General Attacks on Elliptic Curve Based Cryptosystems
by Merabi Chicvashvili and Ron Ryvchin
Spring 2014
Project advisor: Barukh Ziv
Introduction

In this project we aim to solve the ECDLP (Elliptic Curve Discrete Logarithm Problem) and optimize the running time of the solution. Elliptic Curve based cryptosystems are becoming increasingly popular these days because they provide a high security level with significantly smaller keys than is possible with their RSA counterparts. The security of this cryptosystem is based on the hardness of the ECDLP. The connection to the ECDLP will be explained in details later in this document.

In this project we had two general purposes:

1) Solving 64-bit challenges faster than the previous record.
2) Solving up to 70-bit challenges.

In addition, we had a special challenge which gives a perfect example for an unsecured elliptic curve, this we also be discussed in more details in the next sections.
Background

Elliptic Curves
Elliptic curve basic equation is as follows: \( y^2 = x^3 + a \cdot x + b \), which creates this kind of looking graphs:

In our project we will focus on elliptic curves and points over prime fields. An elliptic curve over prime field \( F \) is described by the following equation:
\[
y^2 + a \cdot x y + b \cdot y = x^3 + c \cdot x^2 + d \cdot x + e \quad \text{such that: } a, b, c, d, e \in F
\]
The examples above are elliptic curves over \( R \).
The set of points \((x, y) \in F\) which are legal solutions to given elliptic curve form a group structure with a special point called ‘point at infinity’ (\( \infty \)) as the unit member of the group. This group is what Elliptic Curve cryptosystem rely on.

Using Elliptic Curves for Encryption
We will describe a schematic algorithm for using elliptic curves over prime fields for encryption:

- Alice will choose a secure elliptic curve over \( F_p \) \( s.t \) \( p \) is a prime number, the elliptic curve will be denoted by \( E(F_p) \).
- Alice will choose a point \( P = (x, y) \) on the curve, The point order will be denoted as \( n \).
- Alice will choose a random number \( l \in [0, n - 1] \) for her private key.
- Alice’s public key is \( Q = l \cdot P \mod p \).
Lets say now that Bob is willing to send Alice a message, the domain of the algorithm is: $a, b, p, P$ and $Q$ is the public key.

- Bob will represent his message as a point on the curve that will be denoted by $R$.
- Bob will choose a random number $m$ and perform these multiplications: $m \cdot P$, $m \cdot Q$.
- Bob will send the next messages: $M_1 = m \cdot P$, $M_2 = R + m \cdot Q$.

Alice is going decipher the message by performing a few arithmetic operations:

- $M_1 \cdot l = (m \cdot P) \cdot l = m \cdot P \cdot m \cdot Q$
- $M_2 - m \cdot Q = R + mQ - mQ = R$

Hence $l$ is Alice’s private key and that’s our aim, to find out what’s Alice’s private key.

**ECDLP**

We are looking for Alice’s private key, but it’s not that simple. Finding the private key means solving the following equation: $Q = l \cdot P \mod p$.

In the Discrete Logarithm Problem we are given a group $G$, a group member $g \in G$, and a group member $h \in <G>$ such that $h = g^l$. The problem of finding $l$ is called the DLP. In a similar way finding $l$ for $Q = l \cdot P$ is the ECDLP.

**Attacking Elliptic Curves**

The naive algorithm for attacking an elliptic curve is to find distinct pairs $(c',d'), (c'',d'')$ of integers modulo $n$ such that:

$c' \cdot P + d' \cdot Q = c'' \cdot P + d'' \cdot Q \Rightarrow (c' - c'') \cdot P = (d'' - d') \cdot Q \Rightarrow (c' - c'') \cdot P = (d'' - d') \cdot lP$

$\Rightarrow l = (c' - c'') \cdot (d'' - d')^{-1} \mod n$

A naive method for finding such pairs $(c',d')$, $(c'',d'')$ is to select random integers $c, d \in [0,n-1]$ and to store the following triple in a table: $[c, d, c \cdot P + d \cdot Q]$, repeating this process until a collision is obtained. The birthday paradox says that
we need $\sqrt{\pi n}/2$ iterations until a collision is obtained. The drawback of the algorithm is the storage that is required for this process: $\sqrt{\pi n}/2$ triples.

**Advanced Algorithms**

**Pollard’s Rho Method**

Pollard suggested to solve the memory problem by iterating a random function $f$ over the group $\langle P \rangle$. Starting from a random point $R_0 = c \cdot P + d \cdot Q$ where $c, d \in \mathbb{Z}[0, n-1]$ (random integers modulo $n$), one computes $R_{i+1} = f(R_i)$ iteratively. The following graph is built using all starting points $R \in \langle P \rangle$:

![Graph of Pollard’s Rho Method](image)

Such a sequence consists of a tail of length $\mu$ and a cycle of length $\lambda$, thus forming a rho of length $\rho = \mu + \lambda$. It can be shown that the average tail length ($\mu$) and the average cycle length ($\lambda$), should be $\sqrt{\pi n}/8$, thus giving a rho length of about $\sqrt{\pi n}/2$.

These results say that after about $\sqrt{\pi n}/2$ iterations, we will find a collision, two pairs of integers modulo $n$, $(c', d'), (c'', d'')$ such that:

$c' \cdot P + d' \cdot Q = c'' \cdot P + d'' \cdot Q \Rightarrow (c' - c'') \cdot P = (d'' - d') \cdot Q \Rightarrow (c' - c'') \cdot P = (d'' - d') \cdot lP \Rightarrow l = (c' - c'') \cdot (d'' - d')^{-1} \mod n$

This way the private key $l$ can be deduced if $d' \neq d''$, which happens with probability $1 - \frac{1}{n}$.

**Choosing Iteration Function**
We chose the function $f$ to be an additive random walk. We pre-computed $r$ random points $T_j \ 0 \leq j < r \in \mathbb{P}$ such that $T_j = c_j \cdot P + d_j \cdot Q$ and defined the function as follow: $f(R_i) = R_i + T_j$ where $j = H(R_i)$ with $H$ a partition-function $H : \mathbb{P} \rightarrow \{0, 1, ..., r - 1\}$.

This creates an additive walk in $\mathbb{Z}/n\mathbb{Z}$ since each point $R_i$ can be written as $R_i = c_i \cdot P + d_i \cdot Q$ for which:

- $c_{i+1} = c_i + c_j \mod n$
- $d_{i+1} = d_i + d_j \mod n$

Satler and Schnorr have shown that the above approach is sufficiently random for $r > 8$. Teske has found experimentally that a value of $r \geq 20$ is more convenient, that’s why we chose $r$ to be 32.

**Pohlig-Hellman Attack**

The Pohlig-Hellman algorithm efficiently reduces the computation of $l = \log_P Q$ to the computation of discrete logarithms in the prime order subgroups of $\langle P \rangle$. It follows that the elliptic curve of order $n$ is only as strong as the highest prime factor of $n$.

Suppose that the prime factorization of $n$ is: $n = p_1^{e_1}p_2^{e_2}...p_r^{e_r}$. Our strategy is to compute $l_i = l \mod p_i^{e_i} \ 1 \leq i \leq r$ and solve the system of congruences:

- $l = l_1 \mod p_1^{e_1}$
- $l = l_2 \mod p_2^{e_2}$
- $\ldots$
- $l = l_r \mod p_r^{e_r}$

There is a unique way for writing each $l_i$: $l_i = z_0 + z_1 \cdot p_i + z_2 \cdot p_i^2 + \ldots + z_{e_i-1} \cdot p_i^{e_i-1}$, for each $z_i$ the digits $z_0, z_1, ..., z_{e_i-1}$ are computed one at a time as follows:
We first compute $P_0 = \frac{n}{p_i} P$ and since the order of $P_0$ is $p_i$, we have $Q_0 = \frac{n}{p_i} Q = l \cdot \frac{n}{p_i} P = l \cdot P = z_0 \cdot P$, hence $z_0 = \log_{P_0} Q_0$ can be obtained by solving an ECDLP in $\langle P_0 \rangle$. Next we compute $Q_1 = \frac{n}{p_i^2} (Q_0 - z_0 P)$ thus we have:

$$Q_1 = \frac{n}{p_i^2} (Q_0 - z_0 P) = \frac{n}{p_i^2} (l - z_0) P = (l - z_0) \frac{n}{p_i^2} P = (z_0 + z_1 \cdot p_i - z_0) \left( \frac{n}{p_i^2} \cdot P \right) = z_1 \left( \frac{n}{p_i^2} P \right) = z_1 \cdot P_0^2$$

hence $z_1 = \log_{P_0} Q_1$ and again it can be obtained by solving an ECDLP in $\langle P_0 \rangle$.

The general form is as follows (after computing $z_0, z_1, ..., z_{t-1}$):

$$Q_t = \frac{n}{p_i^t} \cdot (Q - z_0 P - z_1 p_i P - z_2 p_i^2 P - ... - z_{t-1} p_i^{t-1} P)$$

After finding each $l_i$ we solve the congruences (The Chinese Remainder Theorem guarantees a unique solution) using Gauss Algorithm.

**Floyd’s Cycle-Detection Algorithm**

Floyd’s cycle-detection algorithm, also called the ‘tortoise and the hare’ algorithm, is a pointer algorithm which uses two pointers which move through sequence at different speeds. One pointer is pointing to $x_i$ and the other one is pointing to $x_{2i}$ ($x_i, x_{2i}$ are elements in the sequence), at each step of the algorithm it increases $i$ by one, which means moving the tortoise by one step and the hare 2 steps. At each step we perform an equality test, the first value of $i$ which the two pointers have equal values will detect the cycle.

The algorithm’s running time complexity is $O(\mu + \lambda)$ which $\mu$ is the Pollard’s rho tail length and $\lambda$ is the cycle length.

**Brent’s Cycle-Detection Algorithm**

Brent’s cycle-detection algorithm requires also 2 pointers to the sequence, however it is based on a different principle: searching for the smallest power of 2 that is larger than both $\mu, \lambda$ (the tail and the circle length). The algorithm compares $x_{2i-1}$ with each subsequent sequence value up to the next power of 2, and it stops when it finds a match. This method finds the circle length correctly and there is less function evaluation in each step. The running time complexity is
\( O(\mu + \lambda) \) too, but it can be shown that the number of function evaluations (point additions in our case) can never be higher than for Floyd’s algorithm. According to Brent, his algorithm speeds up the Pollard’s rho attack by 24%.

**Jacobian Coordinates**

Jacobian coordinates are used to represent elliptic curve points on prime curves \( y^2 = x^3 + a \cdot x + b \). They allow us to perform Curve Arithmetics without the use of a field inverse. In Jacobian coordinates the triple \((X, Y, Z)\) represents the affine point \((\frac{X}{Z}, \frac{Y}{Z})\)

**Montgomery Inverse**

The Montgomery inverse of an integer \( a \in [1, p - 1] \) is \( b \) such that
\[
b = a^{-1} \cdot 2^n \mod p \text{ where } p \text{ is prime and } n \text{ is the number of bits of } p.
\]

The computation of Montgomery inverse is divided into two phases. In the first phase we compute the Almost Montgomery Inverse \( a^{-1} \cdot 2^k \mod p \), which is calculated in \( k \) iterations, where \( k \) depends on input values. In the second phase we can either compute the classical inverse, or the Montgomery inverse \( a^{-1} \cdot 2^n \mod p \), by repetitive halvings or a single additional multiplication (in ECDLP context). This is done by pre-computing all the inverses of powers of 2 up to \( n \), \( n \leq 70 \) in our challenges.

The algorithm is as follows:
Phase I
Input: $a \in [1, p - 1]$ and $p$
Output: $y \in [1, p - 1]$ and $k$, where $y = a^{-1}2^k \pmod{p}$
and $n \leq k \leq 2n$
1. $u := p, v := a, r := 0, s := 1$
2. $k := 0$
3. while $(v > 0)$
4. if $(u$ is even) then
5. $u := u/2, s := 2s, k := k + 1$
6. else (if $v$ even) then
7. $v := v/2, r := 2r, k := k + 1$
8. else
9. $x := (u - v)$
10. if $(x > 0)$ then
11. $u := x/2, r := r + s, s := 2s, k := k + 1$
12. else
13. $v := -x/2, s := r + s, r := 2r, k := k + 1$
14. if $(r > p)$ then
15. $r := r - p$
16. return $y := p - r$ and $k$.

Phase II
Input: $y \in [1, p - 1], p$ and $k$ from Phase I
Output: $r \in [1, p - 1]$, where $r = a^{-1} \pmod{p}$, and $2k$ from Phase I
17. for $(i = 1$ to $k)$ do
18. if $(r$ is even) then
19. $r := r/2$
20. else
21. $r := (r + p)/2$
22. return $r$ and $2k$.

Implementation
General
The entire code was written in c++ using NTL library for big integers operations. We began with setting up the NTL library in our project and running some tests to understand and master the interface of the library. Next we focused on writing a library which allow to make arithmetic operations over an elliptic curve. The domain parameters of the curve are saved as class variables meaning that for every curve we expect different results for every arithmetic operation, which means we assure generic behavior. After implementing the elliptic curve arithmetic library we finally could start on implementing the basic attack algorithm that we chose to work with- Pollard’s Rho Attack (using Floyd’s cycle detection algorithm). After implementing, testing and debugging Pollard’s Rho Attack we started implementing optimized methods and tried various techniques as discussed in the Optimizations section. The final step was solving the special challenge which could be solved efficiently by the Pohlig-Hellman reduction, this method was the last one we implemented.

Optimizations
We implemented various methods for optimizing the running time of the attack but not all of them are in our final version.

Point representation
We implemented the elliptic curve arithmetic library in a way that each method could work with Affine coordinates or Jacobian coordinates (and also mixed coordinates - e.g. Affine-Jacobian addition). In the final version we work only with Affine coordinates, simply because we are getting the best results in this representation.
The reason for that is that while using Jacobian coordinates, the partition function in the main loop must return the same value for different representations of the same point. We could not find any way to map different representations of the same point to the same value without using modular inverse. We tried to use different partition functions, which did not work. Then we tried to use Montgomery inverse, but NTL implementation of the ordinary inverse was faster.
That’s why, although the point arithmetic was much faster, it costed us in using a modular inverse in the partition function, so overall it did not speed up the whole attack (and even slowed it down because there were more field additions and multiplications).

Cycle Detection
We implemented both Floyd’s method and Brent’s method and compared the running time. As expected, Brent’s method gives the best results.

‘middle’ point optimization
Tail and cycle of the rho shape should have about the same length. However, when one of them is much longer than the other, Brent’s algorithm can be significantly improved. For example, if the tail is very long, we will enter the cycle with large power of 2 and do several full rotations. Only after we stop inside the cycle for the first time, the second pointer will complete the attack on the next iteration. When the cycle is very large, we spend several iterations on the cycle until we reach the power of 2 that is larger than the cycle’s length. Our idea was to keep an extra point that is also used to check collision. The point is the middle point, i.e. if we do $2^i$ steps, we keep the point after $2^{i-1}$ steps. During the first $2^{i-1}$ steps we compare the moving pointer to the point where the previous iteration finished, and to the middle point of the previous iteration. This can save $2^{i-2}$ steps for large cycles, which is about 1/8th and up to 1/4th of the total number of point additions. During the second $2^{i-1}$ steps we compare the moving pointer to the middle point of the current iteration, which can help with small cycles in case where the last iteration stopped on the tail. In our experiments, collision in middle point help at least one run of attack on each challenge. The time speedup due to this optimization ranges between 14%-24%.
For conclusion, we are using Pollard’s Rho Attack with Brent’s cycle detection, and our improvement. The elliptic curve arithmetics is done using Affine coordinates. In case that the order of a point is not prime we are using Pohlig-Hellman’s attack.

**Challenges and Results**

The following notation used:
- p - prime field modulus
- EC - elliptic curve coefficients (a,b)
- N - elliptic curve order
- P,Q - points on elliptic curve

**Challenge 1 - 30 bits**
- p = 754526683
- EC = (72537189, 706168557)
- N = 754564781
- P = (391592639, 187105396)
- Q = (699400157, 82294806)
- result: private key (l) - 363877079
- minimum time - 0.047 sec (24176 function calls)
- maximum time - 0.188 sec (77424 function calls)
- average time - 0.82 sec (37974 function calls)

**Challenge 2 - 40 bits**
- p = 1018754028791
- EC = (871520218049, 1007486607871)
- N = 1018752667981
- P = (215868580937, 253947951840)
- Q = (283497910998, 322669501418)
- result: private key - 538469048685
- minimum time - 0.969 sec (365389 function calls)
- maximum time - 5.875 sec (2200397 function calls)
average time - 3.2829 sec (1125607 function calls)

**Challenge 3 - 50 bits**

p = 862310130873029  
EC = (753664604275938, 359867061355737)  
N = 862310073862651  
P = (396159790871760, 299026752948778)  
Q = (594219074514457, 329535619534883)  
result- private key - 585566467573708  
minimum time - 17.922 sec (5471207 function calls)  
maximum time - 197.485 sec (54876055 function calls)  
average time - 82.0063 sec (25661003 function calls)

**Challenge 4 - 64 bits**

p = 17778887349362591359  
EC = (953276788483020, 17350004542596651438)  
N = 17778887341958652103  
P = (2072463837547420019, 13457440224317410775)  
Q = (14290627919077884680, 8330780035610710830)  
result- private key - 10341205411675263072  
minimum time - 2506.81 sec ~ 46 min (436215366 function calls)  
maximum time - 23916.5 sec ~ 6.5 hours (6261487497 function calls)  
average time - 14034.122 sec ~ 3 hours 53 minutes (3499239651 function calls)

**Challenge 5 - 70 bits**

p = 1015081100275112877913  
N = 1015081100220367148209  
P = (439822130273169935564, 47451178375742709665)  
Q = (24776174763769395364, 116622655754568570415)  
The EC parameters were deduced by the two points on the curve (linear equation system).  
result- private key - 928603898983145397169  
minimum time - 17877.5 sec ~ 4 hours 57 minutes (4924092173 function calls)
maximum time - 328611 sec ~ 3.8 days (90946847050 function calls)
average time - 140758.233 sec ~ 39 hours 5 minutes (38666966411 function calls)

**Special Challenge**

Given an elliptic curve \( y^2 = x^3 + a \cdot x + b \) over prime field with
\[ p = 414507122857381699247. \]
The following point was chosen on this curve:
\[ P = (215672232155085007005, 176420948314972445409) \]
A secret key \( k \) was chosen, and the point \( Q = kP \) was computed:
\[ Q = (49818534942346740253, 67908233076804365605) \]
Find \( k \).

The purpose of the challenge was to test Pohlig-Hellman reduction.
The EC parameters were deduced by the two points on the curve.
To find \( n \) we used a hint that for some curves \( n \) divides \( p^k - 1 \) for some very small \( k \). We factorize \( p^k - 1 \) for \( k=1, 2, \ldots \) and try to see if a product of any subset of prime factors can be the order of the group. We found
\[ n=309336737741654472 \] for \( k=2 \). Since the order is not a prime number, we applied Pohlig-Hellman reduction on this challenge.
**result** - private key - \( k = 1815014470915166467 \).

Although \( n \) is large, its largest prime factor is 28202267, and the whole attack (including factorization and solving the modular equations system) finished in about 3 minutes.

**Bibliography**
[1] V. Shoup, "NTL: A Library for doing Number Theory"
http://www.shoup.net/ntl/