Project in Computer Security *September 2013*

Elliptic Curve Cryptography ***Final Report***

Amit Markel ***Leonid Nemirovskiy*** *Supervisor.* ***Barukh Ziv***

Technion - Israel Institute of Technology and Science

Contents

Part I. Introduction

1 Abstract - Elliptic Curves in cryptography

Elliptic curve cryptography (*ECC*) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields.

ECC is becoming more popular in the practical matter in cryptography hence our attention on this subject.

The group order is one of the *most* important aspects of identifying a cryptographically strong curve, thus it always has been in the spotlight of research; and thence being the main target of interest for us as well.

Characteristic two. *ECC* may be used over large prime fields or binary extension fields, we chose to focus on characteristic two mainly due to performance gains - the way that the hardware is based on is binary storage, therefore we are taking the advantages of this design; consequently, the curve results in a simpler form, affecting performance and perhaps ultimately perceiving the math better, despite the implementation being more tricky in contrast to using a large prime field of greater characteristic.

As we describe in the first **Theory** section, in fields of characteristic two, the form of the curve is given by,

$$E : y^2 + xy = x^2 + a_2x^2 + a_6 \, .$$

However, for gaining an additional performance speedup without compromising any strong cryptographic attribute, we fix $a_2 = 0$, hence getting the even simpler form,

$$y^2 + xy = x^2 + a_6 \, ,$$

thus we always assume a_2 to be zeroed, as explained in that particular later section, as well as that in our implementation we use polynomial bases as field elements.

In comparison. *ECC* depends on the computation of the discrete logarithm of random points on an elliptic curve regarding a known base public point, the public key; whereas earlier public key crypto-systems rely on the difficulty of factoring large numbers which are composed of several large prime factors.

Efficiency. *ECC*'s shiny attribute is its usage of shorter key sizes, in contrast to *RSA* or *Discrete Logarithm* (*DL*) in that matter. This allows narrowed usage of memory, storage and network traffic: a 256-bit *ECC* key offers the same level of security as a 3072-bit *RSA* key. Solving *ECDLP* using any known algorithm requires $O\left(\sqrt{n}\right)$ time-complexity, hence if the size of the field, that the elliptic curve is over, is 2^n then the mentioned algorithm would take $O\left(2^{\frac{n}{2}}\right)$ steps to complete; in contrast to solving the RSA problem or classical DL with n as we use in ECC (approximately 200-300 bits), solving the problem can

be achieved in sub-exponential time, e.g. for breaking RSA: one may factor n using the *General Number Field Sieve Algorithm* with time-complexity of $O\left(e^{\sqrt[3]{\frac{64}{9}b}\cdot\sqrt[3]{\log^2 b}}\right)$ where b is the number of bits.

Mobile. Another aspect is the efficiency on weaker devices holding less processing power and limited network bandwidth; this is enabled by the small key size, and thus, condensed signature; while still maintaining high-end security 12-fold.

Part II. Mathematical Background

2 EC over finite fields

2.1 Basics

Elliptic curves may be defined over large prime fields or either binary extension fields. Due to performance issues we shall concentrate on Binary Extension Fields so we can get a simple form of the curve, taking the advantages the way Hardware is built on - characteristic two.

Definition 1. *An elliptic curve E over $\mathbb{F}_q$ such that $q = 2^p$ (where we set a **prime** p for enhanced security) is defined by,*

$$E: \quad y^2 + xy = x^3 + a_2x^2 + a_6 \quad \text{where} \;\; a_2 \in \mathbb{F}_q \;\; \text{and} \;\; a_6 \in \mathbb{F}_q^* \; .$$

For better performance we use curves where $a_2 = 0$ (doing so does not impact any cryptographic quality), hence we get even the simpler form,

$$E: \quad y^2 + xy = x^3 + a_6 \; .$$

2.2 Point Group

Definition 2. *Let E be an elliptic curve. The point $\mathbf{P} = (x, y) \in \mathbb{F}_q^2$ is defined to be on E when it satisfies E's equation.*

In order to build a group of points of the curve we must define the point at infinity, marked by $\mathcal{O}$. The set of all points on the curve along with $\mathcal{O}$ form a group, with $\mathcal{O}$ as its zero element.

Inversion. Let $\mathbf{P} = (x, y) \in E$ then $\mathbf{Q} = (x, x + y) = -\mathbf{P}$.

Addition. Let $\mathbf{P} = (x_1, y_1) \in E$, $\mathbf{Q} = (x_2, y_2) \in E$ then if $\mathbf{P} \neq \mathbf{Q}$ then $\mathbf{P} + \mathbf{Q} = -\mathbf{R}$ where $\mathbf{R}$ is the intersection of the straight line $\overline{\mathbf{PQ}}$ and E. Therefore we get

$$\mathbf{P}+\mathbf{Q} = (x_3, y_3) = \left(\left(\frac{y_1 + y_2}{x_1 + x_2} \right)^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + a_6 \;\; , \;\; \left(\frac{y_1 + y_2}{x_1 + x_2} \right) (x_1 + x_3) + x_3 + y_1 \right) \in E \; .$$

Doubling. Let $\mathbf{P} = (x, y) \in E$ then let l be the tangent line to the curve at $\mathbf{P}$, let $\mathbf{R}$ be the only (the third) point of intersection of l with the curve, and define $2\mathbf{P} = -\mathbf{R}$. Thence we get

$$2\mathbf{P} = (x_3, y_3) = \left(x_1^2 + \frac{a_6}{x_1^2} \;\; , \;\; x_1^2 + \left(x_1 + \frac{y_1}{x_1} \right) x_3 + x_3 \right) \in E$$

We next show an example of points on an elliptic curve which form a cyclic group, in which its elements can be obtained by multiplies of a generator element.

2.2.1 Example of an EC group

We use a representation of GF $\left(2^3\right)$ as $F\left[\xi\right] / \left(\xi^3+\xi+1\right)$.

$$\begin{array}{lcl} E & : & y^2+xy=x^3+\xi^4 \\ \#E\left(\mathrm{GF}\left(2^8\right)\right) & = & 8 \end{array}$$

$1\mathbf{g}$	$=$	$($	ξ^3	$,$	ξ	$)$	$(110,010)$
$2\mathbf{g}$	$=$	$($	ξ	$,$	ξ^2	$)$	$(010,001)$
$3\mathbf{g}$	$=$	$($	ξ^6	$,$	ξ^6	$)$	$(101,101)$
$4\mathbf{g}$	$=$	$($	0	$,$	ξ^2	$)$	$(000,001)$
$5\mathbf{g}$	$=$	$($	ξ^6	$,$	0	$)$	$(101,000)$
$6\mathbf{g}$	$=$	$($	ξ	$,$	ξ^4	$)$	$(010,011)$
$7\mathbf{g}$	$=$	$($	ξ^3	$,$	1	$)$	$(110,100)$
$8\mathbf{g}$	$=$			$\mathcal{O}$			$\mathcal{O}$

$\mathrm{F}[\xi]/\left(\xi^3+\xi+1\right)$, aka polynomials mod ξ^3+ξ+1 .

2.3 The Projective Plane

Addition in standard affine coordinates requires one to use inversions over $\mathbb{F}_q$, which substantially reduce performance in comparison to multiplications and additions, hence we use projective coordinates in order to substitute the mentioned inversions with multiplications and additions.

Initially one affixes Z with 1, and matches an affine point to a projective one in this manner,

$$(x,y) \mapsto X:Y:1 \, .$$

Lopez-Dahab's [4] projective coordinates provide the *best results* in terms of *performance* when used over Binary Fields. A point $X:Y:Z$ in *Lopez-Dahab* corresponds to its affine form by,

$$\left(X/Z,\ Y/Z^2\right) \ .$$

Lopez-Dahab Addition. Let $\mathbf{P}=X_1:Y_1:Z_1$, $\mathbf{Q}=X_2:Y_2:Z_2$ such that $\mathbf{P} \neq \pm\mathbf{Q}$, then $\mathbf{P}+\mathbf{Q}=X_3:Y_3:Z_3$ is given by,

$$\begin{array}{rcl} A_1 & = & X_1Z_2 \\ A_2 & = & X_2Z_1 \\ C & = & A_1+A_2 \\ E_1 & = & Y_1Z_2^2 \\ E_2 & = & Y_2Z_1^2 \\ F & = & E_1+E_2 \\ G & = & CF \end{array}$$

$$\begin{aligned} Z_3 &= Z_1 Z_2 D \\ X_3 &= A_1 (E_2 + B_2) + A_2 (E_1 + B_1) \\ Y_3 &= (A_1 G + E_1 D) D + (G + Z_3) X_3 \ . \end{aligned}$$

Lopez-Dahab Doubling. If $\mathbf{P} = X_1 : Y_1 : Z_1$ then $2\mathbf{P} = X_3 : Y_3 : Z_3$ is given by,

$$\begin{aligned} S &= X_1^2 \\ T &= Z_1^2 \\ Z_3 &= ST \\ T &= a_6 T^2 \\ X_3 &= S^2 + T \\ Y_3 &= \left(Y_1^2 + T\right) X_3 + T Z_3 \ . \end{aligned}$$

Comparing Lopez-Dahab and affine representation by number of operations.

Number of operations for EC arithmetic. If one implements the **standard** arithmetic they would have to compute the following steps.

Addition	Doubling
$A_1 = y_1 + y_2$	$D_0 = x_1^2$
$A_2 = x_1 + x_2$	$D_1 = D_0^{-1}$
$A_3 = A_2^{-1}$	$D_2 = a_6 D_1$
$A_4 = A_1 A_3$	$D_3 = D_0 + D_2$
$A_5 = A_4^2$	$D_4 = x_1^{-1}$
$A_6 = A_2 + A_4 + A_5 + a_6$	$D_5 = y_1 D_4$
$A_7 = x_1 + x_3$	$D_6 = x_1 + D_5$
$A_8 = A_4 A_7$	$D_7 = D_6 x_3$
$A_9 = A_8 + x_3 + y_1$	$D_8 = D_0 + D_7 + x_3$

.

Comparison.

	Addition				Doubling			
Projective	Multiplications	Squarings	Additions		Multiplications	Squarings	Additions	
	13	4	9		4	5	4	
Affine	Multiplications	Squarings	Additions	Inversions	Multiplications	Squarings	Additions	Inversions
	2	1	8	1	3	1	4	2

We will elaborate on our implementation in a later section, however, after implementing the above EC arithmetic (addition, doubling, powering, using the projective plane); next we discuss finding a cryptographically strong EC.

3 Cryptographically Strong

One counts the number of points on a given curve to check whether it is divisible by a large prime number q (and possibly by more much smaller factors) and finds a point $\mathbf{P}$ such that the order of $\mathbf{P}$ is q (one surely exists); then one can use the cyclic subgroup defined by $\langle \mathbf{P} \rangle$, the generating group of $\mathbf{P}$, which is cryptographically strong due to the algebraic properties its elements acquire.

We check that the number of points is of the form $\mathsf{q} \cdot m$ where m is a small number in the form $\prod m_i^{\alpha_i}$ where m_i are small primes, thus getting a large order $\langle P \rangle$: one easily finds a point of order q by randomly choosing a point $\mathbf{X}$ on the curve and checking whether $m\mathbf{X} \neq \mathcal{O}$. We have a very high probability of getting $\mathbf{X}$ of order $\mathsf{q}m$, hence we achieve the point $m\mathbf{X}$ whose order is q, as needed.

4 Point Counting

Overview. We chose to focus on a memory efficient algorithm variant of Satoh's algorithm, Vercauteren's Algorithm, who extended Satoh's algorithm to characteristic two. Let E be an elliptic curve over a finite field $\mathbb{F}_q$ where $q = 2^n$ and an invariant element $j(E) \notin \mathbb{F}_4$, then the algorithm computes the number of points requiring $O\left(n^3 \log n \log\log n\right)$ operations. We implemented a memory efficient version requiring $O\left(n^2\right)$ memory instead of $O\left(n^3\right)$.

The number of points $\#E(\mathbb{F}_q)$ satisfies $\#E(\mathbb{F}_q) = q + 1 - t$, where t is the trace of the Frobenius endomorphism $F : E \to E : (x, y) \mapsto (x^q, y^q)$. From Hasse's theorem $|t| \leq 2\sqrt{q}$ holds.

Lifting. In order to calculate the number of points on a given curve, one must compute the trace of Frobenius t first. As stated, we handle curves wielding only a_6 which is represented by polynomials over $\mathbb{F}_2$ and is difficult and an unsettling representation to compute. The Frobenius endomorphism can be decomposed, hence allowing lifting for every a_6^i in the process, resulting in a multiplication that yields polynomials over $\mathbb{F}_{2^i}$, as the reduction modulo 2 always remains the same as the original curve.

Example of an arbitrary lift. We lift the curve $E : y^2 + xy = x^3 + a_6$ to 2-adic extension with precision 2^3 so that the reduction modulo 2 results in the same polynomial. We acquire the lifted curve with A_6, given by,

$$\begin{aligned} a_6 &= 1 \cdot \xi^7 + 0 \cdot \xi^6 + 1 \cdot \xi^5 + 1 \cdot \xi^4 + 0 \cdot \xi^3 + 1 \cdot \xi^2 + 0 \cdot \xi + 1 \\ A_6 &= 1 \cdot \xi^7 + 4 \cdot \xi^6 + 1 \cdot \xi^5 + 7 \cdot \xi^4 + 0 \cdot \xi^3 + 1 \cdot \xi^2 + 2 \cdot \xi + 3 \end{aligned}$$

Canonical Lifting. The canonical lift is a unique lift which gives the best properties for computing the trace of Frobenius. There are many ways to lift a curve, hence the required canonical way of lifting one – that one which does not alter the endomorphism ring, and thus the trace of Frobenius, which indeed

remains constant as well as its dual. We are using a known method based on Vélu's formulas to actually compute the trace with adequate precision.

The idea of Satoh's algorithm, formally. Lifting both the curve E and the Frobenius endomorphism F to the valuation ring $\mathcal{R}$ of a degree n unramified extension $\mathcal{K}$ of the p-adic field $\mathbb{Q}_p$. Since this lifting is done in a canonical way, the trace of the lifted Frobenius $\mathcal{F}$ equals to the trace of Frobenius, t. However, 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 and can be lifted explicitly by lifting its kernel. Analyzing the action of the lift $\hat{\mathcal{F}}$ of $\hat{F}$ on the formal group of the canonical lift $\mathcal{E}$, we obtain an expression for the trace of $\hat{\mathcal{F}}$ which equals to the trace of Frobenius t. [8]

The canonical lift of an elliptic curve. The main step in Satoh's algorithm is lifting the curve E and the Verschiebung $\hat{F}$ to the valuation ring $\mathcal{R}$ of a degree n unramified extension $\mathcal{K}$ of $\mathbb{Q}_p$. Among the many possible lifts of E from $\mathbb{F}_q$ to $\mathcal{R}$ there is one which has particularly nice properties, called the canonical lift. The canonical lift $\mathcal{E}$ of a non-supersingular elliptic curve E over $\mathbb{F}_q$ is an elliptic curve over $\mathcal{K}$ which satisfies the following two properties: the reduction modulo p of $\mathcal{E}$ equals to E and $\mathrm{End}(E) \cong \mathrm{End}(\mathcal{E})$ as a ring. *Deuring* has shown that the canonical lift $\mathcal{E}$ always exists and is unique up to isomorphism.

Specifically - Characteristic two. Let E be an elliptic curve over a finite field $\mathbb{F}_q$, with $q = 2^n$ and $j(E) \notin \mathbb{F}_4$. It is well known that either E or its quadratic twist is isomorphic over $\mathbb{F}_q$ with an elliptic curve given by an equation of the form $y^2 + xy = x^3 + a_6$, with $a_6 \in \mathbb{F}_q^*$. Therefore, we can restrict ourselves to this case.

Let $\mathcal{K}$ be a degree n unramified extension of $\mathbb{Q}_2$ and $\mathcal{R}$ its valuation ring. Then $\mathcal{R}$ is isomorphic to $\mathbb{Z}_2[T]/f(T)$, with $f \in \mathbb{Z}_2[T]$ a monic polynomial of degree n such that its reduction modulo 2 is irreducible in $\mathbb{F}_2[T]$. In practice all computations are carried out in the ring $\mathcal{R}$ mod 2^N, which can be presented as $\left(\mathbb{Z}/2^N\mathbb{Z}\right)[T]/f(T)$.

Computing the trace. Computing the trace of the Frobenius endomorphism $F : E \to E$ is the goal. We assume we have computed a good approximation of the canonical lift $\mathcal{E}$ of E. Since canonical lifting preserves the endomorphism ring, the trace of Frobenius is unchanged. Moreover the trace of an endomorphism is the same as the trace of its dual. We Have $\mathrm{Tr}\, F = \mathrm{Tr}\, \mathcal{F} = \mathrm{Tr}\, \hat{\mathcal{F}}$, where $\mathcal{F}$ is the Frobenius on $\mathcal{E}$ and $\hat{\mathcal{F}}$ is its dual. As explained above, the Frobenius F can be decomposed into the product of little Frobenius isogenies which cycle the curve E through its d conjugates, and similarly for the duals. Thus $\mathrm{Tr}\, \hat{\mathcal{F}} = \mathrm{Tr}\, \left(\hat{\Sigma}_{d-1} \circ \hat{\Sigma}_{d-2} \circ \cdots \circ \hat{\Sigma}_0\right)$.

The next step is to go to the formal groups of the curve $\mathcal{E}$ and its conjugates. Let us denote the local parameter $-X/Y$ of $\mathcal{E}$ by τ, and the local parameters of $\mathcal{E}_i$ by τ_i. Expressing the action of $\hat{\mathcal{F}}$ on this parameter, we get the form $\hat{\mathcal{F}}(\tau) = \sum_{k\geq 1} c_k \tau^k$, and a proposition by Satoh shows that the equation $x^2 - \text{Tr}\left(\hat{\mathcal{F}}\right) x + q = 0$ satisfied by $\hat{\mathcal{F}}$ implies that $\text{Tr}\left(\hat{\mathcal{F}}\right) = c_1 + \frac{q}{c_1}$.

Thus, computing the first coefficient c_1 is enough to solve the problem. The key point is that, since $\hat{\mathcal{F}}$ is the composition of morphisms, it is possible to compute c_1 as the product of the first coefficients in the expansions of the factors in the formal groups. More precisely, we can write $\hat{\Sigma}_i : \mathcal{E}_i \to \mathcal{E}_{i+1}$ on the formal groups as $\hat{\Sigma}_i(\tau_i) = g_i \tau_i + O\left(\tau_i^2\right)$. Then we have $c_1 = \prod_{0\leq i<d} g_i$, and $\text{Tr}\, F \equiv \prod_{0\leq i<d} g_i \mod q$.

Note that this product of g_i is an expression for the norm from $\mathbb{Z}_q$ down to $\mathbb{Z}_p$ of g_0 and thus it is certainly in $\mathbb{Z}_p$. In addition it gives the trace of F to within $O\left(p^d\right)$, since if higher precision was desired it would be necessary to take the term q/c_1 into account.

The final step is to compute each g_i from the curves $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$ and the kernel of $\hat{\Sigma}_i$. This is done with the help of Vélu's formulae. Each g_i can be expressed as a rational fraction in terms of the data we lifted, and multiplying them together gives the trace with sufficient precision. [6]

Part III. Implementation

5 Overview

Language. The entire project is implemented in `C++` using GCC 4.7.3_1 compiler on a MacBookPro (UNIX 03 certified).

Libraries.

NTL	Handles field operations, large number processing,
GF2X	Improves polynomial handling performance,
GMP	Tweaks large number manipulation performance.
SMALLSHA1	Computes SHA-1 hashes for the ECDSA library.
GNU Parallel	Spawns process instances of a same program which generates a random curve and checks its cryptographic value, otherwise dumps it. A "winning" process terminates all others, and returns its curve to the shell, to be piped to the user's program.

6 Implementation

6.1 Documentation

The project was implemented as a *library*, **LibECC**, providing certain cryptographical services as follows.

- Creating random cryptographically-strong elliptic curves, and manipulating them.
  - Creating custom **EC** objects with known a_6 and/or *order*.
  - Generating random points on a given curve; checking whether a **Point** belongs to the curve.
- Elliptic curve point arithmetics for cryptographic operation use, for example *EC-El-Gamal* encryption and decryption, as we present later at the **Challenge** section.
- **Sig** - a library on top of **LibECC**, which allows generation and verification of digital signatures (*ECDSA*) using elliptic curves.

6.1.1 Classes overview

Field The **Field** class initializes a preloaded trinomial or pentanomial F for a specified field size N, to be used as the field modulus for **FieldElement**.

FieldElement A **FieldElement** is a polynomial mod F with coefficients over GF (2). The **FieldElement** class supports addition, multiplication, power, comparison, squaring, inversion, conversion (to/from Hex string representation of the coefficients, or to/from a **NTL::ZZ** number derived from representing the coefficients as a byte array) and input/output operators.

EC

Point The class **Point** holds three **FieldElement**s, as discussed in the **Theory** part of the document, we use the **Lopez-Dahab** representation. A point may be compared to the $\mathcal{O}$ element ($x \neq 0, y = z = 0$), converted to its *Affine* form as discussed in **Theory**: $x \leftarrow \frac{x}{z}$, $y \leftarrow \frac{y}{z^2}$, $z \leftarrow 1$.

EC The class **EC** stores the **FieldElement** a_6, may hold a calculated **Point** of good order (the large prime factor of the curve's order), and the order and its mentioned large prime factor. Constructed with an a_6 of type **FieldElement** or Hex representation via **std::string**. Supports *operator()* which evaluates a point to check whether it belongs to the curve, random point generation, point addition, multiplication, doubling, powering, inverting and counting with **Satoh**.

ECBuilder Runs the main algorithm `build()` implied in **Theory**: until we get a cryptographically strong curve, we generate one at random, calculate its order, and test the ratio of its order to a large prime q as discussed to a specified threshold. The algorithm `build_with_point_of_order()` finds a point **P** as well whose order is that large prime number.

PrimeExtractor Given an **NTL::ZZ** number, `Extract(number)` checks for every prime number p for the first `NUMBER_OF_PRIMES` whether $number \equiv_p 0$, if true, then $number \leftarrow number/p$ until $number \not\equiv_p 0$. The process stops when `NTL::ProbPrime(number)` is *True* or all prime numbers have been used.

HexToBinary Converts a number in base 16 in **std::string** format as number in base 2 in **std::string** format.

Protocols Implementation of **ECDSA** for generating and verifying signatures, however usage of the **Sig** class is recommended for end use.

Sig Small library of **ECDSA** as a layer over **Protocols**. One may create a **Sig** object, call `init_keys()` to re-initiate all **KeyTuples**, `set_key(d,key_index)` to load personal private keys for different goals, `generate(message,key_index)`, `verify(message,key_index,signature)`.

Satoh Used by **EC**. Takes a **FieldElement** representing the a_6 element of the curve. `Count_Points()` returns $2^N + 1 -$ `Compute_Trace`(). One should construct an **EC** object and call `get_order()` instead.

RandomNTL To ensure randomness, one should call `LIBECC_Randomize()` at the beginning of their program, otherwise `LIBECC_Randomize_Default_Seed()` is called, which issued `LIBECC_Randomize_Seed(14*17)` is issued by default - this macro expects a *seed* of type **unsigned long**.

FindIrreduciblePolynomial **NTL::GF2X** `FindIrreduciblePolynomial(n)()` may be used if desired for testing purposes, although not optimized: does not return a polynomial whose the number of zeroed coefficients is minimal thus getting a trinomial or pentanomial.

Time One may issue the following to time a block of operations in **seconds**. One must use a **double** type variable.

Algorithm 1 Get the time in seconds since program start with **Time**

```
#define FIELD_MODULUS_BITS 127
#include "LibECC.hpp"
... double t;
    Time_Start(t);
    // do stuff
    Time_End(t); ...
```

6.1.2 Usage requirements

- One must include **LibECC.hpp** at the beginning of their program, and (literally) on top of that, define `FIELD_MODULUS_BITS`, the field size in bits by issuing

  #`define` `FIELD_MODULUS_BITS` <number of bits>

  at the complete top.

  ```
  #define FIELD_MODULUS_BITS 127
  #include "LibECC.hpp"
  ...
  ```

  The following sizes have preloaded trinomials or pentamonials for use to define the field modulus polynomial; however if one wishes to use a different value they should add their polynomial of choice via the *Preprocessor*, as elaborated below.

  $$3, 11, 37, 127, 144, 239, 240, 271, 283, 288, 480, 487, 571\ .$$

The first and last coefficients of the polynomial are set by default.

- One may define both FIELD_MODULUS_BITS and FIELD_MODULUS_COEFFICIENTS at the beginning of their program. If FIELD_MODULUS_COEFFICIENTS is not defined then **FindIrreduciblePolynomial** will be used to create the polynomial if FIELD_MODULUS_BITS is not one of the numbers in the list above.

```
#define FIELD_MODULUS_BITS 283
#define FIELD_MODULUS_COEFFICIENTS {5,7,12}
#include "LibECC.hpp"
...
```

- Call LIBECC_Randomize() at int Main(···) to initialize **NTL** and **C** randomness features. Alternatively, one may set a specific **unsigned long** *seed*, e.g. LIBECC_Randomize_Seed(*seed*). Please do note that LIBECC_Randomize_Default_Seed() is available, and sets the mentioned *seed* to $14 * 17$.

- In order to use the supplied **ECDSA** library, issue the following

#define LIBECC_SIG

6.1.3 Examples

Algorithm 2 Building a cryptographically strong **EC** using the **ECBuilder**

```
EC ec = ECBuilder().build();
cout << ec << endl;
```

Possible output.

```
0x648842b554db90b14bf2933508e8e9af
170141183460469231714655346614483614532
247561612861723765245289752737
```

The first line is a_6, the second - $\#E(\mathbb{F}_{2^n})$, the third - q (the large prime factor).

Algorithm 3 Using a cryptographically strong **EC** for **ECDSA**

```
#define LIBECC_SIG
#define FIELD_MODULUS_BITS 127
#include "LibECC.hpp"
...
// build_with_point_of_order() must be issued instead of
    build() in order to use signatures; otherwise use ec.
    point_of_order_init() at a later time.
EC        ec = ECBuilder().build_with_point_of_order();
int       num_of_keys = 1417;
string    message     = "Hello␣world!";
int       key_index   = 17;
Signature signature   = sig.generate(message,key_index);

if (sig.verify(message,key_index,signature))
    // Signature Accepted.
else
    // Signature REJECTED.

sig.init_keys(); // randomize entire keys database.
PROTOCOL_ECDSA::KeyTuple kt = sig.get_keytuple(17);
// one may read kt.r, kt.s.

sig.set_key("171717141414171717",17);

Point public_key("0x3f713d3f9d88eb435e5fba7397f01292",
                 "0x22a45b9ec718604eb67048c28d2d26a7");
sig.verify(message, public_key, signature);
```

Algorithm 4 Using a custom **EC** by a_6 represented in base 16 as **std::string**

```
string a6 = "0x648842b554db90b14bf2933508e8e9af";
EC     ec = EC(a6);
// calculate number of points (allowing lazy init).
ec.get_order();

// must be called before using Sig with a custom EC.
ec.point_of_order_init();
```

Algorithm 5 Using a custom **EC** by a_6 and known order in base 10

```
string a6    = "0x648842b554db90b14bf2933508e8e9af";
string order = "170141183460469231714655346614483614532";
EC     ec    = EC(a6,order); // accepts NTL::ZZ as well.
```

Algorithm 6 EC Point arithmetic

```
EC ec = ECBuilder().build();
FieldElement x("0x3f713d3f9d88eb435e5fba7397f01292");
FieldElement y("0x22a45b9ec718604eb67048c28d2d26a7");
Point p    = ec.random_point(); // belongs to ec.
Point zero = Point();
Point pxy  = Point(x,y);
if (!ec(pxy)) {
    // the point pxy is not on the curve.
    // do stuff.
}
Point& reference_to_p = p_add(p,pxy) // p = p+pxy.
p_double(pxy);
p_power(p,ec.get_q);
p_inverse(p);
if (zero.isNeturalElement())
    // perform operations should a point be O.
p.toAffine();
pxy.toNetural(); // sets a point to the zero element.
cout << (p!=pxy ? "Different" : "Same") << endl;
cout << p.x.toHex() << endl << p.x.toZZ() << endl;
```

Algorithm 7 Miscellaneous **EC** methods

```
EC ec = ECBuilder().build();
Point p = ec.random_point();

// this must be called should one wishes to use
   signatures from this point on with this ec.

ec.point_of_order_init();
Point point_of_order_q = ec.point_of_order_get();
```

6.2 Efficient Satoh Point Counting

6.2.1 Lifting the j-invariants

For $i \in [1, n)$ define the elliptic curve E_i by the equation $y^2 + xy = x^3 + a^{2^{n-i}}$ and let $\mathcal{E}_i$ be the canonical lift of E_i. Using Proposition 2 [8], we can compute $J_i \equiv j(E_i) \bmod 2^N$, starting from $J_{i+1} \equiv j(E_{i+1}) \bmod 2^{N-1}$, using a univariate Newton iteration on the polynomial $\Phi_2(X, J_{i+1})$, with

$$\Phi_2(X,Y) \quad = \quad X^3 + Y^3 - X^2Y^2 + 1488\left(XY^2 + X^2Y\right) - 162000\left(X^2 + Y^2\right)$$

$$+ 40773375XY + 8748000000\,(X+Y) - 157464000000000.$$

The `Algorithm Lift_Previous_J_Invariant` computes coefficients $A, B, C \in \mathcal{R} \bmod 2^N$, such that

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

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 $J_0 \equiv j\,(\mathcal{E}_0) \mod 2^N$ then it suffices to start with $j\,(E_{N-1}) \equiv j\,(\mathcal{E}_{N-1}) \mod 2$ and iterate this algorithm $N-1$ times, which immediately leads to algorithm `Lift_First_J_Invariant`.

Algorithm 8 (`Lift_Previous_J_Invariant`)

Input. $J_{i+1} \in \mathcal{R} \mod 2^N$ with $J_{i+1} \equiv j\,(\mathcal{E}_{i+1}) \mod 2^{N-1}$ and a precision N.
Output. $J_i \in \mathcal{R} \mod 2^N$ with $J_i \equiv j\,(\mathcal{E}_i) \mod 2^N$.

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. $C \equiv J_{i+1}^3 - 162000J_{i+1}^2 + 8748000000J_{i+1} - 157464000000000 \mod 2^N$
4. $J_i = \text{Lift\_Previous\_J\_Invariant\_Rec}(J_{i+1}, A, B, C, N)$
5. **Return** J_i

Algorithm 9 (`Lift_Previous_J_Invariant_Rec`)

Input. 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.
Output. 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
   (a) $J_i = J_{i+1}^2 \mod 2$
2. Else
   (a) $N' = \lceil \frac{N}{2} \rceil$
   (b) $J_i = \text{Lift\_Previous\_J\_Invariant\_Rec}(J_{i+1}, A, B, C, N')$
   (c) $J_i \equiv J_i - \frac{J_i^3 + AJ_i^2 + BJ_i + C}{3J_i^2 + 2AJ_i + B} \mod 2^N$
3. **Return** J_i.

Algorithm 10 (`Lift_First_J_Invariant`)

Input. A j-invariant $j_0 \in \mathbb{F}_{2^n} \backslash \mathbb{F}_4$ and a precision N.
Output. $J_0 \in \mathcal{R} \mod 2^N$ with $J_0 \equiv j_0 \mod 2$, and $\Phi_2\left(J_0, \Sigma\left(J_0\right)\right) \equiv 0 \mod 2^N$.

1. $J_0 \equiv j_0^{2^{(n-N+1)}} \mod 2$
2. For $i = 2$ to N do
   (a) $J_0 = \text{Lift\_Previous\_J\_Invariant}(J_0, i)$
3. **Return** J_0.

6.2.2 Computing the Trace

In this section we give an explicit formula for the first coefficient c_i of the formal group expression of $\hat{\Sigma}_i$. This suffices to compute the trace of Frobenius t, since $t \equiv \prod_{i=0}^{n-1} c_i \mod q$.

Algorithm 11 (`Compute_Trace`)

Input. A j-invariant $j \in \mathbb{F}_{2^n} \backslash \mathbb{F}_4$ of an elliptic curve E.
Output. The trace of Frobenius $t = q + 1 - \#E\left(\mathbb{F}_q\right)$ of E.

1. $N = \lceil \frac{n}{2} \rceil + 13$, $M = N - 10$
2. $J = \text{Lift\_First\_J\_Invariant}(j, N)$
3. $CN = 1$, $CD = 1$
4. For $i = 0$ to $n - 1$ do
   (a) $J' = \text{Lift\_Previous\_J\_Invariant}(J, N)$
   (b) $Z = -\frac{\left(J^2+195120J+4095J'+660960000\right)/2^{12}}{(J^2+J(563760-512J')+372735J'+8981280000)/2^9} \mod 2^N$
   (c) $T = \left(12Z^2 + Z\right)\left(J' - 1728\right) - 36 \mod 2^M$
   (d) $CN = CN \times \left(J' - \left(504 + 12096Z\right)T\right) \mod 2^M$
   (e) $CD = CD \times \left(240T + J'\right) \mod 2^M$
   (f) $J = J'$
5. $t = \text{Sqrt}\left(CN/CD, 1, M\right) \mod 2^{M-1}$
6. If $t > 2\sqrt{q}$ then $t = t - 2^{M-1}$
7. **Return** t.

6.3 Elliptic Curve Digital Signature Algorithm

We implemented a small ECDSA generation/verification library, **Sig** over **EC**. As described in [5]:

Signatures schemes are the digital counterparts to handwritten signatures. They can be used to provide data origin authentication, data integrity, and non-repudiation. Signature schemes are commonly used by trusted certification authorities to sign certificates that bind together an entity and its public key.

ECDSA is the EC analogue of the Digital Signature Algorithm. It is the most widely standardized EC-based signature scheme.

Algorithm 12 ECDSA signature generation

Input. Domain parameters $D = (q, FR, S, a, b, P, n, h)$, private key d, msg m.
Output. *Signature* (r, s).

1. Select $k \in_R [1, n-1]$.
2. Compute $k\mathbf{P} = (x_1, y_1)$ and convert x_1 to an integer $\overline{x_1}$.
3. Compute $r = \overline{x_1} \mod n$. If $r = 0$ then goto step 1.
4. Compute $e = H(m)$.
5. Compute $s = k^{-1}(e + dr) \mod n$. If $s = 0$ then goto step 1.
6. **Return** *Signature* (r, s).

Algorithm 13 ECDSA signature verification

Input. $D = (q, FR, S, a, b, P, n, h)$, public key Q, message m, signature (r, s).
Output. Acceptance or rejection of the signature.

1. Verify that r and s are integers in $[1, n-1]$. If any verification fails ***Reject***.
2. Compute $e = H(m)$.
3. Compute $w = s - 1 \mod n$.
4. Compute $u_1 = ew \mod n$ and $u_2 = rw \mod n$.
5. Compute $X = u_1\mathbf{P} + u_2\mathbf{Q}$.
6. If $X = \infty$ then ***Reject***.
7. Convert x-coordinate x_1 of X to an integer $\overline{x_1}$, compute $v = \overline{x_1} \mod n$.
8. If $v = r$ then ***Accept***, otherwise ***Reject***.

Proof that signature verification works. If a signature (r, s) on a message m was indeed generated by the legitimate signer, then $s \equiv k^{-1}(e + dr) \mod n$. Rearranging yields,

$$k \equiv s^{-1}(d + dr) \equiv s^{-1}e + s^{-1}rd \equiv we + wrd \equiv u_1 + u_2 d\,. \qquad (\mathrm{mod}\ n)$$

7 Challenge

We were given a challenge by our *Supervisor*, ***Barukh Ziv***.

Description. Decipher a short message, encrypted using a simple EC El-Gamal scheme implemented over Binary Field $\mathbb{F}_{2^m}$, using supplied algorithms, where the *Secret Key* is q, the biggest prime of curve's order. Parameters are given in hex notation; MSBs correspond to higher degrees of polynomials over $\mathbb{F}_{2^m}$.

Modulus. $0x800000000000000000000000000000000000000000000000000000000000010a1$

Elliptic curve. $y^2 + xy = x^3 + a_6$

Elliptic curve point. ($0x3625a5fa7c12652ad01369f0f2c4cea3bc2fff7f3ad5b85069b7fed6120e92825640fe61$,
$0x699ed0cbc2127644dcfbbe67e593b3a697b8b65c958a4f05a3a0bd1c6d1e1c578494901$)

Secret key. q

Mapping. Message encryption requires mapping bytes of the message to points on elliptic curve. There are various mapping algorithms; the one used in this challenge is as follows:

1. Divide message to blocks, so every block (except the latest) has number of bytes in binary field $m - 1$. Example: If $m = 65$, then each block has 7 bytes.
2. Prepend a 0-byte to the block.
3. Treat the lower addresses of the block as LSBs of a big integer – this is the x-coordinate of a point on the elliptic curve.

When extracting bytes of the message from a decrypted message, the process is reversed.

Encrypted message. *Contains two blocks.*

[($0x6a401a7be22627040be02cf81eae0bacc2ccb1b884c2da791c0711d9fe1230b90f84bb7$,
$0x608a6e300f6985b4f68caacd7a5781a5d01cd2075f4d96d0227b724e0db17b0b272467b$)
($0x275e054d0729e8d2e2b404c03bba29af09bd6025af28bb77360cef538f8fb994d3f40f3$,
$0x699170da1d2307dfd6ba81b4e8949936b2d95c8cd28b175f1fa1fa589b697cb4200d551$)]

$$\begin{bmatrix} (& 0x523a89c03f0c57d48948ffd751b6e7bfd1c056c29bc9ff069b23cdfbb370966c8a35ce6 & , \\ & 0x4bd2186baf05fb1ffa56171b6b811d10c35cc89a718e382a207392b927400fd7e3d0714 &) \\ (& 0x64fff7d1d6885b422a94a441ffd01c03d130bc0639a1eaba5520a29a9ed4ff23024e49c1 & , \\ & 0x24e42b93d09c9f51d13768c06e8d80643c36e590589a909a1c0fa0c3cd714d204fd8f6d &) \end{bmatrix}$$

Problem.

1. Compute the order of the elliptic curve.
2. The encrypted message contains a link to an article. In this article a particular place in mentioned. What is its distance from equator (with resolution of 100 km)?

Solution. First we solve $y^2 + xy = x^3 + a_6$ for a_6 where x, y stand for the x, y-coordinates of the given elliptic curve point appropriately. Then, we create the elliptic curve with the acquired a_6 and issue the point counting algorithm to find its order, which is given by,

$$1554135113780583256735569525458815125313925838977170229483313945975643138936910107596 8.$$

Next we factor the result, getting

$$2^9 \cdot 17 \cdot 37 \cdot 819214448167029779 \cdot 589074961181371464517421775764263682462059570199 82857345274829,$$

Hence by definition, only to obtain the private key, d, the biggest prime of curve's order.

$$d = 589074961181371464517421775764263682462059570199 82857345274829.$$

We follow the decryption algorithm using EC point arithmetics on each block to compute M, and then extract the appropriate message from the blocks. Combining these blocks yields a URL linking to

`http://damninteresting.com/natures-nuclear-reactors`

The place mentioned in the linked article is **The State of Gabon in West Africa**, which is on the equator, hence its distance from the equator is 0 km.

Algorithm 14 Basic El-Gamal elliptic curve encryption.

INPUT: Domain parameters $(p = 2^m, E, \mathbf{P}, n = q)$, *public* key $\mathbf{Q}$, plaintext m.
OUTPUT: Cipher-text $(\mathbf{C_1}, \mathbf{C_2})$.

1. Represent the message m as a point $\mathbf{M}$ in $E(\mathbb{F}_p)$.
2. Select $k \in_{\text{Random}} [1, n-1]$.
3. Compute $\mathbf{C_1} = k\mathbf{P}$.
4. Compute $\mathbf{C_2} = \mathbf{M} + k\mathbf{Q}$.
5. Return $(\mathbf{C_1}, \mathbf{C_2})$.

Algorithm 15 Basic El-Gamal elliptic curve decryption.

Input: Elliptic curve domain parameters $(p = 2^m, E, \mathbf{P}, n = q)$, *private* key d, Cipher-text $(\mathbf{C_1}, \mathbf{C_2})$.
Output: Plaintext m.

1. Compute $\mathbf{M} = \mathbf{C_2} - d\mathbf{C_1}$.
2. Extract m from $\mathbf{M}$.
3. Return m.

In our case. The following holds: order $(P) = q$ and $\mathbf{Q} = d\mathbf{P}$.

Part IV. Performance

The testing had been run on an Intel Dual Core i7 2.66 GHz (L2 cache per core: 256 KB, L3 cache: 4 MB) with 8GB of RAM.

Every result is the average per one task (10^5 tasks total)			
field bits	addition	doubling	random element $*$ point
127	$0.68\mu s$	$0.22\mu s$	$0.0741ms$
239	$0.82\mu s$	$0.29\mu s$	$0.1802ms$
271	$0.98\mu s$	$0.35\mu s$	$0.2535ms$
487	$1.30\mu s$	$0.47\mu s$	$0.5965ms$

EC Arithmetics.

	Point counting	Finding a point	
field bits	per one curve	in random	of order q
127	$3.91s$	$25.8\mu s$	$0.224ms$
239	$19.15s$	$54.9\mu s$	$1.853ms$
271	$35.31s$	$66.5\mu s$	$1.694ms$
487	$119.95s$	$161.7\mu s$	$6.297ms$

Point Counting.

Field bits	FGH	Skjernaa	Vercauteren	Our version
144	$3.99s$	$2.89s$	$1.91s$	$6.93s$
240	$16.5s$	$17.7s$	$10.9s$	$18.99s$
288	$38.6s$	$27.8s$	$17.6s$	$37.48s$
480	$205.3s$	$157.9s$	$106.5s$	$107.8s$

Point Couting Comparison.

Field bits	Finding a curve	Average attempts
127	$20.05s$	5.13
144	$22.14s$	3.19
239	$125.83s$	6.57
271	$295.39s$	8.37

Finding a cryptographically-strong curve *(not parallelized)*.

Approval of a curve. Let $\#E = qm$, the number of points of the cryptographically-strong curve that we are trying to find. We extract the large prime number q from $\#E$ by finding m, which is of the form $\prod m_i^{\alpha_i}$ where m_i are the first 10,000 prime numbers; otherwise - we discard the curve.

Parallelism. To find a cryptographically good curve, one simply calls the program using **GNU Parallel**, running the same program across all CPUs; the first program to find a good curve returns it with a failing value, causing all other jobs to terminate immediately, when using the correct set of flags to **GNU Parallel**. The **NTL** library is not thread-safe nor re-entrant hence the separate process parallelism.

The workflow. The control flow is piped from one process to another due to limitations of the NTL library, and is given by,

GNU Parallel is used to spawn processes as much as there are CPUs available to find a cryptographically good curve.

↓

The process which finishes first terminates all others and returns the curve's a_6 and *order* to the supplied shell script.

↓

The user's program must deliver the two std::string program parameters a_6 and *order* to the constructor of **EC**.

In practice the time required to process the different random curves is halved by roughly the number of CPUs involved.

ECDSA. *Results are average measurements per task.*
NOTE: *According to prebuffering size, after generating #stacksize signatures, refilling takes place.*

Field bits	Prebuffering		Generation		Verification	
127	fill stack	time	fill stack	time	fill stack	time
	1	0.663ms	1	0.673ms	1	1.318ms
	10	6.536ms	10	0.655ms	10	1.306ms
	100	65.833ms	100	0.660ms	100	1.344ms
	1000	658.147ms	1000	0.659ms	1000	1.283ms
	10000	6.648s	10000	0.666ms	10000	1.315ms
239	fill stack	time	fill stack	time	fill stack	time
	1	1.741ms	1	1.744ms	1	3.451ms
	10	17.249ms	10	1.727ms	10	3.437ms
	100	167.741ms	100	1.679ms	100	3.099ms
	1000	1.672s	1000	1.673ms	1000	3.090ms
	10000	17.399s	10000	1.673ms	10000	3.055ms

Part V. Future work

8 Possible goals

- Improve our implementation to get nicer time results, which would be closer to known implementation time results, for the cases of small field sizes.
- Implement additional point counting algorithms such as AGM, Harley; to be compared vs. the algorithm we have implemented.
- Create an easy-to-use parallelism interface for use within a C++ program, as currently the control flow is piped from one process to another due to limitations of the NTL library *(non re-entrant nor thread-safe)* as described in **Performance** ▷ **Parallelism**.
- Comparison between our implementation over Binary Fields and others over Large Prime Fields of greater characteristic.

9 What we have learnt

We have acquired theoretical as well as practical mathematical knowledge for developing, discovering and learning computer security topics, which always has been the "hot" subject in computer science ever since it came into awareness more and interest for security professionals.

Also, we developed a passion for getting involved and perhaps influencing the cryptography community in the future with new visions and ideas; hence becoming a "hot" subject in our eyes as well, before entering the Industry.

Part VI. Bibliography

1. Ron M. Roth - ***Introduction to Coding Theory*** (2007).

2. Frederik Vercauteren - ***Computing zero functions of curves over finite fields*** (2003).

3. IEEE - ***IEEE P1363 / D13 (Draft Version 13). Standard Specifications for Public Key Cryptography, Annex A, Number-Theoretic Background*** (1999).

4. Tanja Lange - ***A note on L´opez-Dahab coordinates*** (2006).

5. Darrel Hankerson, Alfred Menezes, Scott Vanstone - ***Guide to Elliptic Curve Cryptography*** (2003).

6. Mireille Fouquet, Pierrick Gaudry, Robert Harley - ***An extension of Satoh's algorithm and its implementation*** (2000).

7. Elisabeth Oswald - ***Introduction to Elliptic Curve Cryptography*** (2005).

8. Frederik Vercauteren, Bart Preneel, Joos Vandewalle - ***A Memory Efficient Version of Satoh's Algorithm*** (2001).