorname{div}(R) = \sum_{P \in E} (\operatorname{ord}_P R)P.$$

Let $\mathbb{D}^0$ denote the set of divisors of degree zero. Note that $\mathbb{D}^0$ is a subgroup of $\mathbb{D}$. Let $D \in \mathbb{D}$. We call D a *principal divisor* if $D = \operatorname{div}(R)$ for some non-zero rational function

$R \in \overline{K}(E)$. We denote the set of all principal divisors as $\mathbb{P}$. Let R_1, $R_2 \in \mathbb{P}$. It is fairly clear that $\mathbb{P}$ is a subgroup of $\mathbb{D}^0$ under addition. The following computation verifies this:

$$\begin{aligned}\operatorname{div}(R_1) + \operatorname{div}(R_2) &= \sum_{P \in E} \operatorname{ord}_P(R_1)P + \sum_{P \in E} \operatorname{ord}_P(R_2)P \\ &= \sum_{P \in E} (\operatorname{ord}_P(R_1) + \operatorname{ord}_P(R_2))P \\ &= \sum_{P \in E} (\operatorname{ord}_P(R_1 R_2))P \\ &= \operatorname{div}(R_1 R_2).\end{aligned}$$

Intuitively, the above computation can be explained by realizing that the only points appearing in $\operatorname{div}(R_1)$ and $\operatorname{div}(R_2)$ are the zeros and poles of R_1 and R_2 respectively, along with the point at infinity. Since the product of the two functions would have the same zeros and poles, and also contain the point at infinity, we see that this computation makes sense.

We can now define

$$\mathbb{CL}^0 = \mathbb{D}^0/\mathbb{P}.$$

This is the quotient group of $\mathbb{D}^0$ modulo $\mathbb{P}$ and is called the *divisor class group* of E. Let D_1, $D_2 \in \mathbb{D}^0$. We call D_1 and D_2 *equivalent*, denoted $D_1 \backsim D_2$, if $D_1 - D_2 \in \mathbb{P}$.

We now state two important results regarding divisors on elliptic curves. The result in Theorem 3.4 is known as *Weil Reciprocity*.

Theorem 3.3 ([13], Chapter 9)**.** *Let E be an elliptic curve over a field K. Let*

$$D = \sum_{P \in E} n_P(P)$$

be a divisor of degree zero on E. Then $D \sim 0$ (i.e D is the divisor for some rational function $R \in \overline{K}(E)$) if and only if $\sum_{P \in E}[n_P]P = \infty$.

Let $R \in \overline{K}(E)$ and $D = \sum_{P \in E} n_P(P)$ be a divisor of degree zero such that $\operatorname{div}(R)$ and D have disjoint support. Let

$$R(D) := \prod_{P \in E} R(P)^{n_P}.$$

Theorem 3.4 ([13], Appendix)**.** *Let E be an elliptic curve over a field K and $R, Q \in \overline{K}(E)$ be rational functions on E. Suppose that R and Q have disjoint support. Then*

$$R(\operatorname{div}(Q)) = Q(\operatorname{div}(R)).$$

3.1.4 The Tate Pairing

The next two subsections follow Menezes' exposition in [31]. The Tate pairing is a bilinear pairing defined on the torsion subgroup of the group of points of an elliptic curve over a finite field. Let E be an elliptic curve over $\mathbb{F}_q$ and $\#E(\mathbb{F}_q) = hn$ where n is a prime such that $\gcd(n, q)=1$. Let k be minimal such that $n|q^k - 1$ and recall that $E[n]$ denotes the n-torsion group of E (c.f. Section 2.2). Let μ_n denote the order-n subgroup of $\mathbb{F}_{q^k}^*$ (μ_n is also the group of nth roots of unity). Balasubramanian and Koblitz show that $E[n] \subseteq E(\mathbb{F}_{q^k})$ [5]. Also assume that $\gcd(n, h)=1$ and $n \nmid \#E(\mathbb{F}_{q^k})/n^2$. The Tate pairing is defined as a map

$$e : E[n] \times E[n] \to \mu_n.$$

We now define the Tate pairing. Let $P, Q \in E[n]$. Let $f_P \in \mathbb{F}_q(E)$ be a function with $\mathrm{div}(f_P) = n(P) - n(\infty)$. This ensures f_P has a zero of order n at P and a pole of order n at ∞. Thus, f_P has no other zeros or poles. We have that $nP - n\infty = \infty$ since P is an n-torsion point. Therefore, the existence of f_P is guaranteed by Theorem 3.3. Let $R \in E[n]$ such that $R \notin \{\infty, P, -Q, P - Q\}$ and let $D_Q = (Q + R) - (R)$. The choice of R ensures D_Q and $\mathrm{div}(f_P)$ have disjoint support. The Tate pairing is then defined as

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

One can show that the Tate pairing is well defined and satisfies all the required properties to be a bilinear pairing [6].

If one normalizes f_P, then one can ignore D_Q and work only with Q. In this case, the Tate pairing becomes

$$e(P, Q) = f_P(Q)^{(q^k-1)/n}. \tag{3.1.1}$$

3.1.5 Miller's Algorithm

The difficult part about computing the Tate pairing is computing the function f_P. In general, any function $f_{i,P}$ (or simply f_i if the point P is implied) with divisor $i(P) - (iP) - (i-1)(\infty)$ is called a *Miller* function and in this subsection we describe an efficient algorithm for computing such functions [33]. The success of Miller's algorithm is based on the following observation.

Theorem 3.5 ([21], Theorem 1.1.21)**.** *Let $P \in E[n]$ and let i and j be positive integers. Let l be the line through iP and jP, and let v be the vertical line though $iP + jP$. Then*

$$f_{i+j} = f_i f_j \frac{l}{v} \tag{3.1.2}$$

Proof. Recall the group law for the group of points on an elliptic curve. One can see that l will also intersect the point $-[i+j]P$ and that v will intersect , $[i+j]P$, $-[i+j]P$ and ∞. Thus,

$$\text{div}\left(\frac{l}{v}\right) = (iP) + (jP) - ([i+j]P) - (\infty).$$

We have,

$$\begin{aligned}
\text{div}\left(f_i f_j \frac{l}{v}\right) &= \{i(P) - (iP) - (i-1)(\infty)\} + \{j(P) - (jP) - (j-1)(\infty)\} \\
&\quad + \{(iP) + (jP) - ([i+j]P) - (\infty)\} \\
&= (i+j)P - ([i+j]P) - (i+j-1)\infty \\
&= \text{div}(f_{i+j})
\end{aligned}$$

as required.

□

The basic idea of Miller's algorithm is to start with $f_1 = 1$ and use Theorem 3.5 above with a standard double-and-add approach to compute f_n for some given n. Algorithm 3.1 presents Miller's algorithm. We only require the value $f_n(Q)$ in order to compute the Tate pairing. Thus, Miller's algorithm only computes the value of the intermediate functions f_i at the point Q. Miller's algorithm requires $O(\log(n))$ iterations with each requiring a constant number of arithmetic operations in $\mathbb{F}_{q^k}$. We refer to steps 2 through 9 as the *Miller loop*.

3.1.6 Barreto-Naehrig Curves

One must be careful when choosing an elliptic curve E for implementing a pairing-based protocol. As an example, consider the Tate pairing. The coordinates of the points of the elliptic curve we are working with lie in an extension field $\mathbb{F}_{q^k}$ of the field of definition of E. In this situation, one must be careful to balance security and practicality. If the embedding degree k is too small, then the DLP may not be intractable in $\mathbb{F}_{q^k}$. However,

Algorithm 3.1 Miller's Algorithm to compute the Tate pairing [33]

Require: $P, Q \in E[n]$ and let the binary representation of n be $n = (n_{l-1}n_{l-2}...n_1n_0)_2 \in \mathbb{N}$

1: $f \leftarrow 1$, $T \leftarrow P$,
2: **for** $i = l-2$ to 0 **do**
3: $\quad f \leftarrow f^2 \cdot \frac{l_{T,T}(Q)}{v_{2T}(Q)}$
4: $\quad T \leftarrow 2T$
5: $\quad$ **if** $l_i \neq 0$ **then**
6: $\quad\quad f \leftarrow f \cdot \frac{l_{T,P}(Q)}{v_{T+P}(Q)}$
7: $\quad\quad T \leftarrow T + P$
8: $\quad$ **end if**
9: **end for**
10: $f \leftarrow f^{\frac{q^k-1}{n}}$
11: **return** f

if the embedding degree is too large then computations could easily become unwieldy. Balasubramanian and Koblitz showed that curves with suitably low embedding degrees are rare. In particular, they showed that one can expect $k \approx q$ for a randomly selected prime-order elliptic curve over a randomly selected prime-order field [5]. Furthermore, the probability that $k \leq \log^2 q$ is *very* small. Similar results have been obtained for $\mathbb{F}_q$ where q is a prime power [30].

In 2005, Barreto and Naehrig discovered what are now called Barreto-Naehrig (BN) curves [7]. Let $q(x)$ and $n(x)$ be the polynomials

$$q(x) = 36x^4 + 36x^3 + 24x^2 + 6x + 1,$$
$$n(x) = 36x^4 + 36x^3 + 18x^2 + 6x + 1.$$

Randomly choose an integer x until both $q(x)$ and $n(x)$ evaluate to a prime number and let $q = q(x)$. Let $b \in \mathbb{F}_q^*$ such that $b+1$ is a quadratic residue. Then consider the curve

$$E : y^2 = x^3 + b. \tag{3.1.3}$$

If E does not have order $n(x)$, then select a new $b \in \mathbb{F}_q^*$, such that $b+1$ is a quadratic residue and see if E has order $n(x)$. Proceed until a suitable b is found. Then Equation 3.1.3 will define a BN-curve. In addition to having prime order, BN-curves will have an embedding degree of 12 with respect to $n(x)$. Furthermore, $P = (1, \sqrt{b+1})$ is a point on E that generates $E(\mathbb{F}_q)$. Choosing a sufficiently large prime q ensures that $\mathbb{F}_{q^k}$ will be large enough that the DLP will be intractable.

3.2 Optimal Pairings and the Optimal Ate Pairing

For the remainder of this Chapter let

$$\lambda = q \pmod{n},$$
$$m = (\lambda^k - 1)/n.$$

3.2.1 The Ate Pairing

The Frobenius endomorphism is the map $\phi : E(\mathbb{F}_{q^k}) \to E(\mathbb{F}_{q^k})$ given by $(x, y) \mapsto (x^q, y^q)$. If k is even then we can find d such that $\mathbb{F}_{q^d} \subset \mathbb{F}_{q^k}$ and $\mathbb{F}_{q^k}$ is a quadratic extension. An element in $a \in \mathbb{F}_{q^k}$ can be represented as $a = \alpha + \omega\beta$ where $\alpha, \beta \in \mathbb{F}_{q^d}$ and w is a square root of an element in $\mathbb{F}_{q^d}$ such that $w \notin \mathbb{F}_{q^d}$. This endomorphism then has two eigenspaces in $E(\mathbb{F}_{q^k})[n]$ with eigenvalues 1 and q. Since $x = x^q$ for all $x \in \mathbb{F}_q$, the 1-eigenspace consists of points of $E(\mathbb{F}_q)$. The q-eigenspace consists of points $(\alpha, \omega\beta)$ [5]. In order to define the Ate pairing, we assume k is even and restrict Q to be a point of this form. Let G_1 denote the 1-eigenspace and G_2 denote the q-eigenspace when restricted to $E[n]$. Then both eigenspaces form subgroups of order n when restricted to $E[n]$.

The Ate pairing is an optimized version of the Tate pairing restricted to Frobenius eigenspaces. It should be noted that Miller's algorithm to compute the Tate pairing can be easily modified to compute the Ate pairing, as well as the Optimal Ate pairing. The Ate pairing is referred to as an *optimized* version of the Tate pairing since the Miller loop is purposefully designed to be shorter than that of the Tate pairing. Hence, the Ate pairing is easier to compute. We follow the approach in [42] and derive the Ate pairing. We will then motivate and construct the Optimal Ate pairing.

We start with two lemmas about Miller functions from [21, 42].

Lemma 3.1. *$f_{ab,Q} = f_{a,Q}^b \cdot f_{b,aQ}$ for all $a, b \in \mathbb{Z}$.*

Proof. Observe

$$\begin{aligned}\operatorname{div}(f_{a,Q}^b) &= b(a(Q) - ([a]Q) - (a-1)(\infty)) \\ &= (ba(Q) - b([a]Q) - b(a-1)(\infty)) \\ \operatorname{div}(f_{b,aQ}^b) &= b(aQ) - ([ab]Q) - (b-1)(\infty)).\end{aligned}$$

This gives us

$$\begin{aligned}\operatorname{div}(f_{a,Q}^{b} \cdot f_{b,aQ}^{b})) &= (ba(Q) - b([a]Q) - b(a-1)(\infty)) + b(aQ) - ([ab]Q) - (b-1)(\infty)) \\ &= ba(Q) - ([ab]Q) - (ba-1)(Q) \\ &= \operatorname{div}(f_{ab,Q}).\end{aligned}$$

□

Lemma 3.2. *Let $m \in \mathbb{Z}$ and let $e(Q,P)$ be the Tate pairing with $Q \in G_2$ and $P \in G_1$. Then $e(Q,P)^m = f_{nm,Q}(P)^{(q^k-1)/n}$.*

Proof. We have

$$\begin{aligned}e(Q,P)^m &= f_{n,Q}(P)^{(q^k-1)m/n} \\ &= \frac{f_{nm,Q}(P)^{(q^k-1)/n}}{f_{m,nQ}(P)} && \text{(by Lemma 3.1)} \\ &= \frac{f_{nm,Q}(P)^{(q^k-1)/n}}{f_{m,\infty}(P)} && (Q \text{ is an } n\text{-torsion point}) \\ &= f_{nm,Q}(P)^{(q^k-1)/n}.\end{aligned}$$

□

We want to find a multiple of n such that computing $f_{mn,Q}(P)$ is reduced to computing a power of a function $f_{\lambda,Q}(P)$ with a shorter Miller loop. We claim that such a multiple of n is $\lambda^k - 1$. We define the *reduced Ate pairing* as $a(Q,P) : G_2 \times G_1 \to \mu_n$ given by

$$(Q,P) \mapsto f_{\lambda,Q}(P)^{\frac{(q^k-1)}{n}}. \tag{3.2.1}$$

Lemma 3.3 ([42], Lemma 1)**.** *The reduced Ate pairing is a bilinear pairing which is non-degenerate for $n \nmid m$.*

Proof. We have

$$\begin{aligned}e(Q,P)^m &= f_{nm,Q}(P)^{(q^k-1)/n} && \text{(by Lemma 3.2)} \\ &= f_{\lambda^k-1,Q}(P)^{(q^k-1)/n}.\end{aligned}$$

Since Q is an n-torsion point, we see that $\infty = nQ = nmQ = [\lambda^k - 1]Q$. Let l be the line though $\lambda^k Q$ and $-Q$ and let v be the line through $(\lambda^k - 1)Q$. By applying Theorem 3.5 to the line above we get

$$f_{\lambda^k-1,Q}(P)^{(q^k-1)/n} = \left(f_{\lambda^k,Q}(P) \cdot f_{-1,Q}(P)\frac{l(P)}{v(P)}\right)^{(q^k-1)/n}.$$

Since $\lambda^k Q = Q$ and $(\lambda^k - 1)Q = \infty$, we see that $l(P)$ and $v(P)$ cancel out and we're left with

$$\begin{aligned}\left(f_{\lambda^k,Q}(P) \cdot f_{-1,Q}(P)\frac{l(P)}{v(P)}\right)^{(q^k-1)/n} &= \left(f_{\lambda^k,Q}(P) \cdot f_{-1,Q}(P)\right)^{(q^k-1)/n} \\ &= f_{\lambda^k,Q}(P)^{(q^k-1)/n}.\end{aligned}$$

The last step above is realized by noting that $f_{-1,Q}(P) = 1$. Repeatedly applying Lemma 3.1 gives us

$$\begin{aligned}f_{\lambda^k,Q}(P)^{(q^k-1)/n} &= \left(f_{\lambda,Q}^{\lambda^{k-1}}(P) \cdot f_{\lambda^{k-1},\lambda Q}(P)\right)^{(q^k-1)/n} \\ &= \left(f_{\lambda,Q}^{\lambda^{k-1}}(P) \cdot f_{\lambda,\lambda Q}^{\lambda^{k-2}}(P) \cdot f_{\lambda^{k-2},\lambda^2 Q}(P)\right)^{(q^k-1)/n} \\ &\vdots \\ &= \left(f_{\lambda,Q}^{\lambda^{k-1}}(P) \cdot f_{\lambda,\lambda Q}^{\lambda^{k-2}}(P) \cdot \cdots \cdot f_{\lambda,\lambda^{k-1} Q}(P)\right)^{(q^k-1)/n} \\ &= \left(\prod_{i=0}^{k-1} f_{\lambda,q^i Q}^{\lambda^{k-1-i}}\right)^{(q^k-1)/n}.\end{aligned}$$

By using induction and Lemma 3.1, one can prove the following Lemma:

Lemma 3.4 ([21], Fact 2.2.5)**.** *$f_{a,q^i(Q)}(P) = f_{a,Q}(P)^{q^i}$ for all $a \in \mathbb{Z}$ and $Q \in G_2$.*

Using Lemma 3.4 above we obtain

$$\begin{aligned}\prod_{i=0}^{k-1} f_{\lambda,q^i Q}^{\lambda^{k-1-i}} &= \prod_{i=0}^{k-1} f_{\lambda,Q}^{q^i q^{k-1-i}} \qquad (\text{since } \lambda = q \pmod n) \\ &= \prod_{i=0}^{k-1} f_{\lambda,Q}^{q^{k-1}} \\ &= f_{\lambda,Q}(P)^{\sum_{i=0}^{k-1} q^{k-1}}.\end{aligned}$$

Thus we see that $e(Q,P)^m = f_{\lambda,Q}(P)^{\frac{q^k-1}{n}kq^{k-1}}$. Let $r = k^{-1}q^{-(k-1)} \pmod n$. Then,

$$\begin{aligned} a(Q,P) = e(Q,P)^{mr} &= e(Q,P)^{m[k^{-1}q^{-(k-1)} \pmod n]} \\ &= f_{\lambda,Q}(P)^{\left(\frac{q^k-1}{n}[k^{-1}q^{-(k-1)} \pmod n]kq^{k-1}\right)} \\ &= f_{\lambda,Q}(P)^{\frac{q^k-1}{n}}, \end{aligned}$$

since n is prime, $n \nmid \frac{\lambda^k-1}{n}$ and $m = \frac{\lambda^k-1}{n}$. We also see that $m[(k^{-1}q^{-(k-1)} \bmod n)]$ is relatively prime to n. This, along with the non-degeneracy and bilinearity of the Tate pairing, implies that the reduced Ate pairing is non-degenerate and bilinear. □

We have $\lambda = q \pmod n$. By Hasse's theorem, $n = \#E(\mathbb{F}_q) = q+1-t$ where $|t| \leq 2\sqrt{q}$ and t is the trace of Frobenius for E (see [39], Chapter V). Thus, $q = t-1 \pmod n$. Since BN curves have a strictly positive trace of Frobenius [7], we have that $t > 0$ and $\lambda = t-1$. Therefore λ is almost half the length of n. Thus, computing the reduced Ate pairing shortens the Miller loop in comparison to computing the corresponding Tate pairing.

3.2.2 Optimal Pairings

In this section we follow [42] and present the *Optimal Ate Pairing* which is a variation of the Ate pairing that further shortens the Miller loop with regard to a natural lower bound. We first discuss this lower bound on the length of the Miller loop and then define this pairing.

The *j-th cyclotomic polynomial* $\Phi_j \in \mathbb{Z}[x]$ is defined to be the unique irreducible polynomial with integer coefficients that divides $x^j - 1$ and that does not divide $x^k - 1$ for all $k < j$. One can show that the complex roots of the polynomial are precisely the j-th primitive roots of unity $e^{2\pi i\frac{k}{j}}$ where $k < j$ and gcd(k,j)=1. This implies the equality

$$\Phi_j(x) = \prod_{1\leq k\leq j \ \&\ \gcd(k,j)=1} \left(x - e^{2\pi\frac{k}{j}}\right). \tag{3.2.2}$$

The Möbius inversion formula allows us to explicitly express $\Phi_j(x)$ as

$$\Phi_j(x) = \prod_{l|j} \left(x^l - 1\right)^{\mu(j/l)} \tag{3.2.3}$$

where $\mu(k)$ is the Möbius function which takes on values {-1,0,1} depending on the factorization of k into prime factors. Consider an Ate pairing with Miller function $f_{\lambda_i,Q}$ for

$\lambda_i = q^i \pmod{n}$. Recall that n is a large prime such that $n \mid q^k - 1$. Let $d = \gcd(i, k)$. Then $n \mid \left((q^i)^{k/d} - 1\right)$. Thus, by referring to equation 3.2.3 above and by noting that $\mu(1) = 1$, we see that $n \mid \Phi_{k/d}(\lambda_i)$. Since the degree of any cyclotomic polynomial $\Phi_j(x)$ is $\phi(j)$ where ϕ is Euler's Totient function, we see that the minimal value of λ_i will be close to $n^{1/\phi(k/d)}$. For $d = 1$ we see that the smallest lower bound will be close to $n^{1/\phi(k)}$. This bound is attained for several families of elliptic curves including cyclotomic families [18]. This bound motivates the following definition.

Definition 3.7 ([42]). *Let $e : G_1 \times G_2 \to G_3$ be a bilinear pairing such that $|G_1| = |G_2| = |G_3| = n$. Let the field of definition of G_3 be $\mathbb{F}_{q^k}$. Then we call e an* optimal *pairing if it can be computed in $\log_2 n/\phi(k) + \epsilon(k)$ basic Miller iterations (i.e iterations of the Miller loop) where $\epsilon(k) \leq \log_2(k)$.*

Miller's algorithm for the Tate pairing presented in Section 3.1.5 can also be used to compute the Ate pairing. For an optimal pairing, the length of the Miller loop will be $O(\log(\lambda_i))$. Thus, the above definition of optimality seems appropriate. The following conjecture, and the cases for which it is proven, further indicates that this notion of optimality is the correct one.

Conjecture 3.1 ([42]). *Any bilinear pairing on an elliptic curve without efficiently computable endomorphisms different from powers of Frobenius requires at least $O(\log_2 n/\phi(k))$ basic Miller iterations. Furthermore, the O-constant depends only on k.*

The reason we exclude elliptic curves with efficiently computable endomorphisms different from powers of Frobenius is that the presence of such endomorphisms have enabled many researchers to reduce the number of basic Miller iterations required in the Miller loop. In 2008, Hess proved Conjecture 3.1 for a very general class of pairing functions that includes all known pairing functions, including the Optimal Ate pairing [23].

3.2.3 The Optimal Ate Pairing

In constructing the O-Ate pairing, one uses Lemma 3.2 and attempts to find a multiple of $\lambda = mn$ such that its base-q expansion $\lambda = \sum_{i=0}^{l} c_i q^i$ has small coefficients. The following theorem shows that such an expansion gives rise to a bilinear pairing. Thus, our problem of finding an optimal pairing is reduced to finding a value of λ with small coefficients.

Theorem 3.6 ([42], Theorem 4). *Let $\lambda = mn$ with $n \nmid m$ and write $\lambda = \sum_{i=0}^{l} c_i q^i$. Then $a_{[c_0,\ldots,c_l]} : G_2 \times G_1 \to \mu_n$ defined as*

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

where $s_i = \sum_{j=i}^{l} c_j q^j$. $a_{[c_0,\ldots,c_l]}$ *defines a bilinear pairing if*

$$mkq^{k-1} \not\equiv ((q^k-1)/n) \cdot \sum_{i=0}^{l} i c_i q^{i-1} \ (mod\ n).$$

See [21, 42] for a proof of the above theorem.

We wish to determine $\lambda = \sum_{i=0}^{l} c_i q^i$ with small c_i. Since $n \mid q^k - 1$ we see that $\Phi_k(q) = 0 \pmod{n}$. It is therefore tempting to take $\lambda = \Phi_k(q)$. However, this always leads to a degenerate pairing [42]. In order to avoid degenerate pairings and to obtain a relation with small coefficients, it generally suffices to consider powers q^i for $i = 0, \ldots, \phi(k) - 1$. To do this we try to find a short vector in the $\phi(k)$-dimensional lattice

$$L := \begin{pmatrix} n & 0 & 0 & \ldots & 0 \\ -q & 1 & 0 & \ldots & 0 \\ -q^2 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & & \ddots & \\ -q^{\phi(k)-1} & 0 & \ldots & 0 & 1 \end{pmatrix}.$$

Finding such a vector will give us an integer relation among the powers q^i for $i = 0, \ldots, \phi(k)-1$ that is equal to 0 modulo n. Since k is small this is easy in practice. One could, for example, use enumeration as described in [37]. It is not immediately clear that this approach will give us a short-enough vector for our pairing to be optimal or even that it will give us a vector that induces a non-degenerate pairing.

Recall that the volume of a lattice is defined as the volume of the fundamental parallelepiped associated with L and is thus the absolute magnitude of the determinant of L. One easily sees that the volume of the above lattice is n. As a consequence of Minkowski's theorem we get that there exists a vector $V \in L$ such that $||V||_\infty \leq n^{1/\phi(k)}$ where $||V||_\infty = \max_i |v_i|$ [34]. It is a necessary condition to obtain an optimal pairing that no single value of c_i is greater than $n^{1/\phi(k)}$. Thus, we see that this necessary condition can be fulfilled. However, this condition is not sufficient as, even if it is fulfilled, we may still require too many basic Miller iterations for our pairing to be optimal. Vercauteren suggests a method for solving this shortcoming based on considering multiple short vectors

in L each having a minimum number of coordinates of size $n^{1/\phi(k)}$ [42]. Using his method will often give us an optimal pairing. Additionally, Vercauteren shows that using more general expressions of $\lambda = \sum_{i=0}^{l} c_i q^i$, besides those that arise as vectors in L, will not lead to substantially more efficient pairings than those resulting from the use of his method.

The O-Ate Pairing on BN-Curves

Recall the family of BN-Curves, introduced in Section 3.1.6. This family of curves has $k = 12$. Both q and n are given by equations parameterized by an integer x:

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

There are at least two different ways one can define an Optimal Ate pairing using this curve. Since $\phi(k) = 4$, one can either consider the 4-dimensional lattice L and use one of the shortest vectors in that lattice for the Euclidean norm (there are two) to define λ. Alternatively, we can look for short vectors with a minimal number of coefficients of size x.

We have

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

Hence,

$$\lambda_0 := 6x + 1 + q(x) - q(x)^2 + q(x)^3 = 0 \pmod{n(x)}.$$

Applying Theorem 3.6 with λ_0 gives the following O-Ate pairing on BN-curves where $q = q(x)$:

$$(Q, P) \to f_{6x+2,Q}(P) f_{1,Q}^{q}(P) f_{-1,Q}^{q^2}(P) f_{1,Q}^{q^3}(P) d(P), \tag{3.2.4}$$

where $d(P)$ is given by

$$d(P) = l_{[q-q^2+q^3]Q(P),[6x+2]Q}(P) l_{[-q^2+q^3]Q,[q]Q}(P) l_{[q^3]Q,[-q^2]Q}(P). \tag{3.2.5}$$

The computation of $d(P)$ can be made simpler by using the following lemma:

Lemma 3.5. *$d(P)^{\frac{q^k-1}{n}} = h(P)^{\frac{q^k-1}{n}}$ where $h(P) = l_{[6x+2]Q,qQ}(P) l_{[6x+2]Q+qQ,-q^2Q}(P)$.*

This lemma was discovered by Naehrig, Niederhagen and Schwabe [36] and is proven by manipulating the divisors of $d(P)$ and $h(P)$ to show that they are essentially equal.

One also notes that $f_{1,Q} = f_{-1,Q} = 1$. Since one is only concerned with computing $d(P)^{\frac{q^k-1}{n}}$ in the computation of the O-Ate pairing, one can replace $d(P)$ by the simpler expression $h(P)$ in this computation. Thus, we obtain the following O-Ate pairing on BN-curves,

$$(Q, P) \to (f_{6x+2,Q}(P) \cdot h(P))^{\frac{q^k-1}{n}}. \tag{3.2.6}$$

For the remainder of the discussion let

$$f := f_{6x+2,Q}(P) \cdot h(P).$$

3.3 An Efficient Implementation of the O-Ate Pairing on ARM Processors

In this section we describe an efficient computation of the O-Ate pairing over BN-curves BN-254, BN-446 and BN-638 [20] specifically designed for use on the ARM platform. This implementation was presented by Grewal et al. [21, 22]. Their implementation uses an appropriate adaptation of Miller's algorithm for the computation of the pairing along with many optimizations to increase efficiency. We summarize some of the more-important optimizations and discuss implementation details.

3.3.1 Grewal et al.'s Optimizations

Representation of Extension Fields

The efficient computation of a pairing on a BN-curve relies on arithmetic over finite fields. Therefore, one needs to implement the underlying fields efficiently. The implementation uses $\mathbb{F}_{q^2}$, $\mathbb{F}_{q^6}$ and $\mathbb{F}_{q^{12}}$. The IEEE P1363.3 standard recommends using towers to represent $\mathbb{F}_{q^k}$ [1]. The curve BN-254 has prime q congruent to 3 mod 8, while the curves BN-446 and BN-638 have primes q congruent to 7 mod 8. In both cases, a tower construction is used, the ideas of which come from Benger and Scott [9].

If $q = q(x)$ is the prime characteristic of the field over which a BN-curve is defined and if $x = 7$ or 11 (mod 12), then $y^6 - (1 + \sqrt{-1})$ is irreducible over $\mathbb{F}_{q^2} = \mathbb{F}_q(\sqrt{-1})$. Thus,

we see that $z^3 - (1 + \sqrt{-1})$ is irreducible over $\mathbb{F}_{q^2} = \mathbb{F}_q(\sqrt{-1})$ as well. This gives us the following tower of extension fields:

$$\begin{cases} \mathbb{F}_{q^2} = \mathbb{F}_q[i]/(i^2 - \beta), & \text{where } \beta = -1. \\ \mathbb{F}_{q^6} = \mathbb{F}_{q^2}[v]/(v^3 - \xi), & \text{where } \xi = 1 + i. \\ \mathbb{F}_{q^{12}} = \mathbb{F}_{q^6}[w]/(w^2 - v). \end{cases}$$

For the BN-254 curve defined by $E : y^2 = x^3 + 2$ over the 254-bit prime field $\mathbb{F}_q$ with $x = -(2^{62} + 2^{55} + 1)$, we see that $x = 11 \pmod{12}$ and we can use the above tower scheme. This towering scheme is efficient as the coefficients of the irreducible polynomial are minimal, thus ensuring less operations for field arithmetic.

If $q = q(x)$ is the prime characteristic of the field over which a BN-curve is defined and if $x = 2$, 3, 4, 6, 7 or 8 (mod 9), then $y^6 - (1 + \sqrt{-2})$ is irreducible over $\mathbb{F}_{q^2} = \mathbb{F}_q(\sqrt{-2})$. Thus, we see that $z^3 - (1 + \sqrt{-2})$ is irreducible over $\mathbb{F}_{q^2} = \mathbb{F}_q(\sqrt{-2})$ as well. This gives us the following tower of extension fields:

$$\begin{cases} \mathbb{F}_{q^2} = \mathbb{F}_q[i]/(i^2 - \beta), & \text{where } \beta = -2. \\ \mathbb{F}_{q^6} = \mathbb{F}_{q^2}[v]/(v^3 - \xi), & \text{where } \xi = 1 + i. \\ \mathbb{F}_{q^{12}} = \mathbb{F}_{q^6}[w]/(w^2 - v). \end{cases}$$

The BN-446 curve is given by $E : y^2 = x^3 + 257$ defined over the 446-bit prime field $\mathbb{F}_q$ with $x = 2^{110} + 2^{36} + 1$. The BN-638 curve is defined by $E : y^2 = x^3 + 257$ over the 638-bit prime field $\mathbb{F}_q$ with $x = 2^{158} - 2^{128} - 2^{68} + 1$. In both cases x has the required value modulo 9 to use the second construction. The value of x modulo 9 is not suitable for us to use the first construction, which is slightly faster in practice. Again, this towering scheme optimizes field arithmetic because the coefficients of the irreducible polynomials are minimal.

Field Arithmetic

Arithmetic over finite fields is done using Karatsuba multiplication and squaring while using lazy reduction for inversion routines as in [3]. The main idea of lazy reduction techniques is to minimize the number of modular reductions required by carrying out the maximum amount of preliminary arithmetic possible before reducing. By applying lazy reduction techniques, one $\mathbb{F}_q$ reduction per $\mathbb{F}_{q^2}$ inversion and 13 $\mathbb{F}_q$ reductions per $\mathbb{F}_{q^{12}}$ inversion are saved.

Twists on Elliptic Curves

Let E be an elliptic curve over $\mathbb{F}_q$. An elliptic curve E' is called a *twist* of E if there exists an isomorphism $\psi : E'(\mathbb{F}_{q^r}) \to E(\mathbb{F}_{q^r})$ defined over the extension field $\mathbb{F}_{q^r}$. The minimum extension degree for which there exists an isomorphism is called the *degree* of the twist.

BN-curves have two sextic twists when considered over $\mathbb{F}_{q^2}$ [39]. Let $E : y^2 = x^3 + b$ be a BN-curve defined over $\mathbb{F}_q$. Let ξ be as in Section 3.3.1. Then

$$E' : y^2 = x^3 + \frac{b}{\xi}, \tag{3.3.1}$$

$$E'' : y^2 = x^3 + \xi b, \tag{3.3.2}$$

are sextic twists of E. They are known as a *D-type* twist and an *M-type* twist respectively. Their untwisting isomorphisms are given as ψ' and ψ'' respectively where

$$\begin{aligned}
&\psi' : E' \to E, \\
&\psi' : (x, y) \mapsto (\psi^{\frac{1}{3}}x, \psi^{\frac{1}{2}}y) = (w^2x, w^3y), \\
&\psi'' : E'' \to E, \\
&\psi'' : (x, y) \mapsto (\psi^{\frac{-2}{3}}x, \psi^{\frac{-1}{2}}y) = (\psi^{-1}w^4x, \psi^{-1}w^3y).
\end{aligned}$$

Exactly one of the above twists will map points in G_2 to points on the twisted curve over $\mathbb{F}_{q^2}$. Therefore, G_2 can be represented using either $E'[n](\mathbb{F}_{q^2})$ or $E''[n](\mathbb{F}_{q^2})$. We call this G_2'. Using G_2' in place of G_2 gives us the *twisted O-Ate pairing on BN-curves*

$$a_t : G_2' \times G_1 \to \mu_n. \tag{3.3.3}$$

A pair of points $(Q', P) \in G_2' \times G_1$ is mapped analogously to how the O-Ate pairing maps a pair of points $(Q, P) \in G_2 \times G_1$. However, the point Q is replaced by the point $\psi(Q')$ where $\psi = \psi'$ or ψ'' depending on whether we are using a D-type or M-type twist.

Twists can be used to substantially accelerate the Miller loop. First, one determines the correct twist to use by checking which of the curves has order dividing n. For either of the above tower schemes, one can take $\{1, v, v^2, w, vw, v^w\}$ as a basis of $\mathbb{F}_{q^{12}}$ over $\mathbb{F}_{q^2}$.

If the correct twist is a D-type twist then the computation of the untwisting isomorphism is almost free as w^2 and w^3 are basis elements for representing an element of $\mathbb{F}_{q^{12}}$. Thus, in this case, we compute the pairing on the original curve and perform arithmetic

on the twisted curve. We then use the efficient untwisting map to map the result back to the original curve.

If the correct twist is an M-type twist then the untwisting map is not as efficient. However the twisting map defined by

$$(\psi'')^{-1} : (x, y) \mapsto (w^2 x, w^3 y),$$

is nearly free. Thus, in this case, we compute the pairing on the twisted curve E''.

Final Exponentiation

The final step of Miller's algorithm requires f to be raised to the exponent $\frac{q^k-1}{n}$. Since BN-curves have $k = 12$, we can implement $\mathbb{F}_{q^k}$ as a quadratic extension of $\mathbb{F}_{q^d}$ where $d = 6$. Then,

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

Since k was chosen to be minimal such that $n \mid q^k - 1$, we see that $n \nmid q^d - 1$. Since n is prime, $n \mid q^d + 1$. We therefore split exponentiation into two parts, first, exponentiation by $q^d - 1$ and then $\frac{q^d+1}{n}$.

Let $a \in \mathbb{F}_{q^k}$ then $a = \alpha + \beta s$ where $\alpha, \beta \in \mathbb{F}_{q^d}$ and s is an adjoined square root. By the generalization of the Frobenius endomorphism to extension fields, we see that

$$(\alpha + \beta a)^{q^d} = a - \beta s. \tag{3.3.4}$$

The above relation gives us the ability to compute f^{q^d-1} in an easy way. Computing $(f \cdot h)^{\frac{q^d+1}{n}}$ is more difficult. This exponent factors as

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

Like in Equation (3.3.4), exponentiating by q^2+1 is an easy operation due to the presence of the Frobenius endomorphism. Thus, we need only worry about exponentiating by $\frac{q^4-q^2+1}{n}$.

Computing $f^{\frac{q^4-q^2+1}{n}}$ relies on the observation that computing the O-Ate pairing to some suitable power will still give a bilinear pairing. Instead of using $\frac{q^4-q^2+1}{n}$ as the exponent, we can choose a multiple of it. Fuenetes-Castañeda et al. demonstrate how to choose an

appropriate multiple in [19] and this is how Grewal et al. implement the final exponentiation in [21, 22]. The final exponentiation is computed as

$$af^{6x^2}fb^q a^{p^2}(bf-1)^{p^3},$$

where $a = f^{12x^3}f^{6x^2}f^{6x}$ and $b = a(f^{2x})^{-1}$. At the time that [22] was published, this was the fastest known method.

3.3.2 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 $\mathbb{F}_{q^2}$, respectively.

The authors of the implementation examined the use of Jacobian, affine and homogenous coordinate systems. It was found that homogenous coordinates were most efficient for this implementation. Explicit formulas for some of the arithmetic operations, along with operation costs can be found in [21, 22].

It should be noted that the most efficient method of curve arithmetic for one platform is not necessarily the most efficient method for all platforms. For example, ARM optimization differs from PC optimization because ARM has different performance characteristics. On the ARM platform, 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, the choice of formulas or coordinate systems geared towards one platform may not be optimal for another.

One example of this phenomenon that is applicable to the implementation is as follows. Let $T = (X, Y, Z) \in E'(\mathbb{F}_{q^2})$ be a point on the twist of E in homogeneous coordinates. Aranha et al. observe that $\tilde{m} - \tilde{s} \approx 3\tilde{a}$ on a PC processor [3] . However, on ARM processors, Grewal at al. observe that $\tilde{m} - \tilde{s} \approx 6\tilde{a}$ [22]. In order to compute $2T = (X', Y', Z')$ one is required to compute XY as an intermediate step. One can compute XY by computing XY directly or by computing $\frac{(X+Y)^2 - X^2 - Y^2}{2}$. The first method has costs of $3\tilde{a} + \tilde{s} = \tilde{m}$ on a PC and $6\tilde{a} + \tilde{s} = \tilde{m}$ on an ARM processor. Assuming the cost of division by two is equivalent to the cost of addition and that X^2 and Y^2 are precomputed, the second method has cost $4\tilde{a} + \tilde{s}$ on both platforms. Thus, we see that the first method is most efficient on a PC platform, while the second method is most efficient on the ARM platform.

For detailed operation counts, the reader can refer to [21, 22].

3.3.3 Implementation Results

Grewal et al. presented the results of this implementation on a Marvell Kirkwood 6281 ARMv5 CPU processor at 1.2 GHz, an IPad 2 using an ARMv7 Cortex-A9 MPCore processor at 1.0 GHz, and a Samsung Galaxy Nexus using an ARM Cortex-A9 at 1.2 GHz [21, 22].

The software is based on version 0.2.3 of the RELIC toolkit with a GMP 5.0.2 backend. The authors implemented field addition and multiplication using hand-optimized assembly language for the BN-254 curve. We discuss more details of assembly language in Chapter 4.

The results of the experiment are presented in Table 3.1. For ease of comparison, the results from [2] are also presented there. This implementation's timings are roughly 3-4 times faster than those reported in [2]. ML, FE, O-A(a), O-A(p) denote the amount of time it takes to execute the Miller loop, final exponentiation, the Optimal Ate pairing using affine coordinates and the Optimal Ate pairing using projective (homogenous) coordinates. I/M denotes the ratio of i to m.

See Section 4.3 for a description of our contributions and results pertaining to computing the O-Ate pairing over BN curves.

Table 3.1: Timings for affine and projective pairings on different ARM processors and comparisons with prior literature. Times for the Miller loop (ML) in each row reflect those of the faster pairing.

Marvell Kirkwood (ARM v5) Feroceon 88FR131 at 1.2 GHz [22]													
Field Size	Language	Operation Timing [μs]											
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	ASM	0.12	1.49	1.12	17.53	11.8	0.28	4.08	3.44	23.57	9,722	6,176	16,076	15,898
	C	0.18	1.74	1.02	17.40	10.0	0.35	4.96	4.01	24.01	11,877	7,550	19,427	19,509
446-bit	C	0.20	3.79	2.25	34.67	9.1	0.38	10.74	8.57	48.90	42,857	23,137	65,994	65,958
638-bit	C	0.27	6.82	3.83	52.33	7.7	0.51	18.23	14.93	77.11	98,044	51,351	149,395	153,713
iPad 2 (ARM v7) Apple A5 Cortex-A9 at 1.0 GHz [22]													
Field Size	Language	Operation Timing [μs]											
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	C	0.16	1.28	0.93	13.44	10.5	0.25	3.48	2.88	19.19	8,338	5,483	14,604	13,821
446-bit	C	0.16	2.92	1.62	27.15	9.3	0.26	8.03	6.46	37.95	32,087	17,180	49,365	49,267
638-bit	C	0.20	5.58	2.92	43.62	7.8	0.34	15.07	12.09	64.68	79,056	40,572	119,628	123,410
Galaxy Nexus (ARM v7) TI OMAP 4460 Cortex-A9 at 1.2 GHz [22]													
Field Size	Language	Operation Timing [μs]											
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	ASM	0.05	0.93	0.55	9.42	10.1	0.10	2.46	2.07	13.79	6,147	3,758	10,573	9,905
	C	0.07	0.98	0.53	9.62	9.8	0.13	2.81	2.11	14.05	6,859	4,382	11,839	11,241
446-bit	C	0.12	2.36	1.27	23.08	9.8	0.22	6.29	5.17	32.27	25,792	13,752	39,886	39,544
638-bit	C	0.19	4.87	3.05	38.45	7.9	0.45	12.20	10.39	56.78	65,698	33,658	99,356	99,466
NVidia Tegra 2 (ARM v7) Cortex-A9 at 1.0 GHz [2]													
Field Size	Language	Operation Timing [μs]											
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	C	0.67	1.72	n/a	18.35	10.7	1.42	8.18	5.20	26.61	26,320	24,690	51,010	55,190
446-bit	C	1.17	4.01	n/a	35.85	8.9	2.37	17.24	10.84	54.23	97,530	86,750	184,280	195,560
638-bit	C	1.71	8.22	n/a	56.09	6.8	3.48	31.81	20.55	91.92	236,480	413,370	649,850	768,060

Chapter 4

Assembly Language and Our Optimizations

Our main contribution is in the area of low-level optimization of cryptographic protocols. We modified the key-exchange software to implement elliptic curve arithmetic in pure C and wrote ARMv7 and x86-64 assembly language routines to implement multi-precision integer arithmetic. These assembly routines are used in both the key-exchange software and in the pairing software. In this chapter we first provide an overview of assembly language, focussing on the ARMv7 and x86-64 assembly languages. We then discuss our optimizations to both the key-exchange and pairing software.

4.1 Assembly Language

Assembly languages are low-level programming languages in which there is a strong correspondence between the language and the architecture's machine code instructions. Assembly code is converted into machine code by a program called an *assembler*. Each assembly language is specific to a certain computer architecture. For example, ARMv7 assembly language differs strongly from x86-64 (PC) assembly language. In this section we give a high-level introduction to assembly language and describe some differences between ARMv7 and x86-64 assembly languages. We also focus on some specific features of these languages that we use in our optimizations.

Programs written in assembly language consist of a series of mnemonic processor instructions, pseudo-instructions and data. Mnemonic processor instructions are in one-to-one correspondence to machine code but have meaningful names which makes their use

more user-friendly. The assembler converts these mnemonic instructions directly to the corresponding machine code instruction. A pseudo-instruction is converted to more than one machine code instruction by the assembler. For the remainder of this thesis, we assume the term *instruction* refers to either a mnemonic processor instruction or pseudo-instruction.

Generally, an assembly language instruction will be followed by a list of arguments which consist of data (or the location of data) and sometimes other parameters. The use of data comes in roughly 3 flavours. One can use an *immediate* operand, a *register* operand or a *memory address*. An immediate operand is a specific number or binary string. It is used when you want to pass a specific value to an instruction. Using a register address is the most efficient way of working with data. A register refers a specific data-storage located on the processor. Most instructions can only be executed on data that has already been loaded into registers. In a sense, registers act as variables do in higher-level programming languages. Assembly languages also provide *load* and *store* instructions that load data from memory or immediate operands into registers and store data from registers or immediate operands into memory. A memory address is a data-storage location outside of the processor in main memory. The movement of data from memory to a register (and from a register to memory) is generally expensive and should be minimized.

```
LDR r6, =0xa        %% loads 10 (base-16) into the register r6
MOV r9, [r6]        %% loads the value in register r6 into the register r9
ADD r6, r6, r9      %% adds the value in registers r6 and r9, storing the
                    %% result in r6
```

The above sample of ARM assembly language adds the number 10 to itself. The result is 20 and stored in register `r6`. `LDR` is pseudo-instruction and both `MOV` and `ADD` are mnemonic processor instructions. The `%%` symbol denotes the beginning of comments. The code below performs the same function in x86 assembly language.

```
MOVQ $10, %r6       %% loads 10 into the register r6
MOVQ %r6, %r9       %% loads the value in register r6 into the register r9
ADDQ r9, r6         %% adds the value in registers r6 and r9, storing the
                    %% result in r6
```

Both of these code samples are similar but have subtle differences. Most notably, the x86-64 assembly language method of adding two values stores the result in the same register as one of the values it uses as an operand. Also, each instruction in the x86-64 assembly code sample above is proceeded by a `Q`. This denotes that the size of the operands for each instruction is up to 64-bits each. Different letters denote different maximal operand sizes.

Before proceeding to discuss our optimizations, we wish to highlight a few important features of ARMv7 and x86-64 computer architectures. Our ARM assembly code modifications are designed to work with ARMv7 instruction set with a 32-bit word size. The *word size* refers to the register size. The ARMv7 instruction set has 16 registers `r0,...,r15`. The registers `r13`, `r14` and `r15` are referred to as the stack pointer, link register and program counter respectively. They have each have a specific purpose and should not used for general-purpose calculations by the user. However, registers `r0` to `r12` can be used freely by the user.

Similarly, our x86-64 assembly code modifications are designed to work with the x86-64 architecture. This architecture offers sixteen 64-bit registers `r8,...,r15, rax, rcx, rdx, rbx, rsp, rbp, rsi` and `rdi`. The registers `rbp` and `rsp` are called the base pointer and stack pointer respectively. They should not be used for general-purpose calculations by the user. Comparing architectures, we see that ARMv7 and x86-64 both offer roughly the same amount of registers we can use for our computations, but x86-64 offers registers that have double the number of bits. This means that to store a given multi-precision unsigned integer, we require about half the number of registers on the x86-64 platform than we do on the ARMv7 platform.

Much more functionality can be obtained from assembly code than the above examples suggest. One can perform the other basic arithmetic functions, subtraction, multiplication and division. One can also introduce control structures that change control flow. Bit-shifting, logical and bitwise logical operators are also implemented as instructions. Below, we highlight several important features that we use in our optimizations.

Address Offsetting

Both x86-64 and ARMv7 assembly languages provide address offsetting features. These features allow you to refer to a specific data-storage location specified by a register with a specific offset. This is best illustrated by example.

```
LDR r12, [r1, #4]
```

The above example in ARMv7 assembly language loads the value in register `r1` offset by $4 * 8 = 32$ bits into register `r12`. If a 64-bit number had been loaded into `r1`, then the above instruction would cause `r12` to contain the right-most 32 bits.

```
MOVQ 16(%rax), %eax
```

In the example of x86-64 assembly language above, the value in register `rax` offset by $16 * 8 = 128$ bits is loaded into register `eax`.

Address offsetting can also be used when using registers as parameters in most other instructions.

Program Status Register

Both ARMv7 and x86-64 processors maintain a *program status register*. This is an area of memory on the processor that contains information about the state of the program being executed as well as the most recent operations performed. For example, both ARMv7 and x86-64 processors keep track of whether the most recent arithmetic operation caused an *overflow*. An overflow occurs if the result of an arithmetic operation is too large to be held in a register. For example, if the instruction `ADD r6, r6, r9` is executed and the resulting value was bigger than the size of `r6`, then the *carry flag* in the program status register would be set to 1. On the ARMv7 architecture, another way to set the carry flag is if the result of a subtraction is positive or zero. Similarly, x86-64 processors have a borrow flag. Suppose we are performing a subtraction `SUB %rax, %eax`. This x86-64 instruction subtracts the value contained in `eax` from the value contained in `rax`, storing the result to `rax`. Suppose the value in `eax` is greater than the value in `rax`. Then the borrow flag is set during the execution of the next instruction only. It is then reset to zero unless changed by a subsequent instruction. This flag is used to determine the output of the *subtraction with borrow* instruction `SBB`. Assume `r8` holds the number a and `r9` holds the value b. Then `SBB r8, r9` stores the value a-b-B to `r8` where B is the value of the borrow bit. As we will see later, use of the program status register is crucial to implementing multi-precision arithmetic.

Arithmetic with Accumulator

ARM assembly language conveniently features arithmetic instructions that *accumulate*. For example, the `UMLAL` instruction is the *unsigned multiplication with accumulate* instruction. Its syntax is as follows:

```
UMLAL RdLo, RdHi, Rm, Rs
```

Like the normal unsigned multiplication instruction (`UMULL`), the values in registers `Rm` and `Rs` are multiplied together. However, instead of simply putting the highest word of

the result in the register `RdHi` and the lowest word in register `RdLo`, the highest word of the result is *added* to the value in `RdHi` and the lowest word is *added* to the value in `RdLo`. This feature of ARM assembly language makes implementing multi-precision integer multiplication very convenient.

4.2 Optimizations to the Key-Exchange

In this section we describe the optimizations we made to the key-exchange protocol described in [17, 26]. We first describe optimizations made to the C/Cython portion of the program and then discuss our assembly optimizations.

4.2.1 Porting into C

The original implementation of the key-exchange (c.f. Section 2.5) has elliptic curves implemented in Cython. Cython is designed to provide an interface between C and Python code so that both can be used in the same software. This allows for the convenience of Python while still implementing critical portions in C. However, a pure C implementation is generally more efficient. We created a C-implementation of elliptic curves and re-wrote the key-exchange in pure C for the $\ell_A = 2$ and $\ell_B = 3$ cases.

In the original implementation, public parameters are generated immediately before the key-exchange is executed. This is impractical as these computations are done in Sage and are time-consuming. Furthermore, it is not necessary to have new public parameters for each run of the key-exchange. We separated parameter generation from the actual execution of the key-exchange. Parameter generation is done in Sage and the output is written to a text file. This allows parameter generation to be done on a different device than the key-exchange protocol. The pure C implementation of the key-exchange takes the text file as input.

4.2.2 Assembly Optimizations

We used assembly code to speedup multi-precision integer arithmetic in $\mathbb{F}_q = \mathbb{F}_{p^2}$. *Multi-precision* integers are integers that are larger than a given machine's word-size. The majority of programming languages do not provide direct support for multi-precision numbers. One generally needs to use a specific software package, such as GMP, in order to use them

efficiently. However, if one knows the approximate size of the number in advance, one can sometimes design hand-optimized assembly routines that are more efficient than routines in these software packages. The disadvantage of this approach is that hand-optimized assembly code is platform-specific and non-portable. We employ the following techniques to optimize our assembly implementation:

Loop Unrolling: Since we know the maximal size in bits of the operands, we can unroll all loops. This gives us the ability to avoid conditional branches.

Instruction re-ordering: Often times instructions can compete for the same processor and data resources, causing the code to be slower. By re-ordering non-dependent instructions we can increase the amount of instructions that the processor can execute at the same time. For example, for integer multiplication, loop unrolling makes it possible to load the data required for the next multiplication while the processor is performing the current one.

Register Allocation: All available registers were used in order to minimize moving data from memory to registers.

The key-exchange software uses GMP as an arithmetic backend. GMP stores numbers in consecutive memory locations and uses the *little endian* method which stores the least-significant word at the smallest memory address. We implemented field addition in ARMv7 assembly language and implemented both field addition and field multiplication in x86-64 assembly language. Field multiplication for $\mathbb{F}_q$ requires several additions, 3 integer multiplications and two modular reductions for which Barrett reduction was used. We present details of these implementations below. We will not attempt to explain each assembly instruction in detail but rather try to present the idea of each algorithm we discuss. For detailed instruction descriptions we refer the reader the ARM Reference Manual [4] and the Intel Software Developer's Manual [24].

Field Addition

512-bit $\mathbb{F}_q$ addition was implemented on ARMv7 platform and 768-bit $\mathbb{F}_q$ addition was implemented on the x86-64 platform.

When passing three or fewer parameters to a function, the function will place those parameters in registers `r1-r3` and expect to receive any possible output at the memory address in register `r0`. In Algorithm 4.1, we give a small-scale example of the technique

Algorithm 4.1 96-bit ARMv7 field addition.

```
1: LDM r1!, r3-r5
2: LDM r2!, r6-r8
3: ADDS r3,r6
4: ADCS r4,r7
5: ADCS r5,r8
6: STM r0!, r3-r5
7: STR r3, [r0],#-12
8: STR r3, [r13], #-12
9: LDM r0!, r3-r5
10: LDR r3, PRIME
11: SUBS r4, r3
12: LDR r3, PRIME+4
13: SUBS r5, r3
14: LDR r3, PRIME+8
15: SUBS r6, r3
16: STMCS r0!, r4-r6
```

we used to implement field addition on the ARMv7 platform. We assume we are adding two 96-bit integers instead of two 512-bit integers which is what the actual implementation does. The technique is the same although the latter version is quite a bit longer. We now explain Algorithm 4.1 in detail.

- Line 1 loads the first operand into registers `r3, r4`, and `r5` with the least significant word in `r3`. The `!` character increments the memory address after each load so that the value at `r1` is loaded into `r3`, the value at [address at `r1`] + 4 bytes is loaded into `r4` and the value at [address at `r1`] + 8 bytes is loaded into `r5`. Line 2 loads the second operand into registers `r6, r7`, and `r8` in the same way.

- Lines 3-5 add the first two operands together, storing the result in registers `r3`, `r4` and `r5`. Register `r3` contains the least-significant word, `r4` the next most-significant word and `r5` the most-significant word. `ADC` is the *addition with carry* instruction and the `S` at the end of the instruction indicates that the program status register should be updated based on addition.

- Line 6 stores the result of the addition (`r3`, `r4` and `r5`) to `r0`. The register `r0` now contains the least significant word of the result of adding the two operands together while `r0`+8 bytes contains the most significant word.

- After line 6, `r0` now points to the value at [address at `r0`] +12 bytes. The `STR` instruction is generally used to transfer data between different locations, but on Lines 7-8 it is only being used to force `r0` to point to the value at `r0` and `r12` to point to the value at [address at `r12`] − 12 bytes.

- The `LDR` instruction moves data from a memory location to a register location. On line 10, it is moving the value in `PRIME` to the register `r3`. `PRIME` points to the least-significant word of the order of the prime field $\mathbb{F}_q$. The next significant word is stored at `PRIME` + 4 bytes and the most significant word is stored at `PRIME` + 8 bytes. Lines 9-15 subtract the prime from the result of the addition of the two operands and store that result to `r13`. The register `r0` now contains the result of the (non-modular addition) and `r13` contains the result of the modular addition.

- We must now determine whether the subtraction was necessary. The instruction on line 16 is similar to that on line 6. However, the `CS` indicates that the instruction is only executed if the carry flag in the program status register is set. This flag will only be set if the value in `r3` is less than or equal the value in `r6` (see line 15) meaning that the subtraction was necessary.

When Algorithm 4.1 terminates, we see that the required output value is at the address in `r0`. Note that the `STM` and `LDM` instructions function much like the `STR` and `LDR` instructions. However, the former instructions are capable of moving multiple registers at the same time.

Our implementation of 768-bit field addition on the x86-64 platform is much simpler. It was found that is was just as efficient to use the built-in GMP function `mpz_add` to add the two numbers rather than using an assembly routine. This is likely due to `mpz_add` being coded in assembly for the ARMv7 platform. After adding the two numbers, we then check to see if a subtraction is required by using `mpz_cmp` to compare the size of the result to the prime order of the underlying field. If a subtraction is required, it is done using an assembly routine to subtract the prime from the result. This routine essentially consists of the `SUB` instruction followed by several `SBB` instructions to subtract the correct pieces of the prime from the result and adjust for any borrow flags.

Integer Multiplication

768-bit field multiplication was implemented on the x86-64 platform. Like for field addition, we demonstrate our technique with a minimal example. We assume we are multiplying two 128-bit integers instead of two 768-bit integers. The technique we use is known as *column-wise multiplication* and is analogous to the method school-book multiplication taught to

Algorithm 4.2 128-bit x86-64 integer multiplication.

```
 1: MOVQ 0(%r8), %rax %%A0*B0
 2: MULQ 0(%r9)
 3: MOVQ %rax, 0(%r14)
 4: MOVQ %rdx, %r10
 5:
 6: MOVQ 8(%r8), %rax %%A1*B0
 7: MULQ 0(%r9)
 8: ADDQ %rax, %r10
 9: MOVQ %rdx, %r11
10: ADCQ $0, %r11
11:
12: XORQ %r12, %r12
13:
14: MOVQ 0(%r8), %rax %%A0*B1
15: MULQ 8(%r9)
16: ADDQ %rax, %r10
17: MOVQ %r10, 8(%r14)
18: ADCQ %rdx, %r11
19:
20: MOVQ 8(%r8), %rax %%A1*B1
21: MULQ 8(%r9)
22: ADDQ %rax, %r11
23: ADCQ %rdx, %r12
24: ADCQ $0, %r12
25:
26: MOVQ %r11, 16 (%r14)
27: MOVQ %r12, 24(%r14)
```

young children. Before discussing Algorithm 4.2, we must highlight three important characteristics specific to this assembly language. First, the `MUL` instruction takes one register operand and multiplies the value of that operand by the value of the number in register `rax`. It stores the double-word result of this multiplication in the registers `rax` (lowest word) and `rdx` (highest word). Second, register labels are preceded by the `%` character. Third the `MOV` instruction moves data between different data-storage locations. `MOV` is much like ARMv7's `LDR` and `STR` instructions.

We proceed to discuss details of the algorithm. 128-bit integer multiplication corresponds to multiplying two numbers made up of two words each. We assume the two numbers we wish to multiply together are in registers `r8` and `r9` and that we are placing the output at the address in register `r14`. Each block of code, other than the last, corresponds to multiplying one word in one number by one word in the other, while the `XOR`

(bitwise exclusive OR) instruction is used as an efficient way to clear the contents of a register. For ease of explanation, let the two numbers in `r8` and `r9` be A and B respectively. Let the least significant word of A be $A0$ and the most-significant be $A1$. Then $A0$ is stored at the address at `r8` and $A1$ at (address at `r8`) + 8 bytes. We similarly label the words of B as $B0$ and $B1$. The comments (preceded by `%%`) beside each block of code corresponds to the two words being multiplied together.

- The first block of code multiplies $A0$ and $B0$. The lowest word in the result of this multiplication will be the lowest word in the overall result, thus line 3 updates the output appropriately.
- The second block of code multiplies $A1$ and $B0$. It adds the lowest word of the result of this multiplication to the register `r10` which previously contained the highest word result of the multiplication in the first block. `ADC` is the *addition-with-carry* instruction. Line 10 adds the carry bit (if set) to the register `r11`. The carry bit being set would indicate that there was an overflow when `rax` was added to `r10` in line 8. The `ADC` instruction in line 10 ensures that this overflow is accounted for in the overall result.
- The third block of code multiplies $A0$ and $B1$. It adds the lowest word of the result of this multiplication to the register `r10` which now contains the second-lowest word of the result of the overall multiplication. Line 17 then updates the output accordingly.
- The fourth block of code multiples $A1$ and $B1$. The lowest (highest) word in the result of this multiplication will be the second-highest (highest) word in the overall result. Lines 26 and 27 updates the output accordingly.

Barrett Reduction

768-bit Barrett reduction was implemented on the x86-64 platform for use in $\mathbb{F}_q$ multiplication. The algorithm is implemented using a combination of C and assembly with the aspects that control the flow of the algorithm implemented in C and the arithmetic portions implemented in assembly. We first give a high-level overview of the algorithm and then discuss the details of our use of assembly.

In 1986, P. D. Barrett introduced Barrett reduction to compute $c = a \pmod{n}$ [8]. Barrett is an improvement on naive division algorithms and works assuming that $a < n^2$. The main idea is to replace expensive divisions by multiplications that can be performed much cheaper.

Algorithm 4.3 Barrett reduction for computing $c = a \pmod{p}$ [8].

Require: $a < p^2$, k minimal such that $2^k > p$ and $m = \lfloor 4^k/p \rfloor$
Ensure: $c = a \pmod{n}$
1: $n = m * a$
2: $q = \lfloor n/4^k \rfloor$
3: $r = a - qn$
4: **if** $r < p$ **then**
5: $\quad c = r$
6: **else**
7: $\quad c = r - p$
8: **end if**

We use Barrett reduction to compute $c = a \pmod{p}$ where $\mathbb{F}_q = \mathbb{F}_{p^2}$. We are able to make several pre-computations at the parameter generation stage before the key-exchange is executed and stored in the same text file containing the public parameters for the encryption system. First we compute the minimal k such that $2^k > p$ and then m such that $m = \lfloor 4^k/p \rfloor$. Barrett reduction is outlined in Algorithm 4.3. Functions coded in assembly language perform the computations for lines 1-3, 5 and 7. The assembly code for lines 1, 3, 5 and 7 are straightforward given what we have already discussed in this section. The only feature of the algorithm we have not discussed is how to efficiently compute line 2 using assembly language. Since we are dividing by a power of 2, we need not actually do any division. We simply need to determine which bits of n we wish to keep and define q accordingly. Since we are dividing by 4^k, this amounts to removing the least-significant $2k$ bits.

4.2.3 Results

The timing results for our pure C implementation are presented in Figure 4.1 below. Notice that our pure C implementation is approximately 20 percent faster than the original implementation on the Mac OS platform. Due to the variety of software packages required, the original implementation was not suitable to run on the iOS or Android platform. Thus, a comparison of running times cannot be made on these platforms, even though our implementation supports them. Let p_{512}, p_{768} and p_{1024} denote the 512-bit, 768-bit and 1024-bit primes (respectively) that we use to compute running times. They are defined as follows:

$$p_{512} = 186 \cdot (2^{258}3^{161}) - 1,$$
$$p_{768} = 2 \cdot (2^{386}3^{242}) - 1,$$
$$p_{1024} = 353 \cdot (2^{514}3^{323}) - 1.$$

Implementing field addition in ARM assembly gave a speedup of 1.5% on the Android

Prime	p_{512}	p_{768}	p_{1024}
Quantum Security	**85 bits**	**128 bits**	**170 bits**
Original (Mac OS[1])	0.113 s	0.303 s	0.529 s
Pure C (Mac OS)[1]	0.093 s	0.226 s	0.429 s
Pure C (iOS)[2]	1.06 s	2.68 s	5.30 s
Pure C (Android)[3]	0.629 s	1.77 s	3.81 s
C with ARM assembly field addition (Android)[3]	0.620 s		
C with x86-64 assembly field addition/multiplication (Mac OS)[1]		0.217 s	

[1]Macbook Pro Intel Core i5 @ 2.4 GHz, [2]Ipad 2 ARM Cortex-A9 @ 1 GHz dual-core, [3]Arndale Board ARM Cortex-A15 @ 1.7 GHz dual-core

Figure 4.1: Timings for our C implementation of the key exchange for $\ell_A = 2$ and $\ell_B = 3$

platform for 512-bit values of p. Implementing both field addition and multiplication in x86-64 assembly gave a speedup of 4% on the Mac OS X platform for 768-bit values of p. These improvements are relative to the times in Figure 4.1.

4.3 Optimization of the Pairing Computation

In [22], the authors implement integer multiplication in ARM assembly language to support field arithmetic for the 254-bit pairing computation. We use their technique, which is actually the same technique used to implement integer multiplication in Section 4.2.2, to implement integer multiplication to support field arithmetic for the 446-bit pairing computation.

4.3.1 Integer Multiplication

We illustrate our technique for integer multiplication with a minimal example that multiplies two 64-bit integers. The actual implementation multiplies two 446-bit integers. Since the word size on ARMv7 is 32-bit, this means the numbers will each be two words in length. As before, denote the numbers A and B with $A0$ being the least-significant word and $A1$ being the most significant word. Similarly label the words in B as $B0$ and $B1$.

Algorithm 4.4 64-bit ARMv7 integer multiplication.

```
 1: LDR r12, [r1]
 2: LDR r14, [r2]
 3: MOV r5, #0
 4: UMULL r3, r4, r12, r14 %%A0*B0
 5: MOV r11, #0
 6: MOV r6, #0
 7:
 8: LDR r14, [r2, #4]
 9: MOV r10, #0
10: UMLAL r4, r5, r12, r14 %%A0*B1
11: MOV r7, #0
12:
13: LDR r12, [r1, #4]
14: LDR r14, [r2]
15: MOV r9, #0
16: UMLAL r4, r11, r12, r14 %%A1*B0
17: MOV r8, #0
18:
19: LDR r14, [r2, #4]
20: ADDS r5, r5, r11
21: ADC r6, r6, #0
22: UMLAL r5, r10, r12, r14 %%A1*B1
23:
24: ADD r6, r6, r10
25: STMIA r0!, {r3-r6}
```

Assume `r1` contains A and `r2` contains B. After the algorithm is executed, `r0` contains the output. As previously mentioned, the main idea of Algorithm 4.4 is the same as Algorithm 4.2. Both use column-wise multiplication. However, these algorithms differ slightly in details since both languages have different instruction sets. The `UMLAL` instruction was discussed in Section 4.1. The `UMULL` instruction is similar, however, the result of the multiplication is not added to the destination operands, but it replaces any other values in there. Recall that the `ADDS` and `ADC` instructions are addition and addition-with-carry respectively. The presence of the multiplication with accumulate instruction, as well as the separation of source operands from destination operands for both `UMULL` and `UMLAL` instructions in ARMv7 assembly language, are in stark contrast to the situation in x86-64 assembly language where the source/destination operands overlap. This difference permits more-efficient register use in the ARMv7 assembly version.

Given what we have discussed, the code in Algorithm 4.4 should be fairly straightforward. Each block corresponds to multiplication of one word A by one word in B. The only

instruction that we have not discussed is the `MOV` instruction which has similar behaviour to its counter-part in the x86-84 assembly language. For example, line 5 moves the value of 0 into register `r11`. The final instruction on line 25 is a variant of the `STM` instruction and moves the results of the single-word multiplications into the appropriate locations in the output.

4.3.2 Results

We executed our code on the Arndale Board ARM Cortex-A15 @ 1.7 GHz dual-core platform. Table 4.1 compares the results of our optimization versus the original implementation described in [22]. Our assembly optimization give a 7-8% speed improvement in the computation of both the affine and projective 446-bit Optimal Ate pairing.

Table 4.1: Timings for affine and projective pairings on the Arndale Board (ARM v7) Cortex-A15 at 1.7 GHz. Times for the Miller loop (ML) in each row reflect those of the faster pairing.

Field Size	Lang.	Operation Timing [μs]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	ASM	0.14	0.80	0.43	10.87	13.59	0.23	2.04	1.60	13.00	5,014	3,215	8,678	8,424
	C	0.08	0.86	0.43	10.99	12.78	0.19	2.33	1.83	13.34	5,765	3,545	9,549	9,655
446-bit	ASM	0.13	1.69	0.98	21.32	12.61	0.26	4.74	3.78	26.97	10,875	10,171	29,545	29,792
	C	0.12	1.78	0.96	21.19	11.90	0.23	5.05	4.02	27.52	11,714	10,939	31,828	32,320
638-bit		0.16	3.58	1.78	34.28	9.58	0.27	9.57	7.73	46.57	20,237	26,003	77,388	78,691

Chapter 5

Conclusion

We find that the key-exchange protocol of [17, 26] can be realistically implemented and used on mobile communication devices at reasonable security levels. The protocol represents an attractive option for those seeking practical and quantum-resistant cryptographic primitives for post-quantum cryptography. Furthermore, the same low-level assembly optimization techniques used on the key-exchange protocol can be used to improve performance in other systems that rely on efficient field arithmetic, such as the computation of the O-Ate pairing over BN curves presented in [21, 22].

There are a variety of other avenues for further research. Firstly, the assembly modifications to the key-exchange could be implemented in a better way. They are primarily done using inline assembly which allows assembly code to be written in the same file as regular C code. Although not readily apparent, this method is prone to bugs and inherent inefficiencies. A better approach would be to write separate files containing the assembly code and link them to the C program. This approach was used for the modifications to the pairing computation. Such an approach would likely lead to more efficient code in the case of the key-exchange software.

A second avenue for further research is incorporating the use of Single-Instruction-Multiple-Data (SIMD) techniques into the assembly optimizations. Such techniques have been shown to considerably improve performance of field multiplication in related applications [38]. One feature of SIMD instructions is to allow multiple multiplication instructions to be executed at the same time with virtually the same cost as a single multiplication instruction. If applied in an intelligent way, these techniques could likely improve performance in both the key-exchange and pairing computation.

References

[1] IEEE P 1363.3: Standard for identity-based cryptographic techniques using pairings. draft 3:section 5.3.2. http://grouper.ieee.org/groups/1363/IBC/index.html.

[2] Tolga Acar, Kristin Lauter, Michael Naehrig, and Daniel Shumow. Affine pairings on ARM. Cryptology ePrint Archive, Report 2011/243, 2011. http://eprint.iacr.org/.

[3] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. In *Advances in Cryptology–EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 48–68, 2011.

[4] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. [Online; accessed 25-Feb-2014].

[5] R Balasubramanian and Neal Koblitz. The improbability that an elliptic curve has subexponential discrete log problem under the Menezes, Okamoto, Vanstone algorithm. *Journal of Cryptology*, 11(2):141–145, 1998.

[6] Paulo S. L. M. Barreto, Hae Y. Kim, Ben Lynn, and Michael Scott. Efficient algorithms for pairing-based cryptosystems. In *Advances in Cryptology–CRYPTO 2002*, volume 2442 of *LNCS*, pages 354–369, 2002.

[7] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography 2005*, volume 3897 of *LNCS*, pages 319–331, 2005.

[8] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology–CRYPTO' 86*, volume 263 of *LNCS*, pages 311–323, 1987.

[9] Naomi Benger and Michael Scott. Constructing tower extensions of finite fields for implementation of pairing-based cryptography. In *Proceedings of the Third International Conference on Arithmetic of Finite Fields*, volume 6087 of *LNCS*, pages 180–195, 2010.

[10] Daniel Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In *Progress in Cryptology–AFRICACRYPT 2008*, volume 5032 of *LNCS*, pages 389–405, 2008.

[11] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database, 2007. http://www.hyperelliptic.org/EFD/index.html.

[12] Jean-Luc Beuchat, Jorge E González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In *Pairing-Based Cryptography–Pairing 2010*, volume 6487 of *LNCS*, pages 21–39, 2010.

[13] I. F. Blake, G. Seroussi, and N. P. Smart. *Advances in Elliptic Curve Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2005.

[14] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *Advances in Cryptology–CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55, 2004.

[15] Reinier Bröker. Constructing supersingular elliptic curves. *Journal of Combinatorics and Number Theory*, 1(3):269–273, 2009.

[16] Jean-Marc Couveignes. Hard homogeneous spaces. Cryptology ePrint Archive, Report 2006/291, 2006. http://eprint.iacr.org/.

[17] Luca De Feo, David Jao, and Jerome Plut. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. Cryptology ePrint Archive, Report 2011/506, 2011. http://eprint.iacr.org/.

[18] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010.

[19] Laura Fuentes-Castañeda, Edward Knapp, and Francisco Rodríguez-Henríquez. Faster hashing to G_2. In *Selected Areas in Cryptography 2012*, volume 7118 of *LNCS*, pages 412–430, 2012.

[20] C. C. F. Pereira Geovandro, Marcos A. Simplício Jr., Michael Naehrig, and Paulo S. L. M. Barreto. A family of implementation-friendly BN elliptic curves. *Journal of Systems and Software*, 84(8):1319–1326, 2011.

[21] Gurleen Grewal. Efficient Pairings on Various Platforms. Master's thesis, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2012.

[22] Gurleen Grewal, Reza Azarderakhsh, Patrick Longa, Shi Hu, and David Jao. Efficient implementation of bilinear pairings on ARM processors. In *Selected Areas in Cryptography 2013*, volume 7707 of *LNCS*, pages 149–165, 2013.

[23] Florian Hess. Pairing lattices. In *Pairing-Based Cryptography–Pairing 2008*, volume 5209 of *LNCS*, pages 18–38, 2008.

[24] Intel. *Intel 64 and IA-31 Architectures Software Developer's Manual*. [Online; accessed 26-Feb-2014].

[25] Tadashi Iyama, Shinsaku Kiyomoto, Kazuhide Fukushima, Toshiaki Tanaka, and Tsuyoshi Takagi. Efficient implementation of pairing on BREW mobile phones. In *Advances in Information and Computer Security–IWSEC 2010*, volume 6434 of *LNCS*, pages 326–336, 2010.

[26] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-Quantum Cryptography–PQCrypto 2011*, volume 7071 of *LNCS*, pages 19–34, 2011.

[27] Antoine Joux. A one round protocol for tripartite Diffie-Hellman. In *Proceedings of the 8th international conference on algorithmic number theory–ANTS-VIII*, volume 1838 of *LNCS*, pages 385–393, 2000.

[28] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology–CRYPTO' 99*, volume 1666 of *LNCS*, pages 388–397, 1999.

[29] J. Lagarias and A. Odlyzko. Effective versions of the Chebotarev density theorem. In *Algebraic number fields: L-functions and Galois properties*, Symposium Proceedings of the University of Durham, pages 409–464, 1975.

[30] Florian Luca, David Mireles Morales, and Igor Shparlinski. MOV attack in various subgroups on elliptic curves. *Illinois Journal of Mathematics*, 48(3):1041–1052, 2004.

[31] Alfred Menezes. An introduction to pairing-based cryptography. 1991. http://cacr.uwaterloo.ca/~ajmeneze/publications/pairings.pdf.

[32] Alfred Menezes, Wu, Yi-Hong, and Robert Zuchherato. *An elementary introduction to hyperelliptic curves*. Appendix in *Algebraic Aspects of Cryptography*. Springer, 1998.

[33] Victor S. Miller. The Weil pairing, and its efficient calculation. *Journal of Cryptology*, 17(4):235–261, 2004.

[34] Hermann Minkowski. *Geometrie der Zahlen*. B.G. Teubner, Leipzig, 1910.

[35] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[36] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *Progress in Cryptology–LATINCRYPT 2010*, volume 6212 of *LNCS*, pages 109–123, 2010.

[37] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *Advances in Cryptology–ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 390–405, 2008.

[38] Ana Helena Sánchez and Francisco Rodríguez-Henríquez. NEON implementation of an attribute-based encryption scheme. In *Applied Cryptography and Network Security 2013*, volume 7954 of *LNCS*, pages 322–338, 2013.

[39] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *GTM*. Springer, New York, 1992.

[40] John Tate. Endomorphisms of abelian varieties over finite fields. *Inventiones Mathematicae*, 2:134–144, 1966.

[41] Jacques Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences Paris Séries A-B*, 273:A238–A241, 1971.

[42] Frederik Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1):455–461, 2010.