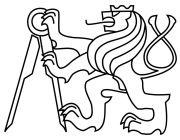
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Babystep-Giantstep Algorithm and Solution of Elliptic Curve Discrete Logarithm Problem
Student:	Bc. Martin Holec
Supervisor:	Ing. Ivo Petr, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2018/19

Instructions

One of the standard methods of solution of the discrete logarithm problem (DLP) in a group of points of an elliptic curve is the Babystep-Giantstep algorithm (BSGS). The aim of this work is to investigate the state-of-the-art of the algorithm.

In particular, the student will
- investigate various variants and adjustments of BSGS (basic version, Grumpy giants, interleaving, etc.) and give their thorough description
- implement these variants in a suitable language
- compare average numbers of operations necessary to obtain a collision.

The comparison will be done using randomly generated instances of DLP in groups of moderate prime order and, besides elliptic curves given by Weierstrass equation, will also take into account Edwards and Montgomery elliptic curves.

References

[1] Steven D. Galbraith, Ping Wang and Fangguo Zhang. Computing Elliptic Curve Discrete Logarithms with Improved Baby-step Giant-step Algorithm. Cryptology ePrint Archive, Report 2015/605, 2015

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 16, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Applied Mathematics

Master's thesis

Baby-step Giant-step Algorithm and Solution of Elliptic Curve Discrete Logarithm Problem

Bc. Martin Holec

Supervisor: Ing. Ivo Petr, Ph.D.

29th June 2018

Acknowledgements

I would like to sincerely thank my supervisor Ing. Ivo Petr, Ph.D. for his willing help with my obstacles during the creation of this thesis. I would also like to thank my girlfriend Kateřina and friends Ivka and Terka for their persistent support and especially my brother for sharing our frustration about our theses.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 29th June 2018 .....

Czech Technical University in Prague
Faculty of Information Technology
© 2018 Martin Holec. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Holec, Martin. *Baby-step Giant-step Algorithm and Solution of Elliptic Curve Discrete Logarithm Problem.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Jedna ze standardních metod řešení problému diskrétního logaritmu na grupě bodů eliptické křivky je Baby-step Giant-step algoritmus (BSGS). Existuje mnoho variant algoritmu a tato práce se zaměřuje na prozkoumání těch nejaktuálnějších. Vygeneroval jsem náhodné křivky ve Weierstrassově, Edwardsově a Montgomeryho formě a porovnával jejich výkon v několika různých variantách BSGS. Došel jsem k závěru, že některé parametry eliptických křivek a forma křivek mají zásadní vliv na výkon jednotlivých algoritmů.

Klíčová slova problém diskrétního logaritmu, eliptická křivka, baby-step giant-step, grumpy giants, generování eliptické křivky, Montgomeryho křivka, Edwardsova křivka

Abstract

One of the standard methods of solution of the discrete logarithm problem (DLP) in a group of points of an elliptic curve is the Baby-Step Giant-Step algorithm (BSGS). There are many variants of this algorithm and this work aims to investigate the state-of-the-art of the algorithm. I generated random elliptic curves in Weierstrass, Edwards and Montgomery form and compared their performance over different variants of BSGS. I came to a conclusion that

some properties of elliptic curves and the form of the elliptic curve got a huge impact on the performance of the algorithms.

Keywords discrete logarithm problem, elliptic curve, baby-step giant-step, grumpy giants, elliptic curve generation, Montgomery curve, Edwards curve

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We live in the world full of secrets. People want to keep their privacy and their secrets really secret so they rely on modern cryptography heavily. These days, elliptic curve cryptography is more and more popular. To find out how does cryptography on elliptic curves exactly work in real life I decided to choose generic concept of discrete logarithm problem (DLP) and examine it on the Weierstrass, Montgomery and Edwards form of elliptic curves. This thesis focuses on performance of Baby-step Giant-step algorithm and various improvements and the impact of form of elliptic curve on a performance. Various forms are chosen to demonstrate whether there is an influence on the performance or not. At the end I brought a new view on the properties of elliptic curves based on cofactor of the elliptic curve and demonstrated the huge impact of this property on a performance of algorithms solving DLP on the elliptic curve.

Chapter 2

Discrete Logarithm Problem

Similarly to logarithm in real numbers we can define powers b^x for every integer x. Discrete logarithm $log_b(y)$ is an integer x satisfying $b^x = y$ so DLP is inverse to exponential function. Discrete logarithm problem is generally solvable in exponential time and some algorithms to solve this problem exist, however none of them runs in polynomial time so they are still considered inefficient.

The problem can be described in formal way:

Definition 1 *[18] If G is a finite group, b is an element of G, and y is an element of G which is a power of b, then the discrete logarithm of y to the base b is any integer x such that $b^x = y$.*

And the Discrete Logarithm Problem is presented as following:

Definition 2 *[19] Given a prime p, a generator b of $\mathbb{Z}_p^*$, and an element $y \in \mathbb{Z}_p^*$, find the integer x, $0 \leq x \leq p-1$, such that $b^x \equiv y \pmod p$.*

It is important to note that in general, G is not required to be cyclic and b is not required to be a generator of G (in case of G being cyclic group). This is necessary to consider because it means that DLP is not solvable in some cases. On the other hand we can tell when the DLP is solvable if we know G is cyclic and we know order of the element α.

Theorem 1 *[19] If G is a cyclic group, b is an element of order N in G, and $y \in G$, then there exists an integer x such that $b^x = y$ if and only if $y^N = 1$.*

There are algorithms designed to solve generic discrete logarithm problem. Some of them are designed to be universal and some got serious advantages in specific scenarios.

The discrete logarithm is defined on common groups. However, from the security point of view the multiplicative groups of finite fields are more interesting. The most common usage of the discrete logarithm problem is over

multiplicative group of the finite field however there are more possibilities. Computation of large powers was the first historical attempt for a one-way function. Exponentiation is really cheap operation in terms of computational power and difficult to invert. DLP became one of pillars of the modern cryptography. It is foundation of the Diffie-Hellman key exchange and ElGamal cryptosystem. This work is focused on discrete logarithm problem on elliptic curves so we will focus on that.

2.1 Naive algorithm

The exhaustive search is the basic naive algorithm. Given the base of logarithm g we compute successive powers using the group operation till we finally find the β. The memory space is constant in this case however the algorithm requires $\mathcal{O}(N)$ operations where N is the order of group. It can be sufficient in demonstrative examples however the order of a group is usually large number, especially in cryptography.

Definition 3 *[7] The order of a group G, written $|G|$, is the cardinality of the set G, the number of elements in the group. This may be finite or infinite.*

2.2 Baby-step Giant-step

Baby-step Giant-step algorithm belongs to members of universal algorithms for solving DLP. It is based on time-memory trade-off. BSGS restricts exhaustive search to subset of the whole group G and exploits important observation. If $y = b^x$, then one can write $x = iM + j$, where $0 \leq i,\ j < M$, typically $M = \sqrt{N}$ where N is order of the group. Hence, $b^x = b^{iM} \cdot b^j$, which implies $b^{-iM} \cdot y = b^j$. At first we compute M values of b^j. Then we compute i steps of $y \cdot b^{-iM}$. Lets use additive notation for better understanding in the future. So we are trying to find x from equation $y = xb$ for $0 \leq x \leq N$ where N is order of group and $y, b \in G$.

The pseudocode is following:

Algorithm 1 Baby-step Giant-step

1: $M \leftarrow \lceil \sqrt{N} \rceil$
2: $b' \leftarrow Mb$
3: **for** $j = 0...M$ **do**
4: Save tuple (jb, j)
5: **end for**
6: **for** $i = 0...M$ **do**
7: Look for $y - ib'$ in saved tuples
8: **end for**

DLP is solved when collision between the first list (baby-steps) and second list (giant-steps) is found because $jb = y - ib'$. Thus we can say $x = iM + j$. It is important to save tuples by the second component in easily searchable structure, for example dictionary. We need the structure to be sorted to achieve look-up complexity $\sqrt{M}$. The whole algorithm needs storage for $2M$ elements - this means M group operations for baby-steps and M group operations for giant-steps. Using $\mathcal{O}$ notation we can say the algorithm needs $\mathcal{O}(M)$ group operations to construct baby-steps and giant-steps and another $\mathcal{O}(M)$ look-up operations to search for collision in these lists so the final running time is $\mathcal{O}(\sqrt{N})$.

This algorithm is deterministic and requires $2\sqrt{N}$ group operations in the worst case and $\frac{3}{2}\sqrt{N}$ group operations on average over uniformly random choices for y [13]. The Baby-step Giant-step has been studied heavily so it is not a big surprise researchers came with improvements. Pollard promoted computation of baby-steps and giant-steps in parallel with significant speed up - almost 17 percent. The average-case running time should be $\frac{4}{3}\sqrt{N}$ [23].

2.3 Pollard's rho method

In 1978 John Pollard presented his idea to solve DLP which is known as *Pollard rho algorithm for logarithms* [24]. It is randomized algorithm based on his *Pollard rho algorithm* which was introduced in 1975. This method comes with the advantage against baby-step giant-step algorithm because it requires almost no memory.

We start with some presumptions - G is a cyclic group with prime order N. We can make this presumption on thanks to the Pohlig-Hellman algorithm which we will discuss later. At first split group G into three roughly equally sized sets according to some property. The property should be easily testable because the function to test it will be used many times during the algorithm. Denote these sets S_1, S_2, S_3. Define a sequence of group elements $x_0, x_1, x_2, \ldots$ by x_0 as a identity element of the group and

$$x_{i+1} = f(x_i) \stackrel{\text{def}}{=} \begin{cases} \beta i x, x_i \in S_1 \\ 2x_i, x_i \in S_2 \\ \alpha + x_i, x_i \in S_3 \end{cases} \tag{2.1}$$

for $0 \leq i$. This sequence of group elements in turn defines two sequences of integers $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ satisfying $x_i = a_i \alpha b_i \beta$ for $0 \leq i : a_0 = 0, b_0 = 0$, and for $0 \leq i$:

$$a_{i+1} = g(x_i) \stackrel{\text{def}}{=} \begin{cases} a_i, x_i \in S_1 \\ 2a_i \bmod N, x_i \in S_2 \\ a_i + 1 \bmod N, x_i \in S_3 \end{cases} \tag{2.2}$$

and

$$b_{i+1} = h(x_i) \overset{\text{def}}{=} \begin{cases} b_i + 1 \bmod N, x_i \in S_1 \\ 2b_i \bmod N, x_i \in S_2 \\ b_i, x_i \in S_3 \end{cases} \tag{2.3}$$

Now we can use the Floyd's cycle-finding algorithm [19] to find the collision. If we arrive at the collision $\alpha^{a_i}\beta^{b_i} = \alpha^{a_{2i}}\beta^{b_{2i}}$, then $\beta^{b_i - b_{2i}} = \alpha^{a_{2i} - a_i}$. We can easily adjust the equation: $(b_i - b_{2i}) \cdot log_\alpha \beta \equiv (a_{2i} - a_i) (\text{mod } N)$. If $(b_i - b_{2i}$ is invertible then the equation is effectively solvable using Extended Euclidean algorithm.

Here is the pseudocode of Pollard's rho method:

Algorithm 2 Pollard's Rho Method

```
a_0 ← 0
b_0 ← 0
x_0 ← 1
for i = 1, ... do
    x_i ← f(x_{i-1})
    a_i ← g(x_{i-1})
    b_i ← h(x_{i-1})
    x_{2i} ← f(x_{2i-2})
    a_{2i} ← g(x_{2i-2})
    b_{2i} ← h(x_{2i-2})
    if x_i = x_{2i} then
        r ← b_i − b_{2i} mod N
        if r = 0 then
            return failure
        end if
        x ← r^{-1}(a_{2i} − a_i) mod N
        return x
    end if
end for
```

[15]

In case of failure the algorithm can be repeated by choosing different random integers a_0, b_0 in the interval $[1, N-1]$, and start with $x = \alpha^{a_0}\beta^{b_0}$.

The Pollard Rho Algorithm needs negligible amount of storage and the running time was heuristically estimated for roughly $\sqrt{N}$ group operations, however it is randomized algorithm so there is a probability of failure and therefore worse performance compared to baby-step giant-step.

2.4 Pohlig-Hellman Algorithm

The algorithm above was based on assumption that order of group is a prime number. In 1978 Pohlig and Hellman presented their approach to discrete logarithm problem by reducing it from DLP of group G to the subgroups of G. It takes advantage of the factorization of the group order. The necessary theorem used in this algorithm is called *Chinese remainder theorem.*

Theorem 2 (Chinese remainder theorem) *If the integers* n_1,n_2, ..., n_k *are pairwise relatively prime, then the system of simultaneous congruences*
$x \equiv a_1 (mod\ n_i)$
$x \equiv a_2 (mod\ n_2)$
$x \equiv a_k (mod\ n_k)$
$\vdots$
has a unique solution modulo $n = n_1 n_2 \dots n_k$.

When we have such a system we can use *Gauss's algorithm*[19]. The solution x to the simultaneous congruences in the CRT may be computed as

$$x = \sum_{i=1}^{k} a_i N_i M_i \bmod n \tag{2.4}$$

where $N_i = \frac{n}{n_i}$ and $M_i = N_i^{-1} \bmod n_i$. These computations can be performed in $\mathcal{O}((\log n)^2)$ bit operations.

Lets continue with the algorithm, Let $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$ be the prime factorization of n. If $x = log_\alpha \beta$, then the approach is to determine $x_i = x \bmod p_i^{e_i}$ for $1 \leq i \leq r$, and then use Gauss's algorithm to recover $x \bmod n$. Each integer x_i is determined by computing the digits $l_0, l_1, \dots, l_{e_i - 1}$ in turn of its p_i -ary representation: $x_i = l_0 + l_1 p_i + \dots + l_{e_i - 1} p_i^{e_i - 1}$ where $0 \leq l_j \leq p_i - 1$. These computations consist of solving DLP over group with prime order. [19]

Algorithm 3 Pohlig-Hellman Algorithm

Find the prime factorization of $n : n = p_1^{e_1}, p_2^{e_2}, \ldots, p_r^{e_r}$, where $e_i \geq 1$.
for i = 1 ... r **do**
  $q \leftarrow p_i$
  $e \leftarrow e_i$
  $\gamma \leftarrow 1$
  $l_{-1} \leftarrow 0$
  $\bar{\alpha} \leftarrow \alpha^{n/q}$
  for j = 0 ... e-1 **do**
    $\gamma \leftarrow \gamma\alpha^{l_{j-1}q^{j-1}}$
    $\bar{\beta} \leftarrow (\beta\gamma^{-1})^{n/q^{j+1}}$
    $l_j \leftarrow log_{\bar{\alpha}}\bar{\beta}$
  end for
  $x_i \leftarrow l_0 + l_1 q + \cdots + l_{e-1}q^{e-1}$
  Use Gauss's algorithm to compute integer x
  $x, 0 \leq x \leq n-1$ such that $x \equiv x_i(\bmod p_i^{e_i})$ for $1 \leq i \leq r$
end for
return x

2.5 Index Calculus

The oldest and the most powerful version of DLP solving algorithms is called *Index Calculus*. Unfortunately not usable for every group however more effective when it is possible to use it - it often runs as subexponential-time algorithm. The algorithm begins with choosing the subset S of elements of G. The subset S is called *factor base*. It should contain elements of G so the majority of G elements can be expressed as products of elements of S. The ways to choose elements of the S belonging there and factorization using elements of S vary and are nontrivial so we will not discuss it here however usually we choose irreducible polynomials or primes. Then we choose number l at random, $0 < l < N$ and compute α^l. When α^l can be factorized in factor base, find $c_i, i = 1, \ldots, t$, for $t = |S|$, such that:

$$\alpha^l = \prod_{i=1}^{t} p_i^{c_i}, c_i \geq 0 \tag{2.5}$$

Lets compute logarithm of both sides and you get:

$$l \equiv \sum_{i=1}^{t} c_i \log_g p_i \pmod{N} \tag{2.6}$$

If it is not possible to compute logarithms above choose different l. Repeat these steps for different values of l until you got t linearly independent equations to create solvable set of linear congruences. Solve these congruences.

Choose number k at random, $0 < k < N$ and compute hg^{-k}. Find such $d_i, i = 1, \ldots, t$ that:

$$\beta\alpha^{-k} = \prod_{i=1}^{t} p_i^{d_i}, d_{\geq}0 \tag{2.7}$$

If we cannot get such a d_i, choose different k. Compute logarithm of both sides:

$$\log_g\beta \equiv \sum_{i=1}^{t} d_i \log_g p_i + k \pmod{N} \tag{2.8}$$

Finally, Kalvoda, Petr and Starosta [15] estimated the complexity. We present it using the *L-notation*.

$$L_q[\alpha, c] := \exp(c(\ln q)^{\alpha}(\ln \ln q)^{1-\alpha}) \tag{2.9}$$

This estimation assumes an optimal selection of factor base. However, this algorithm runs only on multiplicative groups of finite fields. This means we cannot use them for solving DLP on elliptic curves which we will discuss in the next chapter.

Chapter 3

Elliptic Curves

These days elliptic curves became essential part of modern cryptography. Elliptic curve cryptography was introduced independently by Neal Koblitz in 1987 [17] and Victor Miller in 1985 [20]. As we mentioned before, elliptic curves are resistant to solving DLP using index calculus so this is one of many reasons to use them.

3.1 Elliptic Curves over general field

Generally speaking elliptic curve is a plane algebraic curve. Every elliptic curve can be represented using a *generalized Weierstrass form*. Then we can say elliptic curve E in generalized Weierstrass form is defined by:

Definition 4 *For field $\mathbb{F}$, elliptic curve $E(\mathbb{F})$ is a set of solutions $(x, y) \in \mathbb{F}$ of the equation:*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{3.1}$$

where coeficients $a_i \in \mathbb{F}$ together with the *point at infinity*[**?**]. The point at infinity is an artificial point labeled $\mathcal{O}$ which does not lie on the curve. It is made to meet requirements of the group. When the curve E is not singular, set of points of curve E with addition of point at infinity $\mathcal{O}$ creates group. Non-singular elliptic curve means the discriminant $\triangle$ is non-zero. The formula to compute discriminant is following:

- $b_2 = a_1^2 + 4a_2$
- $b_4 = 2a_4 + a_1a_3$
- $b_6 = a_3^2 + 4a_6$
- $b_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2$

- $\triangle = -b_2^2 b_8 - 8b_4^3 - 27b_4^2 + 9b_2 b_4 b_6$

For the case of field of characteristics $K \neq 3$ we will use simplified Weierstrass form:
$y^2 = x^3 + Ax + B$
where A, B are constants from the field. The discriminant is following:

$$\triangle = 4A^3 - 27B^2 \tag{3.2}$$

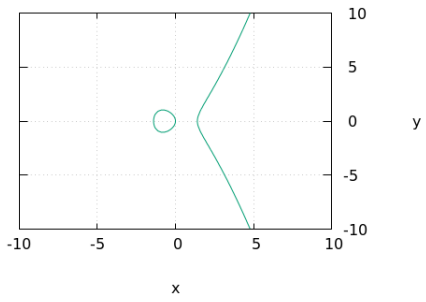


Figure 31: Non-singular elliptic curve $y^2 = x^3 - 2x$

The non-zero discriminant assures us that we got non-singular curve. Singular curves are special kind of elliptic curves and they will not be covered in this thesis.

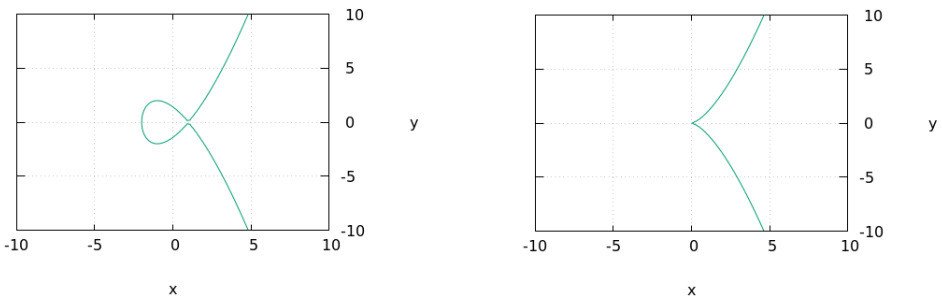


Figure 32: Singular elliptic curves $y^2 = x^3 - 2x$ and $y^2 = x^3$

First and foremost we need to define group operation using points $P, Q, R \in E$, inversion of point P denoted $-P$ and point at infinity $\mathcal{O}$:

1. $P + \mathcal{O} = \mathcal{O} + P = P, \forall P \in E$
2. $P + (-P) = \mathcal{O}, \forall P \in E$
3. $P + (Q + R) = (P + Q) + R, \forall P, Q, R \in E$
4. $P + Q = Q + P, \forall P, Q \in E$

Algebraically we can describe addition of two points using formulas. Inversion of a point P for subtraction is easy because the curve is symmetrical about x-axis. It means we can just change the sign of y-coordinate and get the inversion of the point. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two point on elliptic curve $E = y^2 = x^3 + Ax + B$. We achieved this form of equation by change of variables.[1]. The operations over elliptic curve are summarized in following formulas:

- $-P_1 = (x_1, -y_1)$
- $P_1 + P_2 = \mathcal{O}, P_1 \neq P_2, x_1 = x_2$
- $P_1 + P_2 = 2P_1 = \mathcal{O}, P_1 = P_2, y_1 = 0$
- $\lambda = \frac{y_2 - y_1}{x_2 - x_1}, \nu = \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}$
  $P_1 \neq P_2, x_1 \neq x_2$
- $\lambda = \frac{3x_1^2 + A}{2y_1}, \nu = \frac{-x^3 + Ax + 2B}{2y}$
  $P_1 = P_2, y_1 \neq 0$

and use it in equation:

$$P_1 + P_2 = (\lambda^2 - x_1 - x_2, -\lambda^3 + \lambda(x_1 + x_2) - \nu) \tag{3.3}$$

3.1.1 Montgomery Elliptic Curves

The Weierstrass form of elliptic curve is the most common however not the only one. In 1987 [21] Peter. L. Montgomery introduced his approach to elliptic curves.

Definition 5 *For field $\mathbb{F}$, elliptic curve $E(\mathbb{F})$ is a set of solutions $(x, y) \in \mathbb{F}$ of the equation:*

$$BY^2 = X^3 + AX^2 + X \tag{3.4}$$

where $A, B \in \mathbb{F}$ together with the *point at infinity* and satisfying the condition of non-singularity $B(A^2 - 4) \neq 0$.

Montgomery form of curves is not absolutely universal. The order of curve is always divisible by 4 so it means not every curve in Weierstrass form is transformable to the Montgomery form.

Theorem 3 (Suyama) *[12] If E is an elliptic curve given by a Montgomery model then $4|\#E(\mathbb{F}_q)$.*

This comes with restrictions for the cryptographical usage.

Definition 6 *Cofactor $k = \frac{\#E(G)}{g}$ where g is the order of the largest subgroup of G having the prime order.*

The cofactor of a cryptographically strong elliptic curve needs to be smaller or equal to 4 so the only suitable elliptic curve in Montgomery form is the one with cofactor exactly equal to 4.[1]

The operation $\oplus$ is defined in different way than in the case of Weierstrass form. Let $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ be a points on E^M. To compute $P_1 + P_2$ we use the following formulas[22]:
For $P_1 \neq P_2$ compute $\lambda = \frac{(y_2 - y_1)}{x_2 - x_1}$

- $x_3 = B\lambda^2 - A - x_1 - x_2$
- $y_3 = \lambda(x_1 - x_3) - y_1$

For $P_1 = P_2$ compute $\lambda = \frac{(3x_1^2 + 2Ax_1 + 1)}{2By_1}$

- $x_3 = B\lambda^2 - A - 2x_1$
- $y_3 = \lambda(x_1 - x_3) - y_1$

Lets discuss transformability of the curves. As we said before, not every elliptic curve in Weierstrass form can be represented in Montgomery form. Every Montgomery-form elliptic curve has the point $(0, 0)$ of the order 2[22]. We know there exist elliptic curves in the Weierstrass form which do not have any point of the order 2 so not every Weierstrass-form elliptic curve has got the Montgomery form. However when we can map point $(0, 0)$ of Montgomery-form curve to such a point of the order 2, we can transform the Weierstrass-form elliptic curve to the Montgomery form.

To say whether the curve $E(\mathbb{F}_p)$ in Weierstrass form is transformable or not we test two conditions:

- The equation $x^3 + Ax + B = 0$ has at least one root α in $\mathbb{F}_p$

[1] The complete list of conditions is in the chapter *Implementation*

- The root α is a quadratic residue in $\mathbb{F}_p$

The first condition refers to polynomial factorization over finite field so we can use Berlekamp's algorithm[3] or Cantor–Zassenhaus algorithm[8]. The second condition can be tested using *Legendre symbol*, however the first one is difficult to test in our case so we will focus on the opposite direction of transformation - from Montgomery form to the Weierstrass form. According to Okeya[22] *Any Montgomery-form elliptic curve is transformable to the Weierstrass-form elliptic curve.* Let $E^{A,B} : BY^2 = X^3 + AX^2 + X$ denotes elliptic curve in Montgomery form with parameters A and B. Then we set

- $a = \frac{1}{B^2} - 3(\frac{A}{3B})^2$
- $b = -(\frac{A}{3B})^3 - a\frac{A}{3B}$
- $E^{a,b} : y^2 = x^3 + ax + b$

Where $E^{a,b}$ stands for elliptic curve in Weierstrass form with parameters a and b.

3.1.2 Edwards Elliptic Curves

In 2007 Harold M. Edwards presented paper called *A Normal Form For Elliptic Curves*[10] where he presented the idea of elliptic curve with simple formula for addition. The Edwards curve is defined over non-binary field $\mathbb{F}$[5].

Definition 7 *For field $\mathbb{F}$ with characteristics $K \neq 2$, elliptic curve $E(\mathbb{F})$ is a set of solutions $(x, y) \in \mathbb{F}$ of the equation:*

$$x^2 + y^2 = 1 + dx^2y^2 \tag{3.5}$$

where $d \in \mathbb{F} \setminus \{0, 1\}$.

The element $(0, 1)$ is the neutral element here and the group operation is symmetric and pretty simple. For points (x_1, y_1) and (x_2, y_2) we follow the formula:

$$(x_1, y_1) + (x_2, y_2) = (\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2}) \tag{3.6}$$

The inversion of point $X = (x_1, x_2)$ on E is $-X = (-x_1, x_2)$.

Bernstein and Lange[6] expanded the notion of Edwards form to cover larger class of elliptic curves over the original field:

$$x^2 + y^2 = c^2(1 + dx^2y^2) \tag{3.7}$$

where $cd(1 - c^4d) \neq 0$

and the addition law[26]:

$$(x_1, y_1) + (x_2, y_2) = (\frac{x_1y_2 + x_2y_1}{c(1 + dx_1x_2y_1y_2)}, \frac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)}) \tag{3.8}$$

In 2008 Bernstein, Birkner, Joye, Lange and Peters introduced another view on Edwards elliptic curves and defined *twisted Edwards curve.*

Definition 8 *[5] Fix a field $\mathbb{F}$ with $char(\mathbb{F}) \neq 2$. Fix distinct nonzero elements $a, d \in \mathbb{F}$. The twisted Edwards curve with coefficients a and d is a set of solutions: $E_{a,d} : ax^2 + y^2 = 1 + dx^2y^2$ An Edwards curve is a twisted Edwards curve with $a = 1$.*

Bernstein et al. presents also theorem where they describe transformation from twisted Edwards curve to Montgomery form.

Theorem 4 *[5]*
Fix a field $\mathbb{F}$ with $char(\mathbb{F}) \neq 2$.

- *Every twisted Edwards curve over $\mathbb{F}$ is birationally equivalent over $\mathbb{F}$ to a Montgomery curve. Specifically, fix distinct nonzero elements $a, d \in \mathbb{F}$. The twisted Edwards curve $E_{E_{a,d}}$ is birationally equivalent to the Montgomery curve $E_{M_{A,B}}$, where $A = 2(a + d)/(a - d)$ and $B = 4/(a - d)$. The map $(x, y) \to (u, v) = ((1 + y)/(1 - y), (1 + y)/(1 - y)x)$ is a birational equivalence from $E_{E_{a,d}}$ to $E_{M_{A,B}}$, with inverse $(u, v) \to (x, y) = (u/v, (u - 1)/(u + 1))$.*

- *Conversely, every Montgomery curve over k is birationally equivalent over $\mathbb{F}$ to a twisted Edwards curve. Specifically, fix $A \in \mathbb{F} \setminus \{-2, 2\}$ and $B \in \mathbb{F} \setminus \{0\}$. The Montgomery curve $E_{M_{A,B}}$ is birationally equivalent to the twisted Edwards curve $E_{E_{a,d}}$, where $a = (A + 2)/B$ and $d = (A - 2)/B$.*

These transformations are important for our implementation part. We want to make our experiments on the same group so we know the properties of the curve are preserved.

3.2 Elliptic curves over $\mathbb{F}_p$

Elliptic curves was found as great tool in practical cryptography. Our chosen approach is to limit them to finite fields. The operations are defined similarly to elliptic curve operations over general field. One important property of elliptic curve over finite field is the number of points on it. This is a difficult question and extensive research was made about it. One of the most important findings is the Hasse's theorem:

Theorem 5 (H. Hasse, 1993) *Let p be a prime and let E be an elliptic curve over $\mathbb{F}_p$. Then*

$$|p + 1 - \#E(\mathbb{F}_p)| < 2\sqrt{p} \tag{3.9}$$

We take advantage of this theorem in estimation of bounds for algorithms and finding the exact order of the group.

Solving DLP on elliptic curves could be solved in the same manner as solving DLP on generic group. We can use all algorithms we mentioned before except of index calculus.

Chapter 4

Solving Discrete Logarithm Problem using Baby-step Gianstep

Every concept in cryptography is excessively examined so solving of DLP is. There are known algorithms to solve DLP. Baby-step giant-step is universal and was improved many times.

4.1 Interleaving improvement

Pollard[23] published his idea of improvement of baby-step giant-step algorithm in 2000. The idea is based on parallel computation of both lists and storing them in one big hash table. It means we do not compute baby-steps first, then giant-steps and finally seek for collision. We alternately compute one baby-step and one giant-step trying to find collision in progress. It is more memory demanding because it requires up to twice the storage. We need about a square root of group elements so on average we compute only $\frac{4N^{\frac{1}{2}}}{3} = \frac{4}{3}\sqrt{N}$ steps. The explanation of this result by Pollard the following: "If two random numbers x and y are chosen in the interval $[0, 1]$, the expected value of $max(x, y)$ is $\frac{2}{3}$." The worst case scenario is solved in $\sqrt{2N}$ steps.

We can summarize the idea in the following pseudocode:

Algorithm 4 Baby-step Giant-step with interleaving

$M \leftarrow \lceil \sqrt{N} \rceil$
$P' \leftarrow MP$
for $n_0 = 0...M$ **do**
    Save tuple (n_0P, n_0)
    Save tuple $(Q - n_0P', n_0)$
    Check for collision across whole lists
end for

4.2 Two grumpy giants and baby

The most important improvement for us is algorithm called *Two grumpy giants and baby*. It was presented by Bernstein and Lange in 2013.[4] The principle is still the same, we compute baby-steps and giant-steps however we add second set of giant-steps. The baby-steps are of the form n_0P for $M \approx 0.5\sqrt{N}$ values of n_0. One grumpy giant starts at Q and takes M steps of size $P' = MP$. The second grumpy giant starts at arbitrary multiple of Q, for example $2Q$ and takes steps of size $-P'' = -(M+1)P$. Notice these steps are in the opposite direction. It is necessary to choose different size of steps of both giants. The idea behind it is presented in original article of Berstein and Lange. By using two different giants we can cover bigger part of group. So the size of steps does not necessarily need to differ by one. On the other hand we need to choose size of steps properly so they do not overlap (multiples, etc.). The rest of the algorithm is the same as in traditional Baby-step Giant-step above. We are looking for collisions in our lists of found steps. In particular case:

1. $Q = (i - jM)P$
2. $2Q = (i + k(M+1))P$
3. $Q = (jM + k(M+1))P$

Once we find such a collision between any lists we can say the problem is solved. To be explicit when we find collision:

- $Q = (i - jM)P$

$$n = i - jM \tag{4.1}$$

- $2Q = (i + k(M+1))P$

$$n = (i + j(M+1)) \cdot 2^{-1} \tag{4.2}$$

- $Q = (jM + k(M + 1))P$

$$n = jM + k(M + 1) \tag{4.3}$$

For better understanding the pseudocode is following:

Algorithm 5 Two Grumpy Giants and Baby

$M \leftarrow \lceil \sqrt{N} \rceil$
$P' \leftarrow MP$
$P'' \leftarrow (M + 1)P$
for $n_0 = 0...M$ **do**
    Save tuple (n_0P, n_0)
    Save tuple $(Q - n_0P', n_0)$
    Save tuple $(2Q - n_0P'', n_0)$
    Check for collision
end for

4.3 Effective inversion computation improvement

Not every improvement which is applicable on group of elliptic curve points is suited for a generic group. Elliptic curve operations are specific however we can make use of this specificity for our own sake.

One of such an improvements is usage of simple property of EC in simplified Weierstrass form - inversion of the point is made by reflecting the point across x-axis. Mathematically speaking inversion of point $A = (x, y)$ is point $-A = (x, -y)$. As we can see there is no real computation so we get second point instantly. We use this to speed up computation of our baby-steps. Now we can adjust our basic algorithm. Let N be an order of the group, $M = \sqrt{N}$ and $n = \pm n_0 + Mn_1$ where $-\frac{M}{2} \leq n_0 < \frac{M}{2}$ and $0 \leq n_1 < M$. We are looking for collisions in both lists. If x-coordinate of giant-step lies within x-coordinates of baby-step we can say $Q - n_1P' = \pm n_0P$. Using efficient inversion we got average-case running time $\sqrt{N}$ and worst-case running time $\frac{3}{2}\sqrt{N}$. The proof is simple - on average we need to compute half of giant-steps to success and all baby-steps. However we compute two baby-steps at once, so $\frac{M}{2} + \frac{M}{2} = M = \sqrt{N}$. The worst case means computation of all giant-steps, so $\frac{M}{2} + M = \frac{3}{2}M = \frac{3}{2}\sqrt{N}$.

The pseudocode is following:

Algorithm 6 Baby-step Giant-step with Effective Inversion Improvement

$M \leftarrow \lceil \sqrt{N} \rceil$
$P' \leftarrow MP$
for $n_0 = 0...M$ **do**
    Save tuple x-coordinate and index $(n_0 P.x, n_0)$
end for
for $n_1 = 0...M$ **do**
    Look for x-coordinate of $Q - n_1 P'$ in saved tuples
end for

4.4 Combination of effective inversion and interleaving

Effective inversion and interleaving ideas were already presented so nothing holds us back to use both ideas together. We got a proof both algorithms can be merged with better results than we got before. The worst-case is $\sqrt{2N}$ because we need to compute all baby-steps and all giant-steps. However interleaving idea allows us to compute one giant-step and one baby-step at time. And according to Pollard's analysis [23] we find collision in lists in $\frac{4\frac{M}{2}}{3}$. The leading constant corresponds to $2\frac{\sqrt{2}}{3} \approx 0.9428$.

The idea is described by the following pseudocode:

Algorithm 7 Baby-step Giant-step with Effective Inversion and Interleaving

$M \leftarrow \lceil \sqrt{N} \rceil$
$P' \leftarrow MP$
for $n_0 = 0...M$ **do**
    Save tuple x-coordinate and index $(n_0 P.x, n_0)$
    Look for x-coordinate of $Q - n_1 P'$ in saved tuples
end for

4.5 Two grumpy giants and baby using effective inversion

Grumpy giants are using interleaving by design so the only improvement we can make is usage of effective inversion.

The pseudocode:

Algorithm 8 Two Grumpy Giants and Baby Using Effective Inversion

$M \leftarrow \lceil \sqrt{N} \rceil$
$P' \leftarrow MP$
$P'' \leftarrow (M+1)P$
for $n_0 = 0...M$ **do**
    Save tuple x-coordinate and index $(n_0 P.x, n_0)$
    Save tuple x-coordinate and index $((Q - n_0 P').x, n_0)$
    Save tuple x-coordinate and index $((2Q - n_0 P'').x, n_0)$
    Check for collision across all three lists.
end for

4.6 Effective group operation implementation

We presented way how to make computation of inversion more effective however we can improve effectiveness of the operation in common. Lets focus on affine Weierstrass form. Let

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{4.4}$$

be a formula of elliptic curve and let $X = (x_1, y_1), Y = (x_2, y_2)$ be two general points on E so $X \neq \pm Y$. Lets focus on cost of addition $(x_3, y_3) = X + Y$ over $\mathbb{F}_q$. Then

- $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$
- $x_3 = \lambda^2 + a_1\lambda - x_1 - x_2 - a_2$
- $y_3 = -\lambda(x_3 - x_1) - y_1 - a_1x_3 - a_3$

As we can see there are one inversion, two multiplications and one second power. Moved to symbols if we denote consequently multiplication, power and inversion by $\mathcal{M}, \mathcal{S}, \mathcal{I}$, the addition costs $\mathcal{I} + 2\mathcal{M} + \mathcal{S}$. Point doubling can be described the similar way: $\mathcal{I} + 2\mathcal{M} + \mathcal{S}$. According to [11] the realistic estimate of the ratio $\frac{\mathcal{I}}{\mathcal{M}}$ of inversion to multiplication cost is 8 or higher. Lets $(x_4, y_4) = X - Y = X + (-Y) = (x_1, y_1) + (x_2, -y_2 - a_1x_2 - a_3)$ denote as follows:

- $\lambda = \frac{-y_2 - a_1 x_2 - a_3 - y_1}{x_2 - x_1}$
- $x_4 = \lambda^2 + a_1\lambda - x_1 - x_2 - a_2$
- $y_4 = -\lambda(x_3 - x_1) - y_1 - a_1 x_3 - a_3$

Notice there is expression $(x_2 - x_1)^{-1}$ twice - once in $X + Y$ and once in $X - Y$. This means one inversion is used twice so we can compute $(x_2 - x_1)^{-1}$ and compute both points $X+Y$ and $X-Y$ effectively thus speed up Baby-step Giant-step and also Two Grumpy Giants and Baby algorithm. The additional cost of computing second addition is roughly $2\mathcal{M} + \mathcal{S}$.

Chapter 5

Implementation

The goal of this thesis is to compare baby-step giant-step algorithm variants on the elliptic curves. This means we need to prepare elliptic curves, algorithms and measure their performance. The test of impact of cofactor was performed as follows:

- Generate elliptic curve
- Generate point P on elliptic curve as the basis of the logarithm
- Generate 200 instances of discrete logarithm problem
- Solve DLPs using chosen algorithms and measure their running time

The test of impact of elliptic curve type was performed as follows:

- Generate elliptic curve in Montgomery form
- Generate point P on elliptic curve as the basis of the logarithm
- Generate 50 instances of discrete logarithm problem
- Transform elliptic curve and DLPs into Weierstrass form, Edwards form respectively
- Solve DLPs using chosen algorithms and measure their running time

Lets discuss the points individually.

5.1 Elliptic curve generation

Work starts with implementation of generation of cryptographically strong elliptic curves. I based this part on paper presented by Baier and Buchmann in 2002[2]. The generation is not 100 % reliable because there is no known method to generate large prime numbers. We use probabilistic primality tests - the Miller-Rabin test. The number of independent tests is set to 25 according to ANSI[16]. The way we acquire the prime number is not critical because there is not known attack on elliptic curve cryptosystem which exploits specific properties any primes. So I used the proposed way to generate prime numbers. The idea is based on usage of one-way hash function (SHA-256 in our case).

- Generate two random strings A_{hash}, B_{hash}
- Hash strings such that parameters $A = \text{hash}(A_{hash})$, $B = \text{hash}(A_{hash})$
- Check value of discriminant of the curve - if it is 0, choose another parameters A, B
- Generate random point on the curve
- Get order of the curve using baby-step giant-step algorithm
- Test the curve for cryptographical strength

First two points will not be broadly described here because they belong to language specific problematics. We generate two random strings and transform it using hash function. The transformed strings are parameters A and B of elliptic curve. Now we got elliptic curve template. Following step is to check whether this curve satisfies condition of non-zero determinant. Determinant check is self-explanatory so we will not discuss it either.

During the elliptic curve generation I have experienced issue with distribution of k parameter. When we generate the curve we specify multiplier k which stands for a value of group order divided by the order of the biggest subgroup with prime order. The first condition of cryptographically strong elliptic curve tests whether the cofactor k is small enough ($k \leq 4$). The first idea was to generate elliptic curves at random and sort them according to cofactor k because k could have tremendous effect on the performance of algorithms and generating elliptic curves according to k value could take too much time. The issue occurred when first 100 elliptic curves were generated. The distribution of k heavily favored values 1 and 2 (56, respectively 33 occurrences) at the expense of values 3 and 4 (6, respectively 5 occurrences). So the we are forced to generate elliptic curve with fixed k which at the end worked well.

The important point comes now. In the next step we are going to generate point on this elliptic curve.

5.2 Point generation

The generated point is necessary for getting the order of the curve. Lets follow procedure proposed by Baier and Buchmann [25]. We get the point by generating random value of x-coordinate and compute the y-coordinate afterwards.

When we fit value of our random x-coordinate into the elliptic curve equation we got quadratic equation modulo p. In this case we use so called *Legendre's symbol* to find out whether the $x^3 + Ax + B$ is is a square in $\mathbb{F}_p$ or not.

Definition 9 (Legendre's symbol) *[14]*

We define Legendre's symbol $(\frac{a}{p})$ *where* p *is an odd prime and* a *is any number not divisible by* p*, by*

$$\left(\frac{a}{p}\right) = +1 \text{ if } a \text{ is a quadratic residue } (mod\ p) \tag{5.1}$$

$$\left(\frac{a}{p}\right) = -1 \text{ if } a \text{ is a quadratic non-residue } (mod\ p) \tag{5.2}$$

It is that

$$\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right) \tag{5.3}$$

if $a \equiv b\ (mod\ p)$

Now we know we have generated right x-coordinate for parameters A and B so the next step is to compute y-coordinate. There is an universal effective algorithm proposed by Tonelli and Shanks for solving such an equation for every prime modulus.[9] It is probabilistic so we cannot be sure it is going to return result every time however in this case we need to generate random point so we do not stick with one try and when Tonelli-Shanks algorithm fails we just choose different x-coordinate and start again. Before we continue with the algorithm, lets define the p-subgroup and introduce *Sylow's theorem*.

Definition 10 *If* p *is a prime, then a finite group* G *is called a p-group if* $|G| = p^n$ *for some* $n \geq 0$*. A p-subgroup is a subgroup which is a p-group.*

Theorem 6 (Sylow's theorem) *Let p^e be the largest p-power dividing the order of G.*

1. *The Sylow p-subgroups of G are exactly the subgroups of order p^e.*
2. *The Sylow p-subgroups of G are conjugate in G. In particular*

$$|Syl_pG| = |G : N_G(P)| \text{ for } P \in Syl_pG \tag{5.4}$$

3. $|Syl_pG| \equiv 1(mod\ p)$

We are going to pinpoint the basic idea:

- The number $p-1$ can be always disassembled as $2^e \cdot q$ and we are sure q is odd.
- The multiplicative group $(\mathbb{Z}/p\mathbb{Z})*$ is isomorphic to the additive group $\mathbb{Z}/(p-1)\mathbb{Z}$, hence its 2-Sylow subgroup G is cyclic group of order 2^e
- If a is a quadratic residue mod p

$$a^{(p-1)/2} = (a^q)^{2^{e-1}} \pmod{p}, \tag{5.5}$$

- Lets denote $b = a^q \bmod p$. We can say it is a square in G so there is an even integer $0 \leq k \leq 2^e$ such that

$$a^q z^k = 1 \text{ in } G \tag{5.6}$$

- Now set

$$x = a^{(q+1)/2} z^{k/2} \tag{5.7}$$

To solve this we need to find generator z of G - this is the probabilistic part. The easiest way to find z is to choose a random integer n and compute $z = n^p$. The generator is found when n is quadratic non-residue mod p. [9]

Cohen mentioned that y is a generator of G_r and b is in G_{r-1} when G_r is a subgroup of G whose elements have and order dividing 2^r. This means b is a square in G_r.

At the end of this algorithm we can surely say we have found a legit point at the curve.

The point is in code represented by type called *ECWpoint*:

```
type ECWpoint
    x::BigInt
    y::BigInt
end
```

Point at infinity is called *ECWinfinity*:

```
type ECWinfinity
    inf::Bool

    function ECWinfinity()
        self = new()
        self.inf = true
        return self
    end
end
```

Here is the complete algorithm:

Algorithm 9 Point Generation Algorithm

while $\left(\frac{n}{p}\right) \neq -1$ **do**
  Choose n at random
end while
$z \leftarrow n^q \pmod{p}$
$y \leftarrow z \pmod{p}$
$r \leftarrow e \pmod{p}$
$x \leftarrow a^{(q-1)/2} \bmod p$
$b \leftarrow ax^2 \pmod{p}$
$x \leftarrow ax \bmod p$
while $b \not\equiv 1 \pmod{p}$ **do**
  Find smallest $m \geq 1$ mod such that $b^{2^m} \equiv 1 \pmod{p}$
  if m = r **then**
    a is not a quadratic residue mod p
  end if
  $t \leftarrow y^{2^{r-m-1}} \pmod{p}$
  $y \leftarrow t^2 \pmod{p}$
  $r \leftarrow m \pmod{p}$
  $x \leftarrow xt \pmod{p}$
  $b \leftarrow by \pmod{p}$
end while
return x

5.3 Order of the curve

The order is found by algorithm proposed in article *Counting point on elliptic curves over finite fields* by Schoof [25]. The idea is based on the fundamental property of generator and order of the curve. When we multiply generator G by the order of the group N we got point at infinity - $NG = \mathcal{O}$.

At first, let the random point $P \in E(\mathbb{F}_p)$ be picked. To find order of the curve, we are searching for integer m which satisfies equation $mP = \mathcal{O}$, so we are solving DLP where Q is the point at infinity. When we find more integers m with similar property it means the group of points of curve $E(\mathbb{F}_p)$ consists of smaller groups. According to Lagrange's theorem order of every subgroup H divides the order of group G. This means the random point we chose is generator of such a subgroup however we can use the Lagrange's theorem and try multiples of subgroup's order. However we know some algorithms to find the order of the group.

The second part of algorithm is built on Hasse's theorem. This theorem gives us bounds for our estimated order of the curve. These are $(p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p})$. Now we need to find the order itself. This problem is transformed to DLP and here comes the basic baby-step giant-step algorithm on interval.

- Pick random point $P \in E(\mathbb{F}_p)$
- Compute $s = \sqrt[4]{p}$ baby-steps
- Compute $s = \sqrt[4]{p}$ giant-steps
- Find collision and return m

The point-picking part was already discussed so lets now focus on the difference in this baby-step giant-step algorithm and the generic one. In this particular scenario we are using BSGS algorithm on interval. This excludes all variants of Grumpy giants because second list of giants, the one starting from $2Q$ can exceed the the interval and do not guarantee the collision therefore the solution of DLP.

Baby-steps are computed exactly the same way as before however the range is different. We pick the point and compute s additions. The number s here is chosen according to the bounds of the interval:

$$|(p + 1 - 2\sqrt{p}) - (p + 1 + 2\sqrt{p})| = \sqrt[4]{p} \tag{5.8}$$

By these baby-steps we cover the whole interval so we know the collision is going to occur. We also incorporate the negation trick to cover broader interval and increase the probability of early collision. In fact, we compute $2s + 1$ points - s additions, inversions and point at infinity. Someone can ask why do we compute twice as much points? We need to remember the DLP

is solved over the interval and inversion of computed points do not need to belong to this interval so if we cut our computations in half we might not cover the whole interval.

Next we compute the size of our giant-step $Q = (2s + 1)P$ and starting point of giant-steps $R = (p+1)P$. Efficient multiplications are made by binary extension. Now we compute approximately s giant-steps. The DLP is solved when the collision is found and we can say the order m of the group is

$$m = p + 1 + (2s + 1)i - j \tag{5.9}$$

where i is position of giant-step and j is position of baby-step. This algorithm could fail if there are two numbers m in the interval with $mP = m'P = \mathcal{O}$. It is rare and tells us we have found subcurve of curve and the order d we found divides the order of the whole curve so we can compute the final order using this knowledge.

The whole algorithm is the following:

Algorithm 10 Baby-step Giant-step for Order of the Group

Pick a point P
for $j = 1 : \sqrt[4]{p}$ **do**
    Compute baby-steps jP
end for
$Q \leftarrow (2s + 1)$
$R \leftarrow (p + 1)P$
for $i = 1 : \sqrt[4]{p}$ **do**
    Compute giant-steps $R + iQ$
end for
Find collision between baby-steps and giant-steps
$m = p + 1 + (2s + 1)i - j$
Return m

5.4 Cryptographically strong curve test

The last part of elliptic curve generation is to make sure the curve is cryptographically strong. According to [2] here are three conditions elliptic curve has to meet to be cryptographically strong.

- $|E(\mathbb{F}_p)| = k \cdot r$ with a prime $r > 2^{160}$ and a positive integer $k \leq 4$.
- $r \neq p$
- The order of p in the multiplicative group $\mathbb{F}_r^\times$ of $\mathbb{F}_r$ is at least s, where $s \geq 20$.

The first condition secures elliptic curve against commonly known algorithms to solve DLP. The r_0 is the prime order of the biggest subgroup of the elliptic curve and k is cofactor. The second protects against attack on anomalous elliptic curves and the last one protects against attacks which reduce the DLP in $E(\mathbb{F}_p)$ to DLP in finite field extension of $\mathbb{F}_p$.

When all these conditions are met the secure elliptic curve is generated. In my implementation of elliptic curve we are provided with object attributes where all useful information are saved - parameters A, B, prime p, cofactor k and at the end added order of the curve N. All these values are saved as BigInt types to provide type big enough for cryptographical use except of k which is only in range $1, \ldots, 4$. The final type EC:

```
type ECW
    A::BigInt
    B::BigInt
    p::BigInt
    N::BigInt
    k::Int
    inf::ECWinfinity

    function ECW(A,B,p)
        self = new()
        self.A = A
        self.B = B
        self.p = p
        self.k = -1
        self.inf = ECinfinity()
        return self
    end

    function ECW(A,B,p,N,k)
        self = new()
        self.A = A
        self.B = B
        self.p = p
        self.N = N
        self.k = k
        self.inf = ECinfinity()
        return self
    end
end
```

The EC type got two constructors because we need to have elliptic curve object created when we want to compute order of the curve. So at the end we can return new EC object with order of the curve and cofactor k.

5.5 Generation of DLP instances

Once we have our elliptic curve prepared we can start generating testing data sets. During generation of elliptic curve we used function to generate point on curve so we are going to reuse this function again in every DLP instance. One DLP instance consists of:

- Get point P
- Select random $n, 0 < n \leq N$
- $Q = nP$ using double and add algorithm

More precisely in code:

```
P01 = ECinfinity()
while P01 == ECinfinity()
    P01 = getPoint(curve01)
end

n = rand(1:curve01.N)
Q01 = binaryExpansion(P01, n, curve01)
```

5.6 Transformation of elliptic curves

The transformations are made using the formulas in chapter *Elliptic curves*. This brings us additional classes for Edwards-form curves and for Montgomery-form curves. As we know, the elliptic curve in Montgomery form has very similar attributes as Weierstrass-form elliptic curve, however Edwards elliptic curves are implemented differently. The point at infinity $\mathcal{O} = (0, 1)$, so there is no need for additional type to describe it. This simplification allows us to omit operators such as equivalence or comparison.

Finally, we implemented functions that allows us to transform point, respectively curve, from Montgomery form into Weierstrass form and Edwards form. We generate Montgomery-form elliptic curve as our base curve due to simplicity of formulas and no implementation obstacles which I experienced when started from Weierstrass-form elliptic curve.

The function for transformation Montgomery-form curve into Weierstrass-form curve is specific due to its double purpose. We use this function during generation of elliptic curve in Montgomery form itself. The function to determine order of the group was implemented for Weierstrass-form curve so we transform only parameters of the curve and determine order of the group. The second purpose is simply to transform elliptic curve during generation of DLPs, where the group order is necessary for tests.

The double purpose is demonstrated in the try-catch block at the end of the function:

```
function ECMtoW(curve::ECM)
    A = curve.A
    B = curve.B
    p = curve.p

    s = B
    alpha = (A * gcdx(p, 3*B)[3])%p
    a = (1 * gcdx(p, s^2)[3] - 3*alpha^2)%p
    b = (-alpha^3 - a*alpha)%p

    if a < 0
        a += p
    end
    if b < 0
        b += p
    end

    try
        return ECW(a,b,p,curve.N,curve.k)
    catch
        return ECW(a,b,p)
    end
end
```

5.7 Code structure

The code is split into several modules - standalone Julia files meant to implement some functionality. The main program is called *dlp_solver.jl* and every other module is linked to it. The basic modules are *ec_generator.jl* and *ec_operations.jl*. First one is responsible for correct generation of elliptic curve and the second implements basic structures (or types) like an elliptic curve or point on elliptic curve. It also implements operations on chosen elliptic curve - addition, subtraction, multiplication. Algorithms for operations on elliptic curves were also implemented. For example Tonelli-Shanks algorithm to solve congruence modular arithmetics, binary extension for effective multiplication, point generation and algorithm to get order of elliptic curve.

Module *ec_generator.jl* is used first and can be used alone simply to generate random elliptic curve. The third module - *bs_gs.jl* represents Baby-step Giant-step algorithm and improvements. It requires *ec_operations.jl* to work or any other module providing operations and functions necessary.

5.8 Methodology

The tests were made in SageLab environment using the following resources:

- Processor Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- RAM 32 GB
- OS Gentoo Linux
- Kernel version 4.9.95-gentoo
- Julia version 0.6.2

All of algorithms were run sequential so no multicore advantage was used. Every algorithm were measured by in-built macro *@time* to eliminate initialization of program and Julia's compiler.

I chose Julia language for implementation for its proposed speed and Python-like feel. Not many researches take advantage of this mostly unknown language so I decided to examine Julia's performance and usability. Julia performs the prerun analysis of the code and precompiles functions using C and Fortran libraries. This is done to maximize the speed of the resulting program. However to eliminate this compilation time we need to run every function once, for example on simple instance of DLP on small elliptic curve.

The test are prepared as follows:

- 364 cryptographically strong elliptic curves in Weierstrass form ($p \approx 2^{34}$)

  91 curves where $k = 1$

91 curves where $k = 2$

91 curves where $k = 3$

91 curves where $k = 4$

- 50 DLP instances
- New point generated for every instance
- New multiplier n generated for every instance

The performance test of influence of form of the elliptic curve is different due to elimination of the cofactor k.

- 364 cryptographically strong elliptic curves in Montgomery form ($p \approx 2^{36}$)
- 50 DLP instances
- New point generated for every instance
- New multiplier n generated for every instance

Every performance of every algorithm is saved into file and the arithmetic average is computed afterwards. The results are divided according to algorithm and k value of elliptic curve so we can observe the impact of different cofactor ks.

The basic sample of Julia code is provided with final summary table and graph for better clarity.

5.9 Algorithm implementation and performance

This section demonstrates every algorithm used for tests and the results. The results of impact of elliptic curve form on performance are commented at the end of the chapter due to constant performance over the whole test.

5.9.1 BSGS

The first algorithm is classical baby-step giant-step without any improvement. It is strict computation of two lists of size $\sqrt{N}$.

```
function BSGS(P::ECpoint, Q::ECpoint, curve::EC)
    M::BigInt = ceil(sqrt(N))
    Pdot = binaryExpansion(P, M, curve)
    baby = ECinfinity()

    for n0 in 0:(M - 1)
        if n0 == 0
            babystep[baby] = 0
        else
            baby = +(baby, P, curve)
            babystep[(baby.x, baby.y)] = n0
        end
    end

    giant = Q

    for n1 in 0:(M - 1)
        if n1 == 0
            giantstep[giant] = 0
        else
            giant = -(giant, Pdot, curve)
            giantstep[(giant.x, giant.y)] = n1
        end
    end

    n::BigInt = -1

    for key in keys(giantstep)
        if key in keys(babystep)
            n1 = giantstep[key]
            n0 = babystep[key]
            n = n0 + M*n1
            break
        end
    end
return n%N
```

Julia treats operators as functions and we are allowed to overload them. Notice the notation

```
1 +(baby, P, curve)
```

which is chosen for group operation (point addition). The curve where we compute the operation needs to be specified so we cannot use standard binary notation.

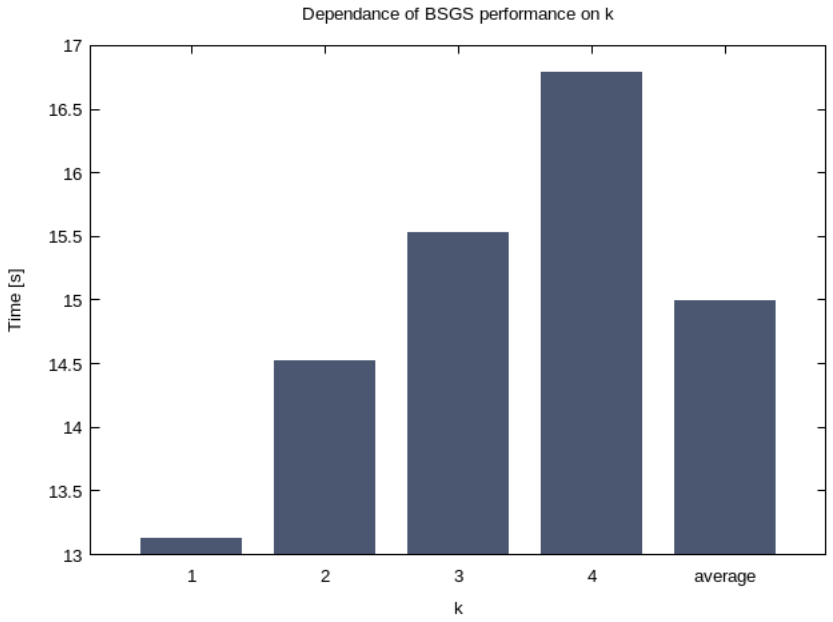


Figure 51: Dependence of the performance on cofactor k and average

The figure above demonstrates the dependance of performance on cofactor k. The difference in the performance is clearly visible however the it is quite surprising, the performance for $k = 1$ especially. We expect $k = 1$ the most difficult to solve because there is just one solution for one congruence.

Table 51: Exact values, extremes and averages

k	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
1	13.125469	5.721561	26.619746	43.591289	202.809870
2	14.520804	7.319894	28.988362	50.409700	199.633308
3	15.533491	6.687442	29.447798	43.051765	189.576178
4	16.792924	7.039855	28.741645	41.921555	171.153305
average	14.993172	6.692188	28.449388	44.634904	189.748958

Performance of the algorithm is consistent all-around however it is slightly decreasing with increasing k. The range of performances is really narrow compared to the following algorithms.

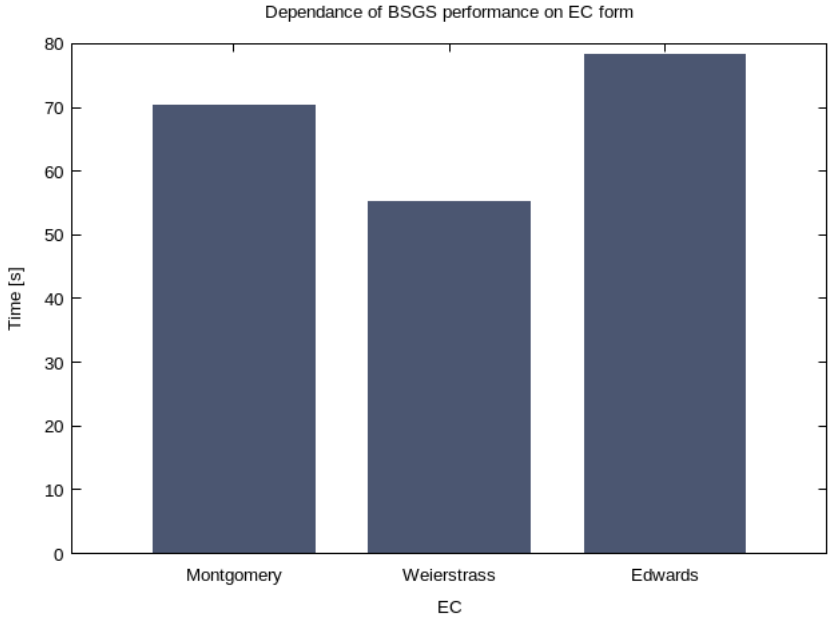


Figure 52: Dependence of the performance on elliptic curve form

Table 52: Exact values, extremes and averages

EC form	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
Montgomery	70.373566	35.791502	109.145135	50.859298	155.093937
Weierstrass	55.172434	31.759635	90.989876	57.564316	164.919088
Edwards	78.259778	42.737542	119.869594	54.609842	153.168837

5.9.2 BSGS with interleaving

Now we begin with improvements of algorithms. The first one is called *interleaving*. We use this trade-off between computation of lists and searching among them for collisions. Both lists here are computed in parallel - one step for baby-steps and one for giant-steps.

```
function BSGS_interleaving(P::ECpoint, Q::ECpoint, curve::EC)
    M::BigInt = ceil(sqrt(N))
    Pdot = binaryExpansion(P, M, curve)
    baby = ECinfinity()

    giant = Q

    for n0 in 0:(M - 1)
        n1 = n0
        if n0 == 0
            .
            .
            .
        else
            baby = +(baby, P, curve)
            babystep[(baby.x, baby.y)] = n0
            if (baby.x, baby.y) in keys(giantstep)
                collision = (baby.x, baby.y)
                break
            end

            giant = -(giant, Pdot, curve)
            giantstep[(giant.x, giant.y)] = n1

            if (giant.x, giant.y) in keys(babystep)
                collision = (giant.x, giant.y)
                break
            end
        end
    end
    n1 = giantstep[collision]
    n0 = babystep[collision]
    n = n0 + M*n1
    return n%N
end
```

The points are used as keys in dictionaries however the comparison of keys - points is not done using operator = so keys do no match even if operator = is defined. This led me to use standard tuple (x-coordinate, y-coordinate) to solve this problem.

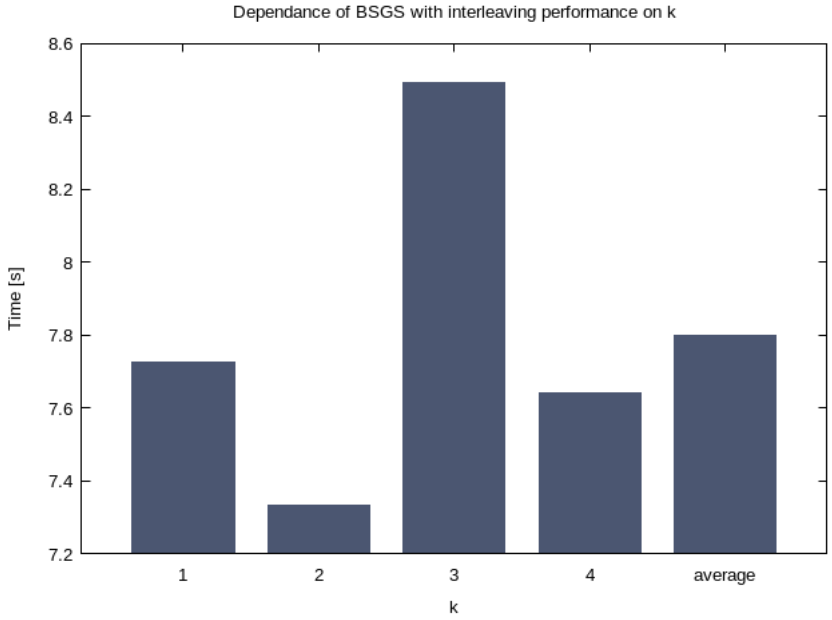


Figure 53: Dependence of the performance on cofactor k and average

As we can see in the graph, there is an interesting tendency in performance. The performance for $k = 2$ is on average significantly lower than for any other value. This could be really important property and it can lead us to change preferences for cryptographically strong elliptic curves. The performance for $k = 3$ is noticeably lower than other values of k.

Table 53: Exact values, extremes and averages

k	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
1	7.726132	0.700646	23.129139	9.068521	299.362433
2	7.332980	0.861346	25.549836	11.746194	348.423652
3	8.493651	1.164999	23.539666	13.716116	277.144258
4	7.639992	0.362043	25.262422	4.738788	330.660340
average	7.798189	0.772259	24.370266	9.903050	312.511873

We can see an increase in the range of achieved performances. This probably is caused by the nature of interleaving - the end of the algorithm immediately when the collision is found.

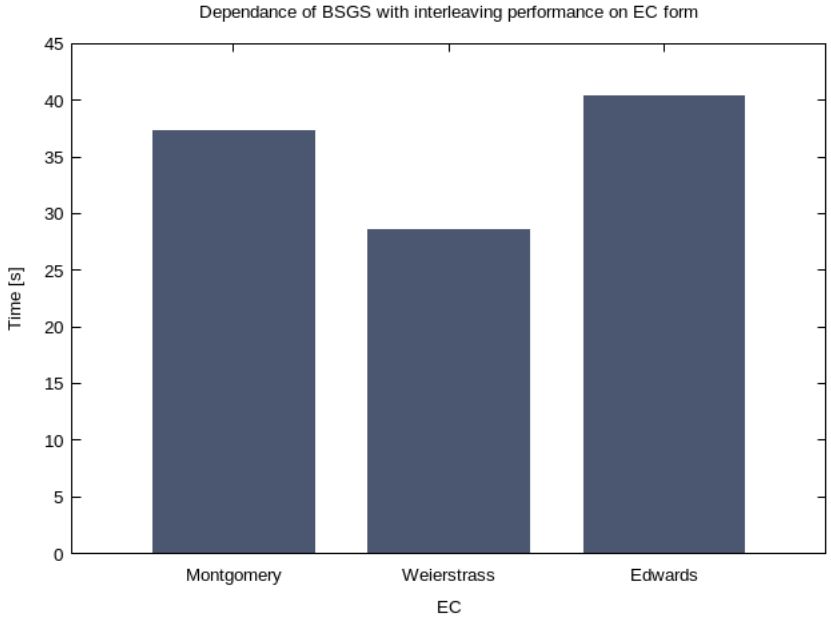


Figure 54: Dependence of the performance on elliptic curve form

Table 54: Exact values, extremes and averages

EC form	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
Montgomery	36.205802	13.472857	90.832155	37.211872	253.110180
Weierstrass	27.614971	10.347181	70.758457	37.469461	294.312503
Edwards	39.753453	15.559296	97.571898	39.139482	255.806615

5.9.3 BSGS for average-case scenario

Case invented to improve performance in average case. We compute only half baby-steps and all giant-steps however the collision is on average found after half giant-steps [13].

```
function BSGS_avg(P::ECpoint, Q::ECpoint, curve::EC)

    M::BigInt = ceil(sqrt(N/2))
    Pdot = binaryExpansion(P, M, curve)

    for n0 in 0:(M - 1)
        if n0 == 0
            babystep[baby] = 0
        else
            baby = +(baby, P, curve)
            babystep[(baby.x, baby.y)] = n0
        end
    end

    giant = Q

    for n1 in 0:(2*M-1)
        if giant != zero
            if (giant.x, giant.y) in keys(babystep)
                i = babystep[(giant.x, giant.y)]
                j = n1
                break
            end
        else
            if giant in keys(babystep)
                i = babystep[giant]
                j = n1
                break
            end
        end
        giant = -(giant, Pdot, curve)
    end

    n = i + M*j
    return n%N
end
```

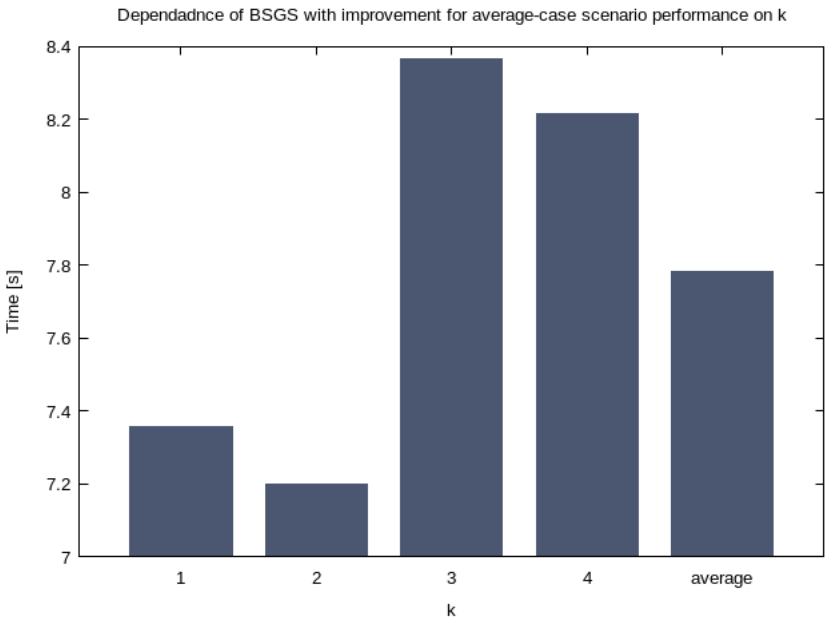


Figure 55: Dependence of the performance on cofactor k and average

Here we see the problem with $k = 2$ again. The decrease in average time needed for solving DLP is significant. Moreover the performance for $k = 3$ is low again and the performance for $k = 4$ dropped also.

Table 55: Exact values, extremes and averages

k	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
1	7.357288	2.296124	20.395209	31.208837	277.210967
2	7.197032	2.489998	17.978332	34.597567	249.802025
3	8.366207	2.357654	22.238265	28.180681	265.810612
4	8.214226	3.371207	20.945501	41.041077	254.990546
average	7.783688	2.628746	20.389327	33.772495	261.949425

The performance is more stable than in the baby-step giant-step with interleaving. We can see that the range of values is still not as stable as it is in the of standard baby-step giant-step however the speedup is so big that we do not consider it as an issue.

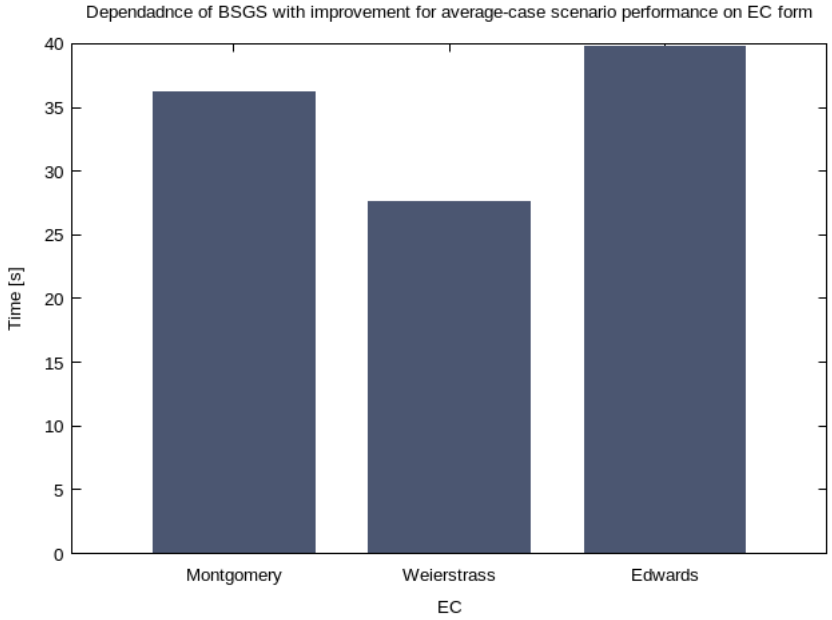


Figure 56: Dependence of the performance on elliptic curve form

Table 56: Exact values, extremes and averages

EC form	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
Montgomery	37.283220	0.430676	94.367626	1.155146	250.877342
Weierstrass	28.582153	0.719022	84.120852	2.515632	256.232231
Edwards	40.390480	0.421135	103.32152	1.042659	245.442570

5.9.4 BSGS with negation

The effective negation takes advantage of simple way to obtain inversion of point in elliptic curves - only swap sign of y-coordinate. With this improvement we do not need to store the whole points and we are oriented only on x-coordinate which provides us with two values (plus and minus). Then we choose the right solution by trial and error.

```
function BSGS_negation(P::ECpoint, Q::ECpoint, curve::EC)

    M::BigInt = ceil(sqrt(N))
    Pdot = binaryExpansion(P, M, curve)
    baby = ECinfinity()

    halfM::BigInt = ceil(M/2)
    for n0 in 0:halfM
        if n0 == 0
            babystep[baby_twins[1]] = 0
        else
            baby_twins = pm(baby_twins[1], P, curve)
            babystep[baby_twins[1].x] = n0
            babystep[baby_twins[2].x] = -n0
            .
            .
            .

    giant = Q

    for n1 in 0:M
        if giant != zero
            if giant.x in keys(babystep)
                i = babystep[giant.x]
                j = n1
                break
        .
        .
        .
        giant = -(giant, Pdot, curve)
    end

    n1 = i + M*j
    return n1
end
```

The implementation of effective inversion led to new operator *pm*. It returns addition and subtraction of two points in a form of list.

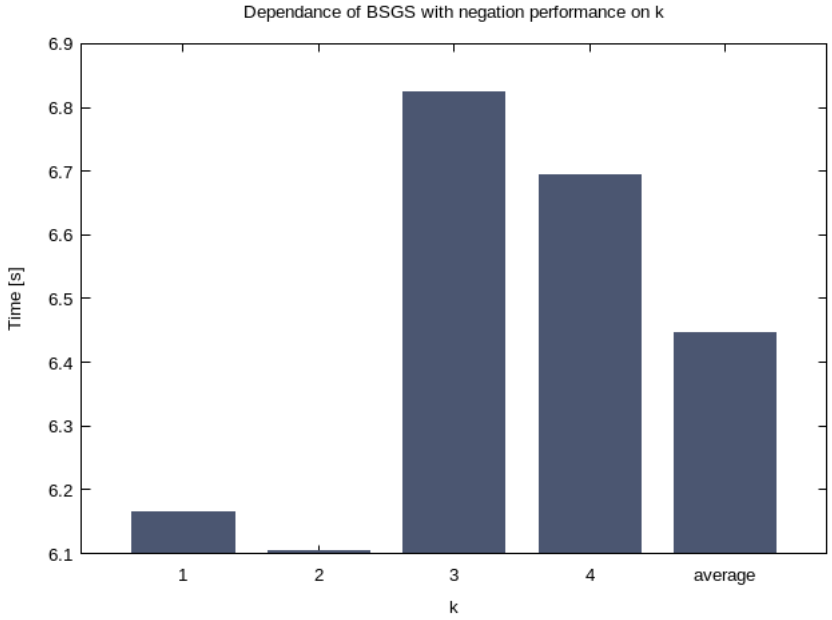


Figure 57: Dependence of the performance on cofactor k and average

This improvement leads to great gain in terms of performance. The possibility to compute two points efficiently allows us to cut our lists in half. Here we see another demonstration of weakness of elliptic curve with $k = 2$. The difference is remarkable especially when compared to $k = 3$.

Table 57: Exact values, extremes and averages

k	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
1	6.165627	2.189847	15.305811	35.517023	248.244212
2	6.104649	1.975018	13.52661	32.352689	221.578843
3	6.824070	2.301508	17.620071	33.726326	258.204733
4	6.693590	2.670852	16.943552	39.901640	253.131027
average	6.446984	2.284306	15.849011	35.432171	245.836067

The trend here is the same as before. We can see no special property here.

5.9.5 Two Grumpy Giants and Baby

First different approach to improve running time of BSGS. We compute two giant-step lists at once. The starting points and sizes of steps are different so we should cover the space more uniformly.

```
1
2  function Grumpies(P::ECpoint, Q::ECpoint, curve::EC)
3
4      M::BigInt = ceil(sqrt(N/2))
5      Pdot = binaryExpansion(P, M, curve)
6      Pdotdot = binaryExpansion(P, M + 1, curve)
7      zero = ECinfinity()
8
9      baby = ECinfinity()
10     grumpy1 = Q
11     grumpy2 = +(Q, Q, curve)
12
13     for n0 in 0:(M - 1)
14         if n0 == 0
15             .
16             .
17             .
18         else
19             baby = +(baby, P, curve)
20             grumpy1 = -(grumpy1, Pdot, curve)
21             grumpy2 = -(grumpy2, Pdotdot, curve)
22
23             try
24                 baby_tuple = (baby.x, baby.y)
25             catch LoadError
26                 baby_tuple = baby
27             end
28             .
29             .
30             .
31             if baby_tuple in keys(grumpies1)
32             || baby_tuple in keys(grumpies2)
33                 collision = baby_tuple
34                 break
35             .
36             .
37             .
38 end
```

Grumpy giants and baby discovered possibility to found collision at the point at infinity. Program tried to access the x-coordinate of point at infinity what results in error. Here comes the idea of try-catch block solving this problem.

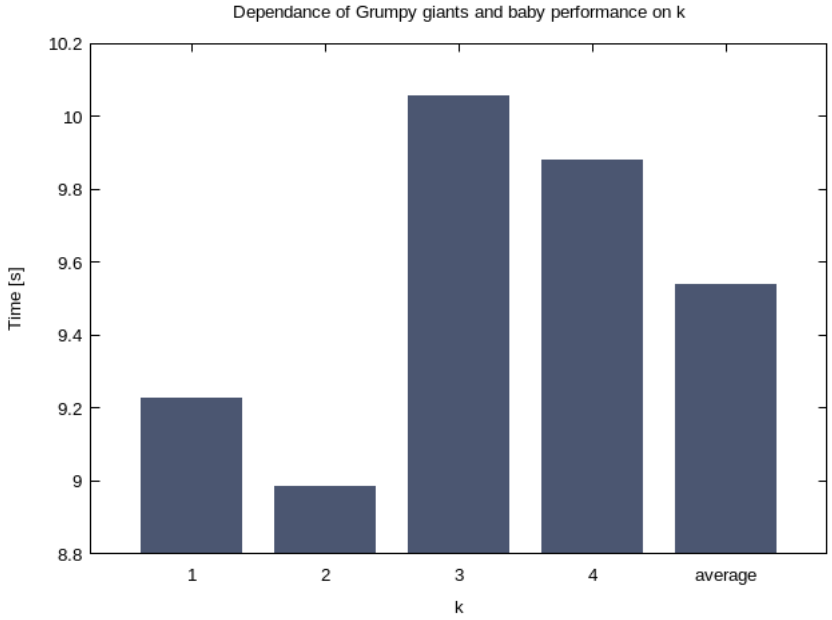


Figure 58: Dependence of the performance on cofactor k and average

Here we have the same suspicious phenomenon - the tremendous advantage for $k = 2$. The performance for $k = 1$ is also interesting because it refutes our expectations for $k = 1$ to be the most difficult to solve.

Table 58: Exact values, extremes and averages

k	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
1	9.226454	0.493646	24.352588	5.350333	263.943097
2	8.983580	0.383498	26.813767	4.268877	298.475308
3	10.057743	0.603972	28.434657	6.005045	282.714090
4	9.878502	0.172987	30.16397	1.751146	305.349622
average	9.536570	0.413526	27.441245	4.336210	287.747546

Notice the difference in min(t) and max(t) for $k = 4$. One instance of ECDLP is solved in time $t \approx 0.173s$ however the different instance is solved in $t \approx 30.164s$, almost 175 times slower.

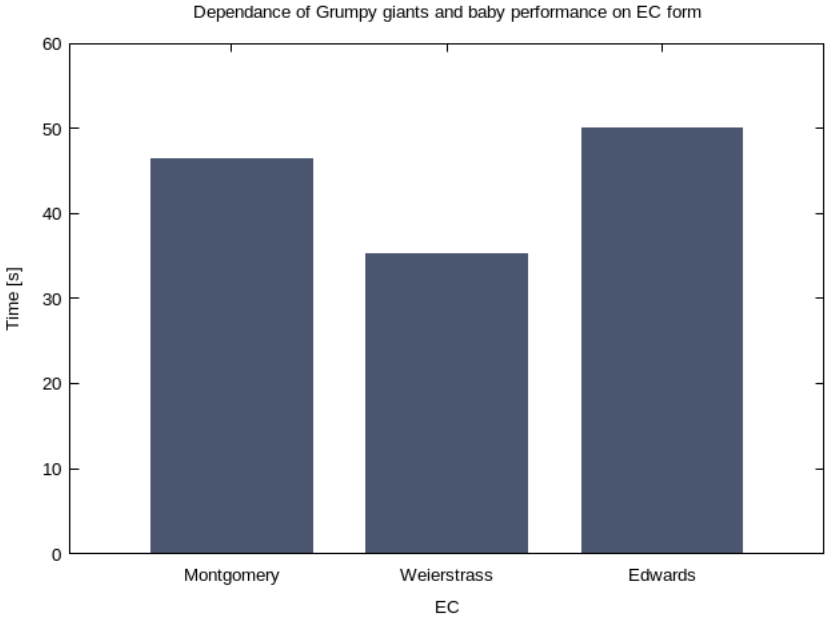


Figure 59: Dependence of the performance on elliptic curve form

Table 59: Exact values, extremes and averages

EC form	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
Montgomery	46.362305	1.440364	115.676475	3.106756	249.505441
Weierstrass	35.321486	2.256696	96.437576	6.389017	273.028080
Edwards	50.107217	1.28697	129.68606	2.568432	258.817123

5.9.6 Two Grumpy Giants with negation

The same concept of effective negation is used for Two grumpy giants and baby algorithm. This speeds up the algorithm significantly.

```
1
2  function Grumpies_negation(P::ECpoint, Q::ECpoint, curve::EC)
3      M::BigInt = ceil(sqrt(N/2))
4      Pdot = binaryExpansion(P, M, curve)
5      Pdotdot = binaryExpansion(P, M + 1, curve)
6      zero = ECinfinity()
7
8      collision = "None"
9      baby = ECinfinity()
10     grumpy1 = Q
11     grumpy2 = +(Q, Q, curve)
12
13     for n0 in 0:M
14         if baby_twins[1] != zero
15             try
16                 babies[baby_twins[1].x] = n0
17                 baby_coord = baby_twins[1].x
18             catch LoadError
19                 babystep[baby_twins[1]] = n0
20                 baby_coord = baby_twins[1]
21             end
22         else
23             babies[baby_twins[1]] = 0
24             baby_coord = baby_twins[1]
25         end
26
27         if baby_twins[2] != zero
28             try
29                 babies[baby_twins[2].x] = -n0
30                 baby_coord = baby_twins[2].x
31             catch LoadError
32                 babies[baby_twins[2]] = -n0
33                 baby_coord = baby_twins[2]
34             end
35         else
36             babies[baby_twins[2]] = 0
37             baby_coord = baby_twins[2]
38         end
39         .
40         .
41         .
42 end
```

Two grumpy giants and baby with effective negation comes with two baby-steps at the same time. This means two times more try-catch blocks.

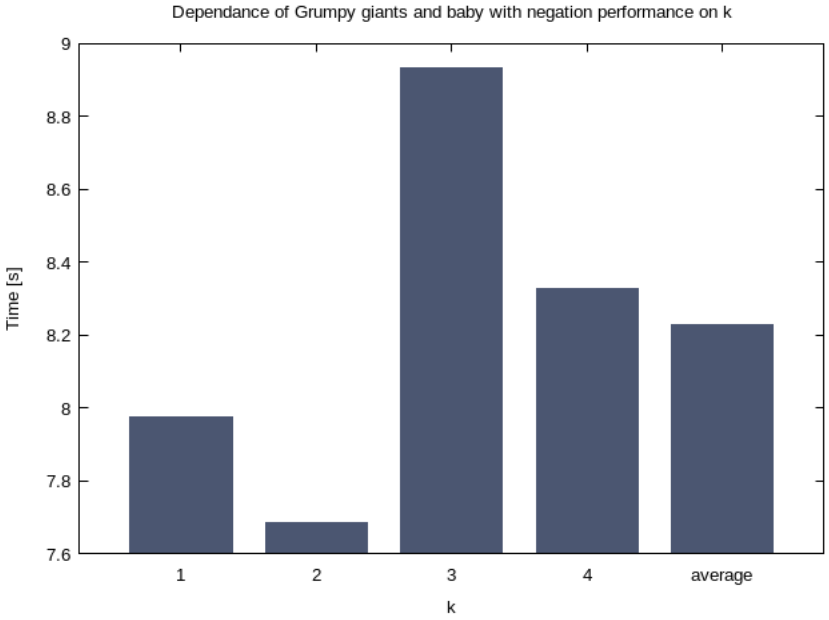


Figure 510: Dependence of the performance on cofactor k and average

We can see the effect of effective inversion. The increase in performance for $k = 4$ is clearly visible and the overall performance of the algorithm is better.

Table 510: Exact values, extremes and averages

k	average(t) [s]	min(t) [s]	max(t) [s]	min(t) [% of avg]	max(t) [% of avg]
1	7.973770	0.393746	23.258653	4.938015	291.689526
2	7.684563	0.287765	27.1102	3.744715	352.787790
3	8.932216	0.648205	29.937078	7.256934	335.158477
4	8.326917	0.129884	24.342643	1.559809	292.336786
average	8.229367	0.3649	26.162144	4.434120	317.911994

The overall performance is better than before however we can see the same range of performances as before.

5.9.7 Comparison of algorithms according to cofactor k and EC form

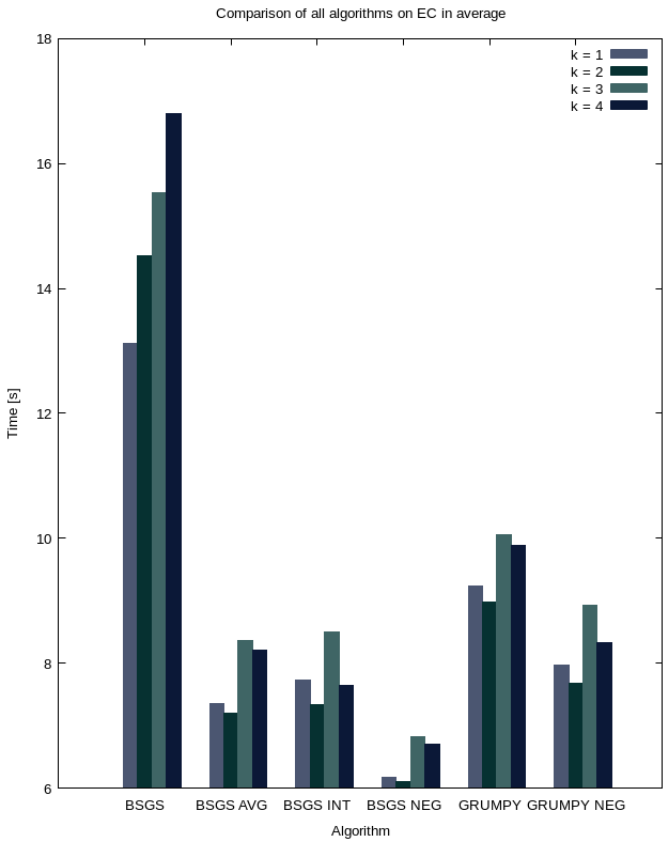


Figure 511: Performance of algorithms depending on k

Now we can clearly see the performance of every algorithm for every case of elliptic curve. The basic baby-step giant-step algorithm is the slowest as expected. The rest of algorithms performs much better however we can say Two grumpy giants and baby algorithm is the second slowest here. This could be due to excessive search across three lists.

It is followed by the version improved by effective inversion. This significantly sped up the algorithm however the logic behind search for the collisions still seems to consume too much resources.

Two baby-step giant-step improvements, interleaving and improvement for average-case scenario deliver very similar results. They differ case by case so we can say interleaving is more suitable for elliptic curve with $k = 4$ and improvement for average-case scenario dominates the rest.

The negation improvement on baby-step giant-step delivers the best results overall. The improvement is such simple however effective. It was the most

efficient algorithm in tests.

Here we can see the trend of $k = 2$ being especially suitable for solving ECDLP. Every other algorithm than standard baby-step giant-step performs the best on the elliptic curves with $k = 2$ and everyone of them struggled with $k = 3$.

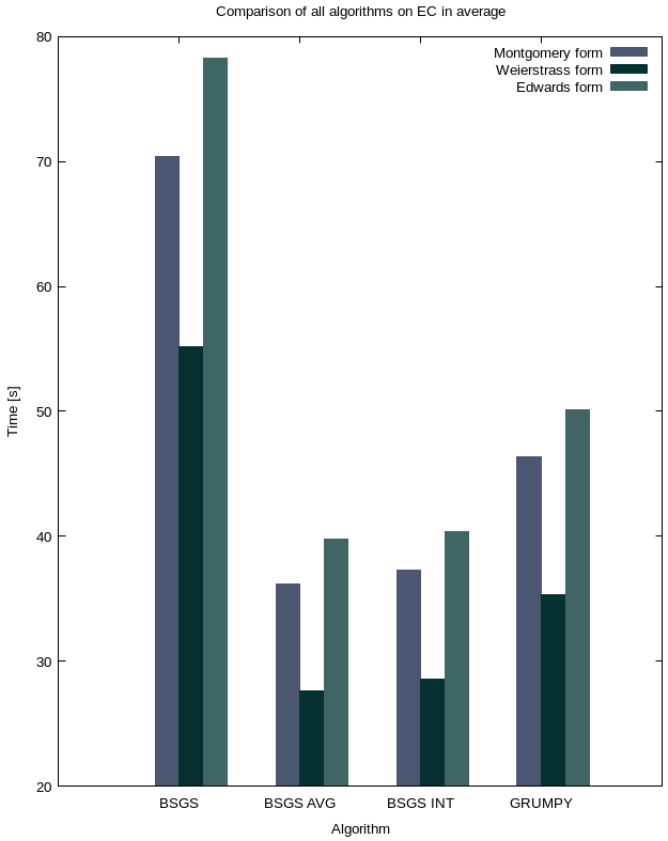


Figure 512: Performance of algorithms depending on EC form

As we can see, the performance of every single algorithm is the best using Weierstrass form of elliptic curve. This may be caused by the implementation of the basic forms of the curves - we did not implement any improvement. The effective negation was omitted so the results are not distorted. In some cases we got better results for alternative forms, especially for grumpy giants, however the results varied significantly.

Chapter 6

Conclusion

At the beginning we mentioned there is no excessive research about influence of parameters of cryptographically strong elliptic curves. We considered 6 variants of baby-step giant-step algorithms. These algorithms solved the discrete logarithm problem on elliptic curves which we generated. The process of generation led us to track cofactor k which defines the divisor of the elliptic curve order. Elliptic curves were sorted according to the cofactor k and the performance of every algorithm was measured on every elliptic curve. It made 109200 instances of ECDLP across 4 values of the k and 91 elliptic curves of order $\approx 2^{34}$ for every such a value.

The impact of elliptic curve form was tested using 30000 instances of ECDLP across 60 elliptic curves of order $\approx 2^{36}$ so larger elliptic curves than in case of influence of the cofactor k.

This thesis covers the connection between solving ECDLP and properties of cryptographically elliptic curves. It is important to examine the influence of cofactor k on performance of solving DLP more deeply. The results suggests there are significant impact on performance based on the selection of the form of elliptic curve and parameters of elliptic curve.

The research can be expanded by a study of improvements for specific forms of elliptic curves to reveal whether the performance could be enhanced generally or at least for some specific elliptic curves. The research of similarities of DLP instances where two grumpy giants and a baby algorithm achieved exclusive results could be greatly beneficial.

Bibliography

[1] Weierstrass equation of an elliptic curve, 2013.

[2] Harald Baier and Johannes Buchmann. Kangaroos, generation methods of elliptic curves, Aug 2002. https://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1030_Buchmann.evaluation.pdf.

[3] E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 46(8):1853–1859, 1967.

[4] Daniel Bernstein and Tanja Lange. Two grumpy giants and a baby. *The Open Book Series*, 1(1):87–111, 2013.

[5] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, pages 389–405, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. *Advances in Cryptology – ASIACRYPT 2007 Lecture Notes in Computer Science*, page 29–50, Sep 2007.

[7] Peter Jephson Cameron. *Introduction to algebra*. Oxford University Press, 2008.

[8] David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154):587, 1981.

[9] Henri Cohen. *A course in computational algebraic number theory*. Springer, 1995.

[10] Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(03):393–423, Sep 2007.

[11] Kenny Fong, Darrel Hankerson, Julio López, and Alfred Menezes. Field inversion and point halving revisited. *IEEE Transactions on Computers*, 53:1047–1059, 2004.

[12] Steven D. Galbraith. *Mathematics of public key cryptography.* Cambridge University Press, 2012.

[13] Steven D. Galbraith, Ping Wang, and Fangguo Zhang. Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm. Cryptology ePrint Archive, Report 2015/605, 2015. `https://eprint.iacr.org/2015/605`.

[14] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers.* Clarendon Press, 1979.

[15] Tomáš Kalvoda, Ivo Petr, and Štěpán Starosta. Materiály k předmětu mi-mky.

[16] Donald Ervin Knuth. *Seminumerical algorithms.* Addison-Wesley, 2016.

[17] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[18] Neal Koblitz. *A course in number theory and cryptography.* Springer, 1994.

[19] A. J. Menezes, Van Oorschot Paul C., and Scott A. Vanstone. *Handbook of applied cryptography.* CRC Press, 2001.

[20] Victor S. Miller. Use of elliptic curves in cryptography. *CRYPTO. Lecture Notes in Computer Science*, 85:417–426, 1985.

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

[22] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. Elliptic curves with the montgomery-form and their cryptographic applications. *Public Key Cryptography Lecture Notes in Computer Science*, page 238–257, 2000.

[23] J. M. Pollard. Kangaroos, monopoly and discrete logarithms, Aug 2000. `http://math.boisestate.edu/~liljanab/Crypto2Spring10/PollardKangaroo.pdf`.

[24] John Pollard. Monte carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[25] René Schoof. Counting points on elliptic curves over finite fields. *Journal de Théorie des Nombres de Bordeaux*, 7(1):219–254, 1995.

[26] Lawrence C. Washington. *Elliptic curves: number theory and cryptography.* Chapman and Hall CRC, 2008.

Appendix A

Acronyms

BSGS Baby-step Giant-step

EC Elliptic Curve

ECDLP Elliptic Curve Discrete Logarithm Problem

Appendix B

Contents of enclosed CD

- `Code` ..... source code directory
  - `bs_gs.jl` ..... implementation of algorithms
  - `dlp_solver.jl` ..... testing script
  - `ec_generator.jl` ..... implementation of generation of elliptic curve
  - `ecw_operations.jl` ..... implementation of operations on Weierstrass elliptic curve
  - `ece_operations.jl` .. implementation of operations on Edwards elliptic curve
  - `ecm_operations.jl` ..... implementation of operations on Montgomery elliptic curve
- `LaTeX` ..... the directory of LaTeX source codes of the thesis
  - `FITthesis.cls` ..... thesis template
  - `cvut-logo-bw.pdf` ..... school logo
  - `mybibliographyfile.bib` ..... bibliography source file
  - `holecma9_dip.tex` ..... thesis source file
- `holecma9_dip.pdf` ..... the thesis text in PDF format