Elliptic Curve Cryptography
Final Report - Project in computer security

Itay Khazon Eyal Tolchinsky Supervisor: Barukh Ziv

October 19, 2014

Contents

1 Introduction .......................................................... 2
2 Mathematical Background .......................................... 2
  2.1 Basics of Elliptic Curves ...................................... 3
  2.2 Group structure of Elliptic Curves ............................ 4
  2.3 Projective Coordinates ....................................... 5
3 Security ................................................................. 6
  3.1 Elliptic Curve Discrete Logarithm Problem .................... 6
  3.2 Cryptographically Strong Elliptic Curves .................... 6
  3.3 Elliptic Curve Diffie-Hellman ................................. 7
  3.4 Point Counting ................................................. 7
4 Implementation ....................................................... 8
  4.1 General Overview ............................................. 8
  4.2 Documentation ................................................ 9
  4.3 Examples ...................................................... 10
  4.4 Montgomery Point Multiplication .............................. 11
  4.5 Point Counting Algorithm .................................... 13
5 Challenge ............................................................. 17
6 Performance .......................................................... 18
7 Conclusion ............................................................ 19
  7.1 What We Have Learnt ......................................... 19
  7.2 Future Goals .................................................. 19
8 Bibliography .......................................................... 19
1 Introduction

Elliptic curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields (usually, \(GF(p)\) and \(GF(2^m)\)).

The use of elliptic curves in public-key cryptography stems from the fact that by defining an addition law, one can make a curve (or the points on it) into an abelian group.

By providing a group structure (in fact, by finding a point that is a generator for a subgroup of a large prime order), one can easily use existing public-key algorithms (such as Diffie-Hellman key exchange, and ElGamal encryption) on the curve’s subgroup, instead of the regular group - \(\mathbb{Z}_p\).

Public-key Cryptography was first published in the late 70’s, with famous algorithms such as RSA and DH. What made public-key cryptography so special was that it’s algorithms are based on mathematical problems which currently admit no efficient solution - especially integer factorization and discrete logarithm problems.

Unfortunately, algorithms like General Number Field Sieve and Quadratic Sieve can be used to solve factorization - and hence classic public-key algorithms - in sub-exponential time.

On the other hand, there are no known sub-exponential algorithms for solving elliptic-curve based cryptography algorithms. For example, solving ECDLP (elliptic curve discrete logarithm problem) using any known algorithm takes at least \(O(\sqrt{n})\) time, meaning that if the field size is \(2^n\) then the algorithm would take \(O(2^n)\). These attributes mean that smaller key sizes can be used in the ECC case for the same level of security\(^1\).

<table>
<thead>
<tr>
<th>Security level (bits)</th>
<th>DL parameter (q)</th>
<th>EC parameter (n)</th>
<th>RSA modulus (n)</th>
<th>DL modulus (p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>160</td>
<td>1024</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>224</td>
<td>2048</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>3072</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>384</td>
<td>8192</td>
<td>3072</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>512</td>
<td>15360</td>
<td>8192</td>
<td></td>
</tr>
</tbody>
</table>

Our goal in the project has been to implement an algorithm to efficiently generate cryptographically strong elliptic curves. Also, for that reason, we were asked to implement a general purpose elliptic curve arithmetic library. To test our algorithms, we were also asked to implement a simple EC based algorithm (e.g. ECDH).

2 Mathematical Background

As mentioned before, elliptic curves can be defined over various fields. For our project, we chose (with our supervisor) to focus on non-supersingular elliptic curves over binary fields - \(GF(2^m)\).

\(^1\)Table from Guide to Elliptic Curve Cryptography.
The reasons for this choice are that point counting methods are faster on binary fields. Also, we did not have any background in binary fields and it sounded interesting and challenging to try. Therefore, the following section will mainly describe the mathematical background of non-supersingular elliptic curves over binary fields.

2.1 Basics of Elliptic Curves

Definition 1 (Elliptic Curve): An elliptic curve $E$ over a field $K$ is defined by an equation

$$E : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and $\Delta \neq 0$, where $\Delta$ is the discriminant of $E$ and is defined as follows:

$$\Delta = -d_2^2 d_6 - 8 d_4^3 - 27 d_6^2 + 9 d_2 d_4 d_6$$

$$d_2 = a_1^2 + 4 a_2$$

$$d_4 = 2 a_4 + a_1 a_3$$

$$d_6 = a_3^2 + 4 a_6$$

$$d_8 = a_1^2 a_6 + 4 a_2 a_6 - a_1 a_3 a_4 + a_2 a_3 - a_4^2$$

If the field characteristic is 2 and $a_1 \neq 0$, then by using admissible change of variables, one can find a isomorphic elliptic curve of the form

$$y^2 + xy = x^3 + ax + b$$

Another advantage is that ECC algorithms over binary fields are easier and faster to implement in hardware (Additions are just XORs).
where \( a, b \in K \). Such a curve is said to be non-supersingular and has \( \Delta = b \).

Therefore, for our purposes, a non-supersingular elliptic curve over a binary field is an equation of the form \( y^2 + xy = x^3 + ax + b \). Furthermore, we fix \( a = 0 \) to simplify the equation even more

\[
y^2 + xy = x^3 + b
\]

The reasons behind the decision to fix \( a = 0 \) were that it doesn’t affect security, it increases performance and it was necessary for our scalar multiplication algorithm - montgomery’s point multiplication.

### 2.2 Group structure of Elliptic Curves

**Definition 2 (EC group structure):** Let \( E \) be an EC. The points of \( E \) are all the points \( P = (x, y) \in GF(2^p)^2 \) which satisfy the curve equation. Also, there is a special “Point at infinity”, marked by \( \mathcal{O} \). \( \mathcal{O} \) serves as the identity element.

The group operation is defined as followed:

- For \( \mathcal{O} \) and any point on \( E \), \( \mathcal{O} + P = P + \mathcal{O} = P \).
- For any point \( P = (x, y) \) on \( E \), \( -P = (x, x + y) \). For \( \mathcal{O}, -\mathcal{O} = \mathcal{O} \).
- For any 3 colinear points on the curve \( P, Q, R \) we have \( P + Q + R = \mathcal{O} \), and therefore \( P + Q = -R \).

The addition rule is: for any 2 points \( P, Q \) on the curve, “draw the line” between them and find the 3rd point on the line \( R \) (if the line never intersects the curve again we use \( R = \mathcal{O} \)), then \( P + Q = -R \). Also, in the case that \( Q = P \) we can’t “draw a line” between them, so we use the line tangent to \( P \). From the attributes of the curve and its “smoothness” (ensured by \( \Delta \neq 0 \)) we get that this addition rule is well defined.

The algebraic formula for adding two points is as shown:

- If \( P = (x_1, y_1), Q = (x_2, y_2) \) and \( P \neq \pm Q \) then \( P + Q = (x_3, y_3) \) where

\[
x_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + b \quad y = x_1^2 + \left(\frac{y_1}{x_1}\right)x_3 + x_3
\]

- If \( P = Q \) then \( P + Q = 2P = (x_3, y_3) \) where

\[
x_3 = x_1^2 + \frac{b}{x_1} \quad y = x_1^2 + \left(\frac{y_1}{x_1}\right)x_3 + x_3
\]

- If \( P = -Q \) then \( P + Q = \mathcal{O} \).

From the addition rule we can naturally define scalar multiplication: If \( k \in \mathbb{Z} \) and \( P \) is a point then \( kP = \underbrace{P + P + \ldots + P}_{k \text{ times}} \).
Example: for $GF(2^4)$ represented as $GF(2)[\alpha]/(\alpha^4 + \alpha + 1)$ with the curve $E : y^2 + xy = x^3 + \alpha^3 x^2 + (\alpha^3 + 1)$ has 22 points:

$\infty$ (0011, 1100) (1000, 0001) (1100, 0000)
(0000, 1011) (0111, 1111) (1000, 1001) (1100, 1100)
(0001, 0000) (0101, 0000) (1001, 0110) (1111, 0100)
(0001, 0001) (0101, 0101) (1001, 1111) (1111, 1011)
(0010, 1101) (0111, 1011) (1011, 0010) (1111, 1010)
(0010, 1111) (0111, 1100) (1100, 1101)

examples of addition are (0010, 1111) + (1100, 1100) = (0001, 0001) and 2(0010, 1111) = (1011, 0010).
The point $P = (1000, 0001)$ has order 11, and its multiples are:

$\begin{array}{llll}
0P = \infty & 3P = (1100, 0000) & 6P = (1011, 1001) & 9P = (1011, 0110) \\
1P = (1000, 0001) & 4P = (1111, 1011) & 7P = (1111, 0100) & 10P = (1000, 1001) \\
2P = (1001, 1111) & 5P = (1011, 0010) & 8P = (1100, 1100) \\
\end{array}$

2.3 Projective Coordinates

The addition of points requires inversions over $GF(2^p)$. Inversions are very costly in most implementations, and therefore reduce performance a great deal (in comparison to additions, squaring and multiplications).

Projective coordinates allow us to do computations in the projective plane and delay inversions until needed, at the cost of more multiplications. Resulting in better performance.

Definition 3 (Projective Plane): The projective plane is the set $K^3 \setminus \{(0,0,0)\}$ with an equivalence relation characterized by two positive integers $c,d$ and defined by $(X_1,Y_1,Z_1) \sim (X_2,Y_2,Z_2)$ if $X_1 = \lambda^c X_2$, $Y_1 = \lambda^d Y_2$, $Z_1 = \lambda Z_2$ for some $\lambda \in K$.

Intuitively, one can imagine the projective coordinate system as one in which every two-dimensional point is represented by a ray in the three-dimensional space (that is indeed the case when $c = d = 1$).

Two dimensional points $(X,Y)$ can be represented with the projective point $(X,Y,1)$ and therefore every point $(X,Y,Z)$ with $Z \neq 0$ represents some point $(X',Y')$ in the regular plane. Projective points $(X,Y,0)$ are called “points at infinity”, and we use them to represent the point at infinity $O$.

For our project we chose to implement Lopez-Dahab’s projective coordinates since they provide the best performance results over binary fields and it works well with our choice of scalar multiplication algorithm.

Lopez-Dahab (LD) coordinates are characterized by $c = 1, d = 2$. The curve equation under LD coordinates is of this form:

$$Y + XYZ = X^3Z + aX^2Z^2 + bZ^4$$
The following table compares the amounts of multiplications, squaring, additions and inversions needed for both addition and doubling algorithms (and shows the value of using LD coordinates):

<table>
<thead>
<tr>
<th></th>
<th>Multiplications</th>
<th>Squarings</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projective</td>
<td>13</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Affine</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Doubling</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Multiplications</td>
<td>Squarings</td>
<td>Additions</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

3 Security

In this section we briefly present the value of elliptic curves regarding computer security. We will present the ECDLP and its importance to ECC, and present the reason for counting the points of the curve - the goal of our project.

3.1 Elliptic Curve Discrete Logarithm Problem

The security of Diffie-Hellman and ElGamal relies on the hardness of the discrete logarithm, which is the analogue of the ordinary logarithm, but in a finite cyclic group.

The discrete logarithm problem (DLP) is: Given a cyclic group $\mathbb{Z}_p$, a generator $g$ and a group element $a$, find a natural number $m$ such that $g^m = a$. The Elliptic curve discrete logarithm problem is similar: given a point $G$ and a point $P$ such that $P = kG$ for some $k$, find a natural number $m$ such that $mG = P$.

As stated in the introduction, there is currently no known sub-exponential time algorithm to solve ECDLP (meaning, if the subgroup generated by $G$ is of size $n$, then any known algorithm will work in time $O(n^\alpha)$ - as far as we know, $\alpha = \frac{1}{2}$ is the best result so far). On the other hand, algorithms like index calculus can solve DLP over $\mathbb{Z}_p$ in randomized sub-exponential time.

3.2 Cryptographically Strong Elliptic Curves

Since the time complexity for solving ECDLP is highly dependent on the order of the generator $G$, its important that the curve $E$ has a subgroup of a large prime order.

In fact, we demand that the order of the group $\#E(\mathbb{F}(2^m)) = n \cdot p$ where $n$ is a small natural number and $p$ is a large prime. If so, we say that $E$ is a cryptographically strong curve and it is safe to use as the basis for cryptographic algorithms. Once we have a strong curve we can easily generate a random point $P$ and with high probability its order will be divisible by $p$ (meaning its order is $p \cdot n'$ where $n'$ is a number dividing $n$). if so, then the point $G = nP$ will have an order $p$ (because $n$ doesn’t divide $p$).
3.3 Elliptic Curve Diffie-Hellman

Once we found a generator $G$ of a large prime order $p$, we can use it for cryptographic algorithms. In this project we chose to implement ECDH as the algorithm to test or EC library.

ECDH is an elliptic curve variant of the famous Diffie-Hellman key exchange algorithm. The algorithm allows Alice and Bob to agree on a shared symmetric key using their asymmetric. It relies on these pre-conditions:

- Alice and Bob must agree on an elliptic curve $E$ and a generator $G$ of prime order $p$.
- Alice and Bob must each have private keys - numbers in the range $[1, ..., p - 1]$ (for this section, Alice's and Bob's keys are $d_A, d_B$).

Alice sends Bob her public key $d_A G$, and Bob sends Alice his public key $d_B G$. Then Alice computes $d_A (d_B G)$ and Bob computes $d_B (d_A G)$. Its easy to see that they now have the same shared secret $d_A d_B G = d_B d_A G$. This point can now we hashed and used a symmetric key for other symmetric cryptographic algorithms like AES.

3.4 Point Counting

To generate strong elliptic curves one has to know the order of the curves generated (to distinguish strong curves from weak curves). For that purpose point counting algorithms are helpful. A general generation algorithm will be as such:

1. Pick random coefficients for the curve.
2. count the number of points on the curve.
3. if the curve order isn’t in the form needed ($\text{order} = n \cdot p$ where $n$ is a small natural number and $p$ is a large prime) return to 1.

For our point counting algorithm, we chose a memory efficient variant of Satoh’s algorithm, Vercauteren’s Algorithm, who extended Satoh’s algorithm to binary fields. If $E$ is a curve over $GF(2^n)$ then the algorithm requires $O(n^3 \log n \log \log n)$ operations and $O(n^3)$ memory (Satoh’s original algorithm required $O(n^3)$ memory because it saved all $n$ j-invariants during the run, while this algorithm saves only $2$ j-invariants at a time).

If $E$ is a curve we denote $\#E(GF(2^n))$ the number of points on the curve. $\#E(GF(2^n))$ satisfies $\#E(GF(2^n)) = 2^n + 1 - t$ where $t$ is the trace of Frobenius endomorphism:

$$F : E \rightarrow E \quad F(x, y) = (x^{2^n}, y^{2^n})$$

Hasse’s theorem states that $|t| \leq 2 \cdot 2^{n-1}$, and therefore it is enough for us to find $|t|$ since we can find its sign from Hasse’s theorem and calculate $\#E(GF(2^n))$ from it.
The point counting algorithm relies on lifting both the curve $E$ and the Frobenius endomorphism $F$ to the valuation ring $\mathcal{R}$ of a degree $n$. Since this lifting is done in a canonical way, the trace of the lifted Frobenius $\mathcal{F}$ equals the trace of Frobenius $t$. Unfortunately, the Frobenius endomorphism $F$ itself is difficult to lift because it is inseparable. Therefore one actually works with the dual of the Frobenius endomorphism $F$, called the Verschiebung $\hat{F}$.

This Verschiebung is separable if and only if $E$ is non-supersingular (which is the case in our project) and can be lifted explicitly by lifting its kernel. Meaning, if we can lift $\hat{F}$ to $\tilde{\mathcal{F}}$ and find its trace, we can find the trace of Frobenius $t$, and compute $\#E(\mathbb{G}F(2^n))$.

To lift $E$ and $\hat{F}$, we used a lift with nice properties called the canonical lift. In the non-supersingular elliptic curve case, The canonical lift satisfies that the reduction modulo 2 of the lifted curve $\mathcal{E}$ equals $E$. Also, the canonical lift doesn’t change the endomorphism ring $(\text{End}(E) \cong \text{End}(\mathcal{E}))$, and therefore the trace of the lifted Verschiebung $\mathcal{F}$ equals the trace of Frobenius.

From these properties we get that $\text{Tr}(F) = \text{Tr}(\mathcal{F}) = \text{Tr}(\tilde{\mathcal{F}})$, and since we lifted Verschiebung $\mathcal{F}$ can be decomposed into the composition of little isogenies $\hat{\Sigma}_i$, which cycle the curve $\mathcal{E}$ through its $n$ conjugates. Meaning:

$$t = \text{Tr}(F) = \text{Tr}(\tilde{\mathcal{F}}) = \text{Tr}(\hat{\Sigma}_{n-1} \circ \hat{\Sigma}_{n-2} \circ \ldots \circ \hat{\Sigma}_0)$$

From each $\hat{\Sigma}_i$ we can compute the leading coefficient $c_i$ of the morphisms induced by it, and since the lifted Verschiebung $\tilde{\mathcal{F}}$ is separable we can use a proposition of Satoh that shows that $\text{Tr}(\tilde{\mathcal{F}}) = c + 2^m$, then we can compute $c$ (and hence the trace) as a product of all the $c_i$s. Meaning that $t = \prod_{0 \leq i < n} c_i \mod 2^n$ (in fact, we compute $\prod_{0 \leq i < n} c_i^2$ to compute $|t|$).

The algorithm computes the coefficients $c_i$ using Velu’s formulae based on the equations of $\mathcal{E}_i, \mathcal{E}_{i+1}$, and the kernel of $\hat{\Sigma}_i$. Since we already know $\mathcal{E}_i, \mathcal{E}_{i+1}$ j-invariants, we can compute their equations using Newton iteration. The kernel of $\hat{\Sigma}_i$ can be computed in our case by lifting a single non-trivial torsion point, again with Newton iteration.

4 Implementation

4.1 General Overview

The project was implemented in C++ using GCC 4.8.1 compiler.

We used only the library NTL which handles large numbers, and polynomials over the integers and over finite fields. We also used STL’s data structures for easy access to data (e.g. vector).
4.2 Documentation

We implemented the project as a library providing certain cryptographical and mathematical services as follows:

- Generating cryptographically-strong elliptic curves (Using the class StrongEC).
- On strong elliptic curves, generating strong random points.
- Using curve arithmetic such as addition, doubling and scalar multiplication on non-supersingular curves of the form $y^2 + xy = x^3 + b$ (Using the class EllipticCurve).
- Generating random points on a curve, checking whether a point is on a curve.
- Key exchange using ECDH (Using the class ECDHClient).

We give a short description for every class made. All classes are tightly linked with NTL library, especially the GF2E class. After the overview we will list important things to do before using the library:

- **Constants:** A class containing the constants used in the various algorithms in the library. The constants are held there to optimize performance.

- **ProjectivePoint:** A class containing 3 GF2E elements, and represent a Lopez-Dahab coordinate in the projective plane. A point may be created using 2 or 3 coordinates (if Z isn’t given it is 1 by default), my be altered and accessed in all coordinates. A point different then $\mathcal{O}$ can be converted to an affine point. A point may be negated and compared to other points (including infinity).

- **EllipticCurve:** A class representing a curve of the form $y^2 + xy = x^3 + b$, and contains a single GF2E element ($b$). This class provides point arithmetic: addition, doubling and scalar multiplication. It can also check if a point is on a curve and return the curve’s $j$-invariant.

- **StrongEC:** A class representing a cryptographically-strong curve. The class contains an EllipticCurve element, the order of that curve, and its factorization to $\text{order} = n \cdot p$ ($n$ is a small integer and $p$ is a large prime). The class also holds a ProjectivePoint element which serves as the generator for the curve’s large prime subgroup. The class provides generation of strong elliptic curves (both random and custom), and allows generation of random points and random strong points. It also allows to check whether a given point is strong or not. It also provides an interface for point arithmetic on the curve.

- **ECDHClient:** A class providing an interface for ECDH, the class contains a StrongEC element, a ZZ element and a ProjectivePoint element representing the private and public keys of the client, and a ProjectivePoint representing the shared secret for that ECDH session. The class allows to create a client with a curve and a private key and allows access and alterations to its fields. The function ecdh receives two clients who agreed on their curve and generator and calculates and updates their shared secret.
To use our classes and implementation, one must include "CryptoLib.hpp" at the beginning of the program. Secondly, one must choose their GF2E modulus (constructing it via the NTL class GF2X and then calling GF2E::init with that element). And third, he needs to call the function Constants::initConstants() which initializes all the constants based on the current modulus.

**Note:** Every change to the GF2E modulus must be preceded with Constants::initConstants().

### 4.3 Examples

In this section we display a couple of short examples to use our classes.

#### Example of initializing the program to run with our library:

```cpp
#include <CryptoLib.hpp>
#include <NTL/GF2XFactoring.h>
#include <time.h>

int main() {
    // Initialize NTL seed to ensure randomness
    NTL::SetSeed(ZZ(INIT_VAL, clock()));
    // Build irreducible polynomial
    NTL::GF2X p;
    NTL::BuildIrred(p, 127);
    // Initialize GF2E modulus
    GF2E::init(p);
    // Initialize constants to fit the new modulus
    Constants::initConstants();
    // your code here!
}
```

#### Example of building points:

```cpp
// building the x,y-coordinates of the point
GF2E x,y;
std::stringstream ss;
ss << "[1001]";
ss >> x; // x = z^3 + 1
ss << "[111]";
ss >> y; // y = z^2 + z + 1
ProjectivePoint p(x,y);
```

#### Example of creating a simple elliptic curve:

```cpp
// Creating GF2E polynomial from a string
std::stringstream ss;
GF2E b;
ss << "[0 1 0 1]";
ss >> b; // b = z^4 + z^3 + z
// Creating an elliptic curve y^2 + xy = x^3 + b
EllipticCurve e(b);
// Printing the b parameter of the curve
std::cout << "b = " << e.getB() << std::endl;
```

#### Example of creating strong curves:

```cpp
// Creating a custom strong EC
```
Example of point generation and point arithmetic:

```cpp
// Generate a strong (of order p) point from a strong EC
StrongEC e1;
ProjectivePoint p1 = e1.genStrongPoint();
ProjectivePoint p2 = e1.genStrongPoint();
// Point arithmetic
ProjectivePoint p_res;
p_res = e1.pointAdd(p1, p2); // p_res = p1+p2
if (!p_res.isINF())
    p_res = p_res.toAffine(); // convert p_res to affine
p_res = e1.pointDoubling(p1); // p_res = 2*p1
p_res = e1.pointMul(p1, ZZ(INIT_VAL, 6)); // p_res = 6*p1
if (!e1.getCurve().isOn(p_res))
    std::cout << "result is not on the curve" << std::endl;
```

Example of ECDH:

```cpp
// A generator is chosen in random by default, thus
// both clients agree on a the curve and generator.
StrongEC e2;
// deciding on both private keys.
NTL::ZZ pv_key_alice = ZZ(INIT_VAL, 1337);
NTL::ZZ pv_key_bob = ZZ(INIT_VAL, 9001);
// building the ECDH clients
ECDHCClient alice = ECDHCClient(e2, pv_key_alice);
ECDHCClient bob = ECDHCClient(e2, pv_key_bob);
// agreeing on the shared secret (1337*9001*generator).
ecdh(alice, bob);
std::cout << "Shared secret: " << alice.getSharedSecret() << std::endl;
// switching the generator for ecdh. (notice that the curve is still e)
ProjectivePoint new_generator = e2.genStrongPoint();
bob.setGenerator(new_generator);
alice.setGenerator(new_generator);
ecdh(alice, bob);
std::cout << "Shared secret: " << alice.getSharedSecret() << std::endl;
```

4.4 Montgomery Point Multiplication

Montgomery method is an efficient way to compute \( kP \) (\( P \) is a point on \( E \) and \( k \) an integer) over a non-supersingular elliptic curve over \( GF(2^m) \).

As we recall from part 2, the formulas for \( 2P = (x_3, y_3) \) where \( P = (x_1, y_1) \) are

\[
x_3 = x_1^2 + b \\
y = x_1^3 + (x_1 + y_1) x_3 + x_3
\]

Notice that the x-coordinate of \( 2P \) does not involve the y-coordinate of \( P \). This observation will be used in the derivation of the Montgomery method.
If $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, one can compute the $x$-coordinate of $P_1 + P_2$ as follows:

$$x_3 = \frac{x_1 y_2 + x_2 y_1 + x_1 x_2^2 + x_2 x_1^2}{(x_1 + x_2)^2}$$

Also, if $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ and $P = (x, y)$. Assume that $P_2 = P_1 + P$, then the $x$-coordinate of $P_1 + P_2$ can be computed as follows:

$$x_3 = \begin{cases} 
  x + \left(\frac{x_1}{x_1 + x_2}\right)^2 + \frac{x_1}{x_1 + x_2}, & P_1 \neq P_2 \\
  x^2 + \frac{b}{x_1^2}, & P_1 = P_2
\end{cases}$$

Finally, if $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ and $P = (x, y)$. Assume that $P_2 = P_1 + P$ and that $x \neq 0$, Then the $y$-coordinate of $P_1$ can be computed in terms of $P$, and the $x$-coordinate of $P_1, P_2$ as follows:

$$y_1 = (x_1 + x) \frac{(x_1 + x)(x_2 + x) + x^2 + y}{x + y}$$

The following algorithm, based on these previous observations, implements Montgomery method in affine coordinates.

---

**Input:** An integer $k \geq 0$ and a point $P = (x, y) \in E$.

**Output:** $Q = kP$.

1. if $k = 0$ or $x = 0$ then output(0, 0) and stop.
2. Set $k \leftarrow (k_{l-1} \ldots k_1 k_0)_2$.
3. Set $x_1 \leftarrow x, \ x_2 \leftarrow x^2 + b/x^2$.
4. for $i$ from $l - 2$ downto 0 do
   Set $t \leftarrow \frac{x_1}{x_1 + x_2}$.
   if $k_i = 1$ then
      Set $x_1 \leftarrow x + t^2 + t, \ x_2 \leftarrow x^2 + b/x^2$.
   else
      Set $x_1 \leftarrow x^2 + b/x^2, \ x_2 \leftarrow x + t^2 + t$.
5. Set $r_1 \leftarrow x_1 + x, \ r_2 \leftarrow x_2 + x$.
6. Set $y_1 \leftarrow r_1(r_1 r_2 + x^2 + y)/x + y$
7. return $(Q = (x_1, y_1))$.

This method requires no precomputations (unlike the NAF methods) and on projective coordinates, requires exactly $6 \ceil{\log_2 k} + 10$ field multiplications for computing $kP$. Also, unlike regular multiplication methods (if the current bit in the integer is 0, we double, and if the current bit is 1 we double and add) the same operations are performed in every iteration of the main loop, thereby potentially increasing resistance to timing attacks and power analysis attacks. This method also works very well with LD projective coordinates, and therefore we decided to implement it in LD coordinates.
4.5 Point Counting Algorithm

We now give the pseudocode and short explanation of the main part of our project - Vercauteren's variant for Satoh's point counting algorithm.

For $1 \leq i < n$ we define the elliptic curve $E_i$ by the equation $y^2 + xy = x^3 + b^{2n-i}$ and let $E_i$ be the canonical lift of $E_i$. We can compute $J_i \equiv j(E_i) \mod 2^N$, starting from $J_{i+1} \equiv j(E_{i+1}) \mod 2^{N-1}$ using a univariate Newton iteration on the polynomial $\Phi_2(X, J_{i+1})$, with

$$\Phi_2(X, Y) = X^3 + Y^3 - X^2Y^2 + 1448 \left( XY^2 + X^2Y \right) - 162000 \left( X^2 + Y^2 \right) + 40773375XY + 8748000000(X + Y) - 15746400000000$$

The algorithm Lift_Previous_J_Invariant computes coefficients $A, B, C \in (\mathbb{Z}/2^N\mathbb{Z})[x]/(f(x))$ (represented by polynomials with coefficients in $\mathbb{Z}/2^N\mathbb{Z}$ reduced modulo $f(x)$), such that

$$\Phi_2(X, J_{i+1}) = X^3 + AX^2 + BX + C \mod 2^N$$

and then calls the recursive algorithm Lift_Previous_J_Invariant.Rec which performs the Newton iteration on the cubic polynomial $X^3 + AX^2 + BX + C$.

With every call of the algorithm Lift_Previous_J_Invariant we gain 1 bit of precision, so if we would like to compute $J_0 \equiv j(E_0) \mod 2^N$ then it suffices to start with $j(E_{N-1}) \equiv j(E_{N-1}) \mod 2$ and iterate this algorithm $N-1$ times, which immediately leads to algorithm Lift_First_J_Invariant.

The algorithm main body is Algorithm 5: Compute_Trace which, after lifting the first j-invariant to the required precision, calculates a new j-invariant each time and uses it and previous j-invariant to lift the kernel of $\hat{\Sigma}$ (part 4.2 in the algorithm) to find $c_2^i$ (parts 4.3, 4.4, 4.5). after calculating all $c_2^i$, we take the square root to find $|t|$. Hasse’s theorem states that $|t| \leq 2 \cdot 2^{2n}$ and therefore if $|t|$ is larger then it is negative (part 6). once we have $t$ we compute the order: $2^n + 1 - t$.

---

Algorithm 2 (Lift.Previous.J.Invariant)

**IN:** $J_{i+1} \in R \mod 2^N$ with $J_{i+1} \equiv j(E_{i+1}) \mod 2^{N-1}$ and a precision $N$.

**OUT:** $J_i \in R \mod 2^N$ with $J_i \equiv j(E_i) \mod 2^N$.

1. $A \equiv -J_{i+1}^2 + 1488J_{i+1} - 162000 \mod 2^N$;
2. $B \equiv 1488J_{i+1}^2 + 40773375J_{i+1} + 8748000000 \mod 2^N$;
3. $C \equiv J_{i+1}^3 - 162000J_{i+1}^2 + 8748000000J_{i+1} - 157464000000000 \mod 2^N$;
5. Return $J_i$. 

---

13
Algorithm 3 (Lift.Previous_J.Invariant.Rec)

**IN:** Elements \( J_{i+1}, A, B, C \in \mathcal{R} \mod 2^N \) with \( J_{i+1} \equiv j(\mathcal{E}_{i+1}) \mod 2^{N-1} \), 
\( \Phi_2(X, J_{i+1}) \equiv X^3 + AX^2 + BX + C \mod 2^N \) and a precision \( N \).

**OUT:** An element \( J_i \in \mathcal{R} \mod 2^N \) with \( J_i \equiv j(\mathcal{E}_i) \mod 2^N \).

1. If \( N = 1 \) Then
   1.1. \( J_i = J_{i+1}^2 \mod 2 \);
2. Else
   2.1. \( N' = \left\lfloor \frac{N}{2} \right\rfloor \);
   2.2. \( J_i = \text{Lift.Previous_J.Inv.Rec}(J_{i+1}, A, B, C, N') \);
   2.3. \( J_i \equiv J_i - \frac{J_i^3 + AJ_i^2 + BJ_i + C}{3J_i^2 + 2AJ_i + B} \mod 2^N \);
3. Return \( J_i \).

Algorithm 4 (Lift.First.J.Invariant)

**IN:** A \( j \)-invariant \( j_0 \in \mathbb{F}_{2^n} \setminus \mathbb{F}_4 \) and a precision \( N \).

**OUT:** \( J_0 \in \mathcal{R} \mod 2^N \) with \( J_0 \equiv j_0 \mod 2 \) and \( \Phi_2(J_0, \Sigma(J_0)) \equiv 0 \mod 2^N \).

1. \( J_0 \equiv J_0^{2^{(n-N+1)}} \mod 2 \);
2. For \( i = 2 \) To \( N \) Do
   2.1. \( J_0 = \text{Lift.Previous_J.Invariant}(J_0, i) \);
3. Return \( J_0 \).
In order to calculate the denominators and square roots we need to use special algorithms to compute the lifted inverse and lifted inverse square root over \((\mathbb{Z}/2^N\mathbb{Z})[x]/(f(x))\):

**Algorithm 5 (ComputeTrace)**

**IN:** A j-invariant \(j \in \mathbb{F}_p \setminus \mathbb{F}_4\) of an elliptic curve \(E\).

**OUT:** The trace of Frobenius \(t = q + 1 - \#E(\mathbb{F}_q)\) of \(E\).

1. \(N = \left\lceil \frac{n}{2} \right\rceil + 13; M = N - 10;\)
2. \(J = \text{LiftFirstInvariant}(j, N);\)
3. \(CN = 1; CD = 1;\)
4. For \(i = 0\) To \(n - 1\) Do
   4.1. \(J' = \text{LiftPreviousInvariant}(J, N);\)
   4.2. \(Z = \frac{(J^2 + 195120J + 4095J' + 660960000)/2^{12}}{(J^2 + J(563760 - 512J') + 372735J' + 8981280000)/2^9} \mod 2^N;\)
   4.3. \(T = (12Z^2 + Z)(J' - 1728) - 36 \mod 2^M;\)
   4.4. \(CN = CN \times (J' - (504 + 12096Z)T) \mod 2^M;\)
   4.5. \(CD = CD \times (240T + J') \mod 2^M;\)
   4.6. \(J = J';\)
5. \(t = \text{Sqrt}(CN/CD, 1, M) \mod 2^{M-1};\)
6. If \(t > 2\sqrt{q}\) Then \(t = t - 2^{M-1};\)
7. Return \(t.\)

**Algorithm 12.10 Inverse**

**INPUT:** A unit \(a \in \mathbb{Z}_q\) and precision \(N.\)

**OUTPUT:** The inverse of \(a\) to precision \(N.\)

1. **if** \(N = 1\) **then**
2. \(z \leftarrow 1/a \mod p\)
3. **else**
4. \(z \leftarrow \text{Inverse}(a, \left\lceil \frac{M}{2} \right\rceil)\)
5. \(z \leftarrow z + z(1 - az) \mod p^N\)
6. **return** \(z\)
To later compute the square root of a, we compute $\frac{1}{\sqrt{a}}$ and multiply it by $a$: $\left(\frac{1}{\sqrt{a}} \cdot a = \sqrt{a}\right)$.
5 Challenge

We were given the following challenge by Barukh Ziv, our supervisor:

Alice and Bob want to establish a secure connection. They decided to use a simple ECDH protocol for key agreement.

The following information is known about their domain parameters (all parameters are given in hex):

1. The underlying field is a binary field with modulus:
   
   0x800000000000000000000000000000000000000000000000000000000000001a.

2. The generating elliptic curve is of the form $y^2 + xy = x^3 + b$.

3. The generator of the elliptic curve is the point $G = (x_{G}, y_{G})$
   
   0x6893b18b4775bfa53e7543ef52fb57950d5d9e891fd
   5c58203fdd795991d952c8,
   0xe1550c0d086d37b3cc2bfc3d059b57ffe90e012c6cc20d4a81bcb830d9
   4a8719e279).

4. Alice and Bob chose a rather peculiar way to generate their private keys: each party chose a different elliptic curve over the same binary field, and calculated the largest odd divisor of its order. If this number was less than the order of the generator $G$, it was chosen as a private key. Otherwise, another elliptic curve was generated.

5. The only information about elliptic curves generated by Alice and Bob is the points on the curves:

   $P_{\text{alice}} = (x_{1}, y_{1})$
   
   0x65df428236bf6001b1570a0e173626646f7268a8ca6a4e4eb
   cd46c34d98aaa240c36552,
   0x714ade1230e5a18a4d71a13b12bbcc54ec7b618f75b3a22ccc7196245b1c
   fa69688b1a)

   $P_{\text{bob}} = (x_{2}, y_{2})$
   
   0x337da53d81aa672790f23747d4e5bf39491487731800db708
   e475f6b9606a6c7899a760,
   0x8eeff89ba80d647794d685f73e46a96cd151bb50ae8b15dab9d56622900e
   d025bd25e)

Questions:

A. What is the order of the group generated by point $G$?

B. What is the shared secret key Alice and Bob generated during ECDH session?

Solution:

First, we use the point $G = (x_{G}, y_{G})$ to solve $y^2 + xy = x^3 + b$ to obtain $b$. Once we have $b$
we count the points on the curve using Satoh, and factor it. In our case the order factored into $2^2 \cdot p$ where

$$p = 0x1ff\text{fffffffffffffffffffffffffffffffffffffffffffe}245059837567481\text{eff95f33935d1771af7}$$

We then checked if G’s order is $p$ and it was.

For the second part, we use Alice’s and Bob’s points $(x_A, y_A), (X_B, y_B)$ to solve their curve equation (as before) and find $b$. Again we count the points on these curves using Satoh and then we divide the order by 2 as much as possible (because by definition their private key is the largest odd divisor of the order of the curves they generated). Once we have their private keys $d_A, d_B$ we compute $d_Ad_BG$ to find the shared secret, $(x_S, y_S)$ where:

$$x_S = 0x28034eb5f54c7b036ae3e6f7e8c35d0a7b44fcb17f03224ad9a9ca2b9ce3cc7c59f4176$$

$$y_S = 0x60d2fa7c2ca4851339c1bafe0e1465a79bdecca676a8e44848475a333a059306676dfe$$

### 6 Performance

The testing were run on an Intel® Core™ i5-4570 Processor 3.20 GHz.

#### EC Arithmetics - Results are an average of $10^5$ tasks

<table>
<thead>
<tr>
<th>field bits</th>
<th>addition</th>
<th>doubling</th>
<th>random element $\times$ point</th>
</tr>
</thead>
<tbody>
<tr>
<td>113</td>
<td>3.3µs</td>
<td>2.48µs</td>
<td>0.085ms</td>
</tr>
<tr>
<td>163</td>
<td>7.3µs</td>
<td>4.45µs</td>
<td>0.24ms</td>
</tr>
<tr>
<td>233</td>
<td>7.02µs</td>
<td>5.45µs</td>
<td>0.39ms</td>
</tr>
<tr>
<td>283</td>
<td>10.2µs</td>
<td>8.7µs</td>
<td>0.84ms</td>
</tr>
<tr>
<td>409</td>
<td>16.9µs</td>
<td>9.01µs</td>
<td>1.7ms</td>
</tr>
</tbody>
</table>

#### Point Counting and Point Generation

<table>
<thead>
<tr>
<th>field bits</th>
<th>Count Points</th>
<th>Generate random point</th>
<th>Generate strong point</th>
</tr>
</thead>
<tbody>
<tr>
<td>113</td>
<td>3.38s</td>
<td>15.66µs</td>
<td>0.052ms</td>
</tr>
<tr>
<td>163</td>
<td>10.24s</td>
<td>48µs</td>
<td>0.074ms</td>
</tr>
<tr>
<td>233</td>
<td>21.51s</td>
<td>53µs</td>
<td>0.115ms</td>
</tr>
<tr>
<td>283</td>
<td>51.67s</td>
<td>104.6µs</td>
<td>0.195ms</td>
</tr>
<tr>
<td>409</td>
<td>102.15s</td>
<td>142µs</td>
<td>0.327ms</td>
</tr>
</tbody>
</table>

#### Finding a cryptographically strong curve

<table>
<thead>
<tr>
<th>field bits</th>
<th>finding a strong curve</th>
<th>average attempts</th>
</tr>
</thead>
<tbody>
<tr>
<td>113</td>
<td>7.201s</td>
<td>2.140</td>
</tr>
<tr>
<td>163</td>
<td>32.042s</td>
<td>3.144</td>
</tr>
<tr>
<td>233</td>
<td>175.409s</td>
<td>8.141</td>
</tr>
<tr>
<td>283</td>
<td>368.604</td>
<td>7.185</td>
</tr>
<tr>
<td>409</td>
<td>1478.259</td>
<td>14.52</td>
</tr>
</tbody>
</table>
7 Conclusion

In this project we investigated a part of the (big) world of elliptic curve cryptography. This fascinating field enables to maintain security with much smaller key sizes than the ones needed for public key cryptography over $\mathbb{Z}_p$, allowing devices with low processing power or low power consumption (like mobile phones or smart cards) to use secure encryption. We have implemented a general purpose elliptic curve arithmetic library, and an efficient algorithm for random generation of elliptic curves with cryptographically strong properties. Also, to test our library, we have implemented a basic public key cryptography algorithm (ECDH). We have completed the challenge given to us by our supervisor, Barukh Ziv, and gained valuable experience for future projects!

7.1 What We Have Learnt

We have gained theoretical and mathematical knowledge in elliptic curve cryptography. As well as understanding and implementing complicated algorithms in practice. We have learnt working with (and debugging) an open source library as big as NTL, which will no doubt help us understand (and debug) the next existing library we have to use (in the next project or perhaps the industry). We have also learnt the value of optimization and that a working algorithm is not enough, and there is always room for optimization and improvement.

7.2 Future Goals

Even after the project ends, we will continue investigating new ways to improve our work:

- Continue optimizing our point counting algorithm.
- We have started implementing another point counting algorithm - AGM. We intend to finish it and optimize it as well.
- In 6-12 months NTL will release a thread-safe version, we intend to use it to parallelize independent parts of the point counting algorithm and the strong curve generation.
- Add more elliptic curve cryptography algorithms (aside from ECDH) to our file “CryptoLib.hpp” such as ECDSA or ElGamal, so we could use our library as a real elliptic curve cryptography library with various services.

8 Bibliography


• Julio Lopez, Ricardo Dahab - *Fast multiplication on elliptic curves over* $GF(2^m)$ *without precomputation* (1999).


