BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

BACHELOR'S THESIS

Brno, 2021 Daniel Štark

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

WEB APPLICATION ON ELLIPTIC CURVE CRYPTOGRAPHY

WEBOVÁ APLIKACE KRYPTOGRAFIE ELIPTICKÝCH KŘIVEK

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	Daniel Štark
SUPERVISOR VEDOUCÍ PRÁCE	M.Sc. Sara Ricci, Ph.D.

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Daniel Štark	***ID:*** 211814
Ročník: 3	***Akademický rok:*** 2020/21

NÁZEV TÉMATU:

Webová aplikace kryptografie eliptických křivek

POKYNY PRO VYPRACOVÁNÍ:

Nastudujte teorii eliptických křivek (tj. definici křivky, vlastnosti, operace nad eliptickou křivkou, kryptografické protokoly využívající eliptické křivky). Výstupem práce bude implementace webové aplikace umožňující spuštění operací nad eliptickou křivkou a základních kryptografických protokolů využívajících eliptických křivek (např. Diffie-Hellmanův protokol). Jako výchozí bod lze využít programovací jazyk Sage. Webová aplikace může být postavena na předchozí studentské práci (desktopová aplikace pro kryptografii eliptických křivek). Součástí práce bude také manuál popisující funkcionalitu webové aplikace.

DOPORUČENÁ LITERATURA:

[1] Washington LC., “Elliptic curves: number theory and cryptography.” CRC press; 2008 Apr 3.

[2] Menezes AJ, Katz J, Van Oorschot PC, Vanstone SA. Handbook of applied cryptography. CRC press; 1996 Oct 16.

Termín zadání: 1.2.2021	***Termín odevzdání:*** 31.5.2021

Vedoucí práce: M.Sc. Sara Ricci, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRACT

Elliptic Curve Cryptography is currently the most used form of public-key cryptography. Theoretical part of this thesis is divided to two chapters. The first chapter describes important topics from algebra and number theory, on which the Elliptic Curve Cryptography is built. This includes groups, finite fields, elliptic curves themselves and the mathematical principles of two well-known and used protocols – ECDH and ECDSA. The second chapter describes the tools, which were used for implementation of user-friendly web application, capable of simulating fundamental operations on elliptic curves and the aforementioned protocols. Key tools, which are introduced in this chapter, are mathematics software system *SageMath* and framework *Spring*, used for implementation of web applications in Java. The third chapter of this thesis describes the way the introduced tools were used, ergo the implementation of the web application itself.

KEYWORDS

Elliptic Curve Cryptography, Web application, *SageMath*, *Spring*, ECDH, ECDSA

ABSTRAKT

Kryptografie na eliptických křivkách je v současné době nejpoužívanější formou asymetrické kryptografie. Teoretická část této práce je rozdělena na dvě kapitoly. První kapitola vysvětluje vybraná témata z algebry a teorie čísel, na kterých je kryptografie na eliptických křivkách postavená. Konkrétně se jedná o grupy, konečná tělesa, eliptické křivky a matematické principy dvou známých a hojně používaných protokolů – ECDH a ECDSA. Druhá kapitola se zabývá popisem nástrojů, které byly použity k implementaci uživatelsky přívětivé webové aplikace, umožňující simulaci jak základních operací na eliptické křivce, tak i dvou výše zmíněných protokolů. Stěžejními nástroji, představenými v této kapitole, jsou matematický systém *SageMath* a framework *Spring*, určený k tvorbě webových aplikací v jazyce Java. Třetí kapitola této práce popisuje jak byly představené nástroje použity, tedy samotnou implementaci webové aplikace.

KLÍČOVÁ SLOVA

Kryptografie na eliptických křivkách, Webová aplikace, *SageMath*, *Spring*, ECDH, ECDSA

ŠTARK, Daniel. *Web application on elliptic curve cryptography*. Brno, 2021, 65 p. Bachelor's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications. Advised by M.Sc. Sara Ricci, Ph.D.

Typeset by the `thesis` package, version 4.00; `http://latex.feec.vutbr.cz`

ROZŠÍŘENÝ ABSTRAKT

Tato bakalářská práce se zabývá kryptografií na eliptických křívkách. Cílem práce je nastudovat a shrnout teorii eliptických křivek a následně znalost této teorie využít k implementaci uživatelsky přívětivé webové aplikace, která umožní studentům si rychle zkontrolovat správnost svých vlastních výpočtů základních operací na eliptické křivce. Porozumění těmto operacím je klíčové pro pochopení principu, na kterém je postavena kryptografie na eliptických křivkách. Webová aplikace rovněž umožňuje simulaci dvou v praxi velmi používaných protokolů – Diffieho-Hellmanova protokolu s využitím eliptických křivek (Elliptic Curve Diffie-Hellman, dále jen ECDH) a protokolu digitálního podpisu s využitím eliptických křivek (Elliptic Curve Digital Signature Algorithm, dále jen ECDSA). Hlavním přínosem této aplikace je zpřístupnění výkonného matematického softwaru, který je velmi často nepřehledný a uživatelský nepřívětivý, studentům, kteří se zajímají o kryptografii na eliptických křivkách. Aplikace tedy umožňuje studentům soustředit se na to, co je zajímá, a nemuset se zabývat složitým syntaxem a nepřehledným rozhraním různých matematických systémů.

Teoretická část je rozdělena na kapitolu matematickou a kapitolu softwarovou. V kapitole matematické jsou nejprve představeny grupy. Grupa je základní algrebraická struktura, definována množinou prvků a jednou binární operací. Grupy jsou důležité pro kryptografii na eliptických křivkách, protože samotná eliptická křivka definovaná na konečném tělesu není nic jiného, než cyklická grupa. Zvláštní pozornost je věnována generátoru grupy, neboť se jedná o jeden z hlavních parametrů asymetrických kryptosystému, včetně ECDH a ECDSA. Dále jsou v matematické kapitole představeny konečná tělesa, algebraické struktury, na kterých jsou eliptické křivky definovány. Nejdelší část kapitoly je věnována eliptickým křivkám samotným. Představeny jsou dva nejpoužívanější tvary eliptických křivek – obecný Weierstrassův (Generalized Weierstrass, dále jen GW) tvar a krátký Weierstrassův (Short Weierstrass, dále jen SW) tvar. Zmíněna je také existence dalších méně známých tvarů, jako příklad je uveden Barretův-Naehrigův (dále jen BN) tvar eliptické křivky. Dále je detailně popsána binární operace sčítání, která charakterizuje grupu tvořenou body na eliptické křivce, a z ní odvozená operace násobení, která je jádrem kryptografie na eliptických křivkách. Výhodné vlastnosti násobení bodů na eliptické křivce popisuje problém diskrétního logaritmu na eliptické křivce (Elliptic Curve Discrete Logarithm Problem, dále jen ECDLP), na kterém je postavena bezpečnost protokolů na eliptických křivkách. Matematická kapitola končí popisem dvou z těchto protokolů, a to již zmíněných ECDH a ECDSA.

Druhá kapitola teoretické části popisuje sofwarové nástroje, které byly použity k implementaci webové aplikace, hlavního výstupu této práce. Prvně jsou však specifikovány vlastnosti, které má tato aplikace splňovat. Detailně je představen

open-source matematický systém *SageMath*, který je použit jako back-end webové aplikace. Pozornost je věnována metodě, kterou předešlý student využil k propojení jeho desktopové JavaFX aplikace s *SageMath* serverem, spuštěném na *localhostu*. Zadání této práce doporučuje využití stejné metody k propojení webové aplikace s *SageMath* back-endem. Dále jsou popsány dva moduly frameworku *Spring*, a to *Spring* Web a *Thymeleaf*, které jsou použity k implementaci samotné webové aplikace v jazyce Java. Softwarová kapitola končí představením volně dostupného frameworku *Bootstrap*, jehož kaskádové styly jsou použity k zajištění uspokojivého vzhledu webové aplikace.

Třetí kapitola se věnuje samotné implementaci webové aplikace. Nejprve je zdůvodněno proč byl použit právě *SageMath* jako back-end webové aplikace. Poté je krátce popsán pokus o využití *SageMathCell* jako back-endu webové aplikace, v tomto přístupu však byly rychle odhaleny velké nedostatky. Dále je popsána reimplementace již zmíněné metody, kterou použil jiný student k implementaci jeho JavaFX aplikace, protože tato metoda se ukázala jako nejlepší možné řešení. Jádro třetí kapitoly tvoří detailní popis webové aplikace. Webová aplikace je nejprve popsána obecným způsobem, princip fungování je pak pomocí množství screenshotů popsán ještě jednou na praktickém příkladu sčítání bodů a výpočtu ECDSA na SW eliptické křivce. Ostatní operace fungují velice obdobně, proto je není potřeba popisovat zvlášť.

Výsledná webová aplikace podporuje křivky ve třech tvarech (SW, GW a BN), avšak křivky všech tvarů je možno jednoduše vyjádřit v GW tvaru. Používání aplikace je poměrně snadné a intuitivní. Uživatel si nejprve zvolí formu eliptické křivky, na které chce provádět výpočty. Je mu zobrazena obecná rovnice formy, kterou si vybral. Do této rovnice pak na vynechaná místa dosadí koeficienty, které definují eliptickou křivku dané formy. Uživatel si tedy nemusí pamatovat značení jednotlivých koeficientů rovnice křivek, které je v případě GW formy poměrně matoucí. Dále uživatel zadá číslo charakterizující konečné těleso, na kterém má být eliptická křivka definována. V případě, že uživatel zadá neplatné parametry (nesingulární křivka nebo číslo, které necharakterizuje konečné těleso) je na tyto nedostatky upozorněn a vyzván, aby zadal jiné parametry. V případě, že zadané parametry jsou validní, je křivka načtena. Uživateli se zobrazí rovnice křivky, kterou zadal, její řád a její body, přehledně roztříděné v tabulce podle jejich řádů. Uživatel si pak může vybrat jednu z pěti operací, kterou chce provést. Body křivky, které chce uživatel použít k nějakému výpočtu, jsou vybírány pomocí dropdown menu ze seznamu, tímto je zamezeno zadání neplatných bodů. Po dokončení operace a zobrazení výsledku si uživatel může vybrat jestli chce provést tu stejnou operaci s jinými body, provést jinou operaci nebo jestli chce změnit parametry křivky.

V případě protokolu ECDSA je implementováno jak vytváření, tak ověřování pravosti digitálních podpisů. ECDSA je složitější než ECDH nebo základní operace na křivce, proto je tato část aplikace opatřena dodatečnými vysvětlivkami, které se mění v závislosti na výsledcích výpočtů protokolu. Například se může stát, že uživatel během vytváření podpisu zvolí nevalidní soukromý klíč. Uživatel je o těchto nedostatcích informován. Dále při ověřování podpisu aplikace rozlišuje mezi situací, kdy je nevalidní zvolený veřejný klíč a kdy je nevalidní podpis, i o tomto je uživatel informován.

Webová aplikace momentálně není spuštěna na žádnem serveru připojenému k internetu, aplikaci si však lze vyzkoušet na předpřipraveném virtuálním stroji. Návod jak aplikaci spustit z virtuálního stroje je součástí příloh této práce.

DECLARATION

ACKNOWLEDGEMENT

I would like to thank my supervisor M.Sc. Sara Ricci, Ph.D for her great guidance, deep knowledge, endless patience and optimistic disposition.

Contents

List of Figures

List of Tables

Introduction

This thesis is devoted to Elliptic–Curve Cryptography (ECC), which has quickly become the dominant approach for designing public-key cryptosystems. ECC offers a partial solution to decrease of performance of public-key cryptosystems caused by enormous key sizes. ECC can provide comparable level of security as the original methods of public-key cryptography, but with significantly smaller key sizes. This is caused by the fact that the mathematical problems, which are used for designing one-way functions, are more complex and difficult to solve on elliptic curves than on prime fields, which were used originally. This added complexity however comes with a price. The operations on elliptic curves are slightly more abstract and could be harder to grasp.

The main goal of this thesis is to provide students with a tool (in a form of a web application), which will help them understand the ECC. The web application should be capable of computing operations used in ECC and displaying results to users, as well as simulating popular ECC protocols. The web application should support elliptic curves with user-defined parameters, as the curves used in practise are not suited to be used as a learning material. There is a great emphasis on the application being user-friendly.

The thesis is divided to three chapters. Chapter 1 covers the mathematical background of elliptic curves, Chapter 2 describes the qualities of software tools, which were explored during the implementation of the web application, and Chapter 3 provides author's commentary on what was achieved with the introduced software tools, including the implementation of the web application itself.

1 Mathematical Background

This chapter covers theoretical knowledge from several fields of mathematics which finds its use in Elliptic Curve Cryptography.

1.1 Groups

A Group $(G, \cdot)$ [1] is a basic algebraic structure with some specific properties defined on a non-empty set of elements and one binary operation. Groups are particularly interesting to ECC, because an **elliptic curve** can be viewed as one, with its **points** being the set of elements.

Definition 1 *Group $(G, \cdot)$ is a set of elements G with binary operation $\cdot$, that associates to each ordered pair (a, b) of elements in G an element $(a \cdot b)$, such that the following axioms are obeyed:*

Axiom 1 *Closure:*

$$\forall\, a, b \in G : \ (a \cdot b) \in G. \tag{1.1}$$

Axiom 2 *Associative:*

$$\forall\, a, b, c \in G : (a \cdot b) \cdot c = a \cdot (b \cdot c). \tag{1.2}$$

Axiom 3 *Identity element:*

$$\forall\, a \in G\ \exists\ e \in G : a \cdot e = e \cdot a = a. \tag{1.3}$$

Axiom 4 *Inverse element:*

$$\forall\, a \in G\ \exists\ a^{-1} \in G : a \cdot a^{-1} = a^{-1} \cdot a = e. \tag{1.4}$$

Group is said to be Abelian (commutative), if it meets additional following axiom:

Axiom 5 *Commutative:*

$$\forall\, a, b \in G : \; a \cdot b = b \cdot a. \tag{1.5}$$

The **order** (cardinality) of a group is defined as the number of elements of the group. Groups, whose order is finite, are called **finite groups**. Order of an elliptic curve (which is a group) is a very important property, because the number of points on the curve directly influences the safety of the EC cryptosystem.

Exponentiation on groups can be defined as repeated application of the groups operation:

$$a^5 = a \cdot a \cdot a \cdot a \cdot a. \tag{1.6}$$

Fact 1 *Every element a from group G, its inverse element a^{-1}, identity element e and every integer n obey these equations:*

$$a^0 = e, \tag{1.7}$$

$$a^{-n} = (a^{-1})^n. \tag{1.8}$$

A **cyclic group** G includes element g, such as every element a of G can be denoted as a power of g to n, where n is an integer:

$$g^n = a. \tag{1.9}$$

The element g is said to generate the group G and is called **generator** of group G. **Order of an element** is described as the number of elements that can be generated by its exponentiation. Therefore, order of a generator is equal to the order of its group. Elements of G, that are not able to generate the entire group, are said to generate a **subgroup** of G. Orders of elements divide the order of the group.

Points on an elliptic curve over finite field form a cyclic group. The generator of the elliptic curve is very important since it is used as the publicly-known base point of the cryptosystem.

1.2 Fields

Fields [2] are the algebraic structures over which the elliptic curves are defined. Finite fields in particular are important to cryptography, because they allow accessing several mathematical problems, which can be used for designing trapdoor (one-way) functions. Examples of such problems are **discrete logarithm problem** or **integer factorisation problem**.

Definition 2 *A field ($\mathbb{F}$, +, $\times$) consists of a set F with two binary operations denoted as + (addition) and $\times$ (multiplication) on F, satisfying the following axioms:*

Axiom 6 *($\mathbb{F}$, +) is an Abelian group with identity element e_a = 0.*

Axiom 7 *The operation $\times$ is associative.*

Axiom 8 *($\mathbb{F}$, $\times$) \{0} is an Abelian group with identity element e_m = 1, such that*

$$a \times 1 = a = 1 \times a, \forall a \in \mathbb{R}. \tag{1.10}$$

Axiom 9 *Every element $a \in$ (F\{0}, $\times$) has multiplicative inverse a^{-1}, such as*

$$a \times a^{-1} = e_m = 1. \tag{1.11}$$

Axiom 10 *The operation $\times$ is distributive over +. That is, for every $a, b, c \in \mathbb{R}$:*

$$a \times (b + c) = (a \times b) + (a \times c), (b + c) \times a = (b \times a) + (c \times a). \tag{1.12}$$

For example, the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$ and the complex numbers $\mathbb{C}$ all form **infinite fields**. However, no element of the set of integers $\mathbb{Z}$ with the exception of 1 and -1 suits the Axiom 9. Therefore the set of all integers $\mathbb{Z}$ does not form a field.

However, the set of non-negative integers less than q, forms a **finite field** under the usual operations $\times$ and + modulo q if and only if q is a **prime number or a power of a prime number** [1]. Such field can be denoted as $\mathbb{F}_q$ [2].

To summarize, a field is a set in which operations of addition and multiplication can be performed without leaving the set. Moreover, subtraction can be defined as the addition of opposite (additive inverse) and division as the multiplication by multiplicative inverse:

$$a - b = a + b_a^{-1}, \tag{1.13}$$

$$a \div b = a \times b_m^{-1}, \tag{1.14}$$

where b_a^{-1} is additive inverse (opposite) of b and b_m^{-1} is multiplicative inverse of b.

1.3 Finite fields

As was stated in previous section, finite fields [2, 3] are crucial structures in public-key cryptography, because of their very special properties. They are the core of cryptosystems which rely on the difficulty of **discrete logarithm problem**, such as ElGamal, Diffie–Hellman scheme and of course ECC.

Definition 3 *A finite field is a field F which contains a finite number of elements. The order of F is the number of elements in F.*

Fact 2 *For any prime p and any positive integer m there exists a finite field with $q = p^m$ elements. This field is unique (up to isomorphism) and is denoted by $\mathbb{F}_q$ or GF(q). GF stands for Galois field.*

The main focus will be on finite fields with $m = 1$, as they are widely used in ECC. These fields are also called **prime** fields and can be denoted as $\mathbb{F}_p$ or GF(p), where p is a prime number. To ensure the prime field $\mathbb{F}_p$ obeys all the axioms stated in previous text, both operations ($+$ and $\times$) have to be **modulo** p.

As an example, addition, multiplication and inverses for each element of $\mathbb{F}_7$ are depicted in Tables 1.1 and 1.2. Note that multiplicative inverse of 0 does not exist. This exception is covered in Axiom 9.

Tab. 1.1: Addition and multiplication in prime field $\mathbb{F}_7$.

$+$	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

$\times$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Tab. 1.2: Additive and multiplicative inverses in prime field $\mathbb{F}_7$.

a	$-a$	a^{-1}
0	0	–
1	6	1
2	5	4
3	4	5
4	3	2
5	2	3
6	1	6

1.4 Elliptic curve

Elliptic curves [3, 4] are the central objects of ECC. Their points can be viewed as a set of solutions of specific equation. These solutions together with an extra point, called **point at infinity** also form an Abelian group under the operation of point addition (+). Elliptic curves defined over finite fields allow accessing **Elliptic Curve Discrete Logarithm Problem** (ECDLP) and use it to create so called **trapdoor function**, core of any public-key cryptosystem.

Definition 4 *An elliptic curve E over a field K is given by generalized Weierstrass (GW) equation*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1.15}$$

where coefficients $a_1, \ldots, a_6 \in K$. We are going to assume K is a prime field $\mathbb{F}_p$ from now on. The equation of some curves can be transformed to short Weierstrass (SW) form

$$E : y^2 = x^3 + Ax + B, \tag{1.16}$$

if the characteristic of the field $\mathbb{F}_p$, denoted char($\mathbb{F}_p$), is **greater** than 3. Note that char($\mathbb{F}_p$) is equal to p, this implies that only elliptic curves defined on a prime field with $p > 3$ can be written in short Weierstrass form.

Elliptic curves are **smooth** (nonsingular), this means that their discriminant Δ can not be equal to 0. Discriminant of an elliptic curve in short Weierstrass form can be computed as

$$\Delta = -16(4A^3 + 27B^2), \quad \Delta \neq 0. \tag{1.17}$$

Elliptic curves can have points in any field, such as $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{C}$. The main focus will be on points in prime field $\mathbb{F}_p$, as it is the one used in ECC. Figures 1.1 and 1.2 show points of an elliptic curve over $\mathbb{Q}$ and $\mathbb{F}_{79}$, respectively.

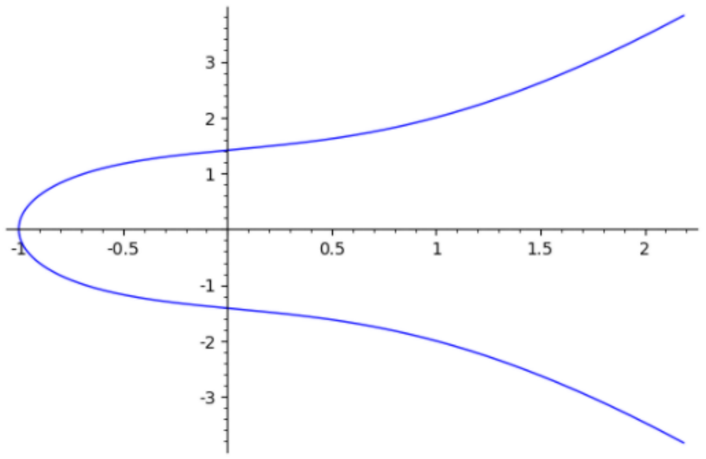


Fig. 1.1: Elliptic curve $y^2 = x^3 + x + 2$ over $\mathbb{Q}$.

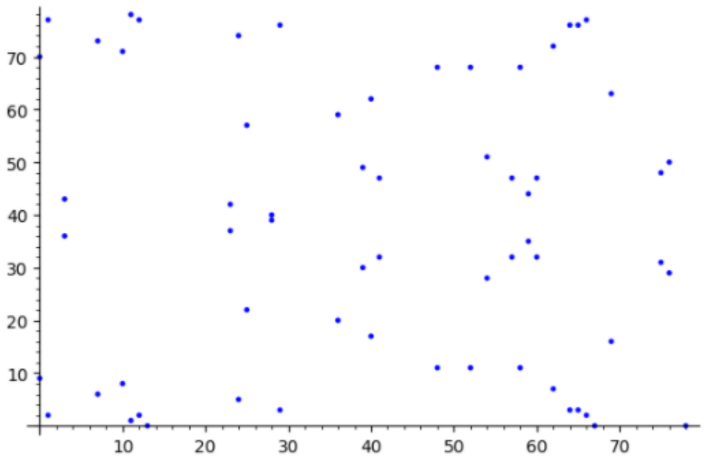


Fig. 1.2: Elliptic curve $y^2 = x^3 + x + 2$ over prime field $\mathbb{F}_{79}$.

1.5 Another forms of elliptic curves

As stated earlier, every curve can be expressed in generalized Weierstrass form. Short Weierstrass form, the most used and best known representation of an elliptic curve, is just a special case of GW form, where coefficients a_1, a_2 and a_3 are left equal to 0. Many more forms of elliptic curves can be derived from GW form in a similar manner – for example if we leave all the coefficients but a_6 equal to 0, we get Barreto-Naehrig (BN) form of an elliptic curve:

$$E : y^2 = x^3 + B. \tag{1.18}$$

Every form of elliptic curves has its distinct properties and usages, for example Barreto-Naehrig elliptic curves are well suited to be used in pairing-based protocols.

1.6 Point addition

The main operation [3], which characterises the group of points located on the curve, is called **point addition** and denoted by $+$. The **point at infinity** P_∞ is the identity element of the group. The result of point addition is computed differently based on the properties of points on which the addition is performed on. There are 4 different cases of point addition, which are introduced in next sections.

1.6.1 Addition of two distinct points

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two distinct **points** on an elliptic curve, $P, Q \neq P_\infty$ and $x_1 \neq x_2$. To evaluate $P + Q = R$, we draw the **line connecting them**. There is a third point of intersection with the curve, that is where $-R$ is located. **Reflecting** the point at the x-axis gives point R. This is shown in Figure 1.3.

The actual **addition formulas** can be derived from the provided geometric definition. Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $R = P + Q = (x_3, y_3)$ be three points on an elliptic curve with $x_1 \neq x_2$. The line intersecting points P and Q has **slope**

$$\lambda = \frac{y_1 - y_2}{x_1 - x_2}. \tag{1.19}$$

The coordinates of point R can be computed as

$$x_3 = \lambda^2 + a_1\lambda - a_2 - x_1 - x_2, \tag{1.20}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 - a_1 x_3 - a_3, \tag{1.21}$$

where a_1, a_2 and a_3 are coefficients of the curve in generalized Weierstrass form.

The formulas for x_3 and y_3 can be **simplified** to

$$x_3 = \lambda - x_1 - x_2, \tag{1.22}$$

$$y_3 = \lambda(x_1 - x_3) - y_1, \tag{1.23}$$

if the curve is given in short Weierstrass (or Barreto-Naehrig) form.

Note that operations subtraction and division are not defined in **finite fields**, they have to be replaced by addition of additive inverse mod p and multiplication by multiplicative inverse mod p respectively.

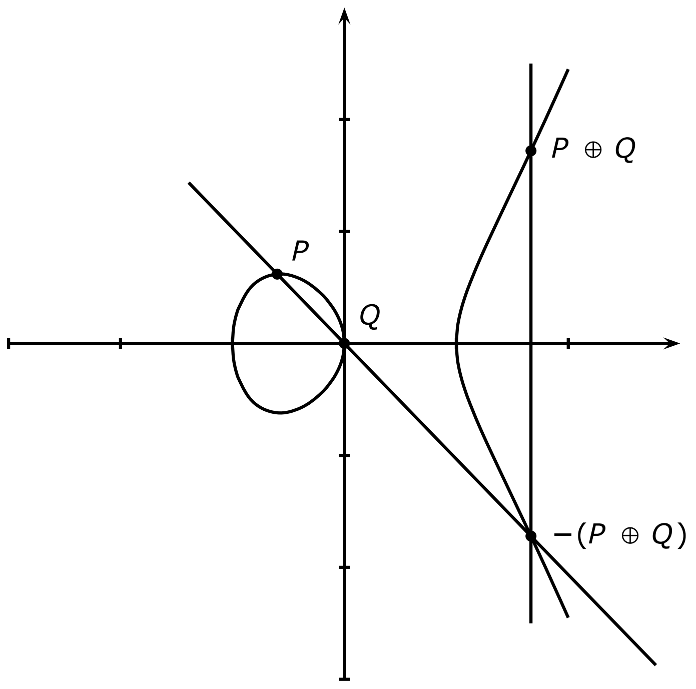


Fig. 1.3: Point addition on an elliptic curve [3].

1.6.2 Addition of inverse points

The sum of two different points, whose x-coordinates are equal, has to be calculated in a different way, as the line connecting the points does not intersect the curve in any other point. Instead, the product of $P + (-P) = (x_1, y_1) + (x_1, y_2)$ is equal to P_∞, the **identity element**. This case of point addition can be viewed as addition of inverse element described in Formula 1.4.

1.6.3 Addition of point at infinity

The point at infinity P_∞ is the identity element of the group. Addition of P_∞ to any other point (even to itself) always results in the other point: $P_\infty + P = P$.

1.6.4 Addition of point to itself

Point doubling, denoted $2P$, can be seen as **adding point to itself**. However, as there are no two separate points to intersect with one line, a **tangent** to the curve that intersects P is used instead. Similarly to the first case of addition, second intersection of the tangent with the curve gives point $-2P$, which has to be **reflected** at the x-axis in order to find $2P$. Point doubling is depicted in Figure 1.4.

The **slope** λ of the tangent line at point $P(x_1, y_1)$ can be computed as

$$\lambda = \frac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3}, \tag{1.24}$$

where a_1, a_2, a_3 and a_4 are coefficients of the curve in generalized Weierstrass form.

The formula for computing the **slope** λ of the tangent line can be **simplified** to

$$\lambda = \frac{3x_1^2 + A}{2y_1}, \tag{1.25}$$

if the curve is given in short Weierstrass form and A is its coefficient. In case of Barreto-Naehrig form, the formula can be further simplified to

$$\lambda = \frac{3x_1^2}{2y_1}. \tag{1.26}$$

Formulas for finding the coordinates of $2P(x_2, y_2)$ are the same formulas used for coordinates of point $R = P + Q$, (Formulas 1.20 and 1.21) for curve in GW form and (Formulas 1.22 and 1.23) for curve in SW and BN form.

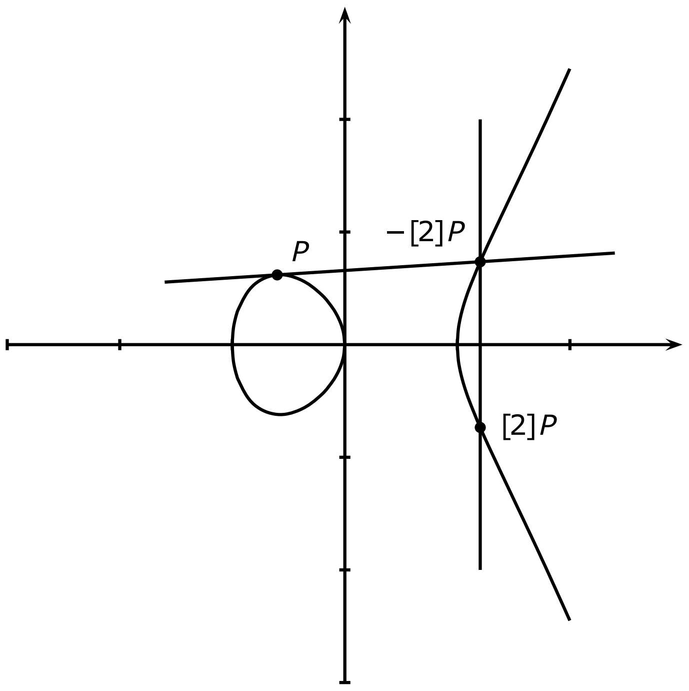


Fig. 1.4: Point doubling on an elliptic curve [3].

1.6.5 Scalar multiplication

Scalar multiplication, denoted nP, where $n \in \mathbb{N} \setminus \{0\}$, is defined as repeated addition of point P to itself n times:

$$R = nP = \underbrace{P + P + P + \ldots + P}_{n}. \tag{1.27}$$

This is the main operation used in ECC. In general scenario, R is the **public key**, n is the **private key** and P is a **generator** of the group of points.

Computing scalar multiplication simply by adding P to itself n times is extremely inefficient, because the scalar n is usually a 256 bit number in real-world implementations. Performing multiplication in this manner would require $2^{256} - 1$ operations, which is not feasible on today computers.

There are some algorithms, which can significantly reduce the number of operations needed to calculate the multiplication. One such algorithm is the **double-and-add** algorithm [5]. This algorithm utilizes a combination of doublings and additions to perform the multiplication. It is best explained by giving an example: Let us say we want to multiply point P by scalar $n = 19$. The most obvious way to do this is to add P to itself $n - 1 = 18$ times:

$$19\,P = P+P+P+P+P+P+P+P+P+P+P+P+P+P+P+P+P+P+P.$$

Same computation with the use of the double-and-add algorithm can be reduced just to 2 additions and 4 doublings:

$$19\,P = 2\,(\,2\,(\,2\,(\,2\,P\,)\,)+P\,)+P.$$

In the worst case scenario, the number of operations needed to perform multiplication using this algorithm is $2(\log_2 s - 1)$, opposed to $s-1$ operations using the naive approach.

Example implementation of this algorithm is depicted in Figure 3.1. The downside [6] of this algorithm is its vulnerability to side-channel attacks, because the value of scalar n directly influences the number of operations needed to evaluate the multiplication. The algorithm can be modified to require constant number of operations.

1.7 Elliptic curve discrete logarithm problem

One of the key features of public–key cryptosystem is the inability to compute a secret key from a public key. In terms of ECC, this means **reversing the operation of scalar multiplication**: computing a scalar n from given points P and $R = nP$.

Definition 5 *The Elliptic Curve Discrete Logarithm Problem (ECDLP) [7] is: given an elliptic curve E defined over a finite field $\mathbb{F}_q$, a point $P \in E(\mathbb{F}_q)$ of order o, and a point $R \in E(\mathbb{F}_q)$ find the integer $n \in$ [0, $o-1$] such that $R = nP$. The integer n is called the discrete logarithm of R to the base P, denoted $n = \log_P R$.*

$$R = nP$$
$$n = ?$$

Computing discrete logarithm is **very difficult problem** with no effective algorithmic solutions, if the following conditions are met:

- order of the curve is big enough,
- order of the point P is big enough (P is preferably the generator of $E(\mathbb{F}_q)$, so its order is highest possible),
- scalar n is big enough.

The most naive and simple algorithm for solving the ECDLP is **brute-forcing** which lies in computing the sequence of points $P, 2P, 3P, 4P, \ldots$ until $R = nP$ is encountered. The run time of this algorithm is o (order of the point P) operations in the worst case scenario, $\frac{o}{2}$ on average. That means the order of the base point P should be big enough to make the performance of $\frac{o}{2}$ operations unfeasible, 2^{80} is considered big enough.

There are however other more sophisticated algorithms for solving the ECDLP, the combination of **Pohlig-Hellman** and **Pollards rho** algorithms being the most effective. These algorithms require run time of $\sqrt{p}$ operations, where p is the largest prime divisor of o. To resist these attacks, largest prime divisor p of the order o of the base-point P should be larger than 2^{160}.

The discrete logarithm problem is harder in the group of points on an elliptic curve than in a multiplicative group $\mathbb{Z}_p^*$. Because of this, elliptic curve cryptosystems are able to function with significantly **smaller key sizes** while providing the same level of security as their $\mathbb{Z}_p^*$ counterparts. Smaller key sizes make the elliptic curve cryptosystems **faster** than their predecessors. Table 1.3 compares key sizes of symmetric cryptosystems, standard $\mathbb{Z}_p^*$ public-key cryptosystems and EC cryptosystems for the same level of security. Today's recommended key sizes are highlighted.

1.8 Elliptic curve Diffie-Hellman protocol

Diffie-Hellman key exchange [2] is a public-key cryptography protocol used for establishing secret symmetric key between two parties via insecure channel. The security of this protocol rests on the intractability of the **Diffie-Hellman problem**, which is closely related to **discrete logarithm problem**. Diffie-Hellman problem makes the extraction of the secret symmetric key impossible even for eavesdroppers who

Tab. 1.3: Comparison of key lengths in bits for same level of security of symmetric cryptosystems, standard $\mathbb{Z}_p^*$ public-key cryptosystems and EC cryptosystems [9].

Symmetric	Standard $\mathbb{Z}_p^*$	EC
80	1024	160
112	2048	224
128	**3072**	**256**
192	7680	384
256	15360	512

have access to full communication between the two parties. It finds its use for example in protocol TLS (Transport Layer Security), which is used for securing internet communication.

The Diffie-Hellman protocol can be carried out in any group in which the discrete logarithm is hard problem, the most obvious one being the multiplicative group $\mathbb{Z}_p^*$. Protocol in $\mathbb{Z}_p^*$ functions as follows:

1. prime p and generator g of $\mathbb{Z}_p^*$ are selected and published
2. Alice chooses a random private key a , computes public key $A = g^a \bmod p$, sends A to Bob via insecure channel.
3. Bob chooses a random private key b, computes public key $B = g^b \bmod p$, sends B to Alice via insecure channel.
4. Alice computes $B^a = (g^b)^a \bmod p = K$, Bob computes $A^b = (g^a)^b \bmod p = K$. K is the secret symmetric key they have established.

The Diffie-Hellman problem can be explained like this: given g^a and g^b, compute g^{ab}. Attacker would need to compute a or b from g^a or g^b first, but this would mean solving the **discrete logarithm problem** in $\mathbb{Z}_p^*$. This means that the Diffie-Hellman problem and dicrete logarithm problem are very closely related.

Elliptic Curve Diffie-Hellman protocol (ECDH) works in a very similar manner, but the nature of elliptic curves allows using smaller key lengths for the same level of security. ECDH protocol functions as follows:

1. Elliptic curve E over finite field (usually prime field) $E(\mathbb{F}_q)$ and generator G of the group, which represents points of E, are selected and published.
2. Alice chooses a random private key a, which is a scalar. Then she computes point $A = aG$, which serves as a public key and sends it to Bob via insecure channel.
3. Bob chooses a random private key b, which is a scalar. Then he computes point $B = bG$, which serves as a public key and sends it to Alice via insecure

channel.

4. Alice computes point $aB = abG = K$, Bob computes point $bA = baG = K$. Secret symmetric key can be derived from point K for example by taking its x-coordinate.

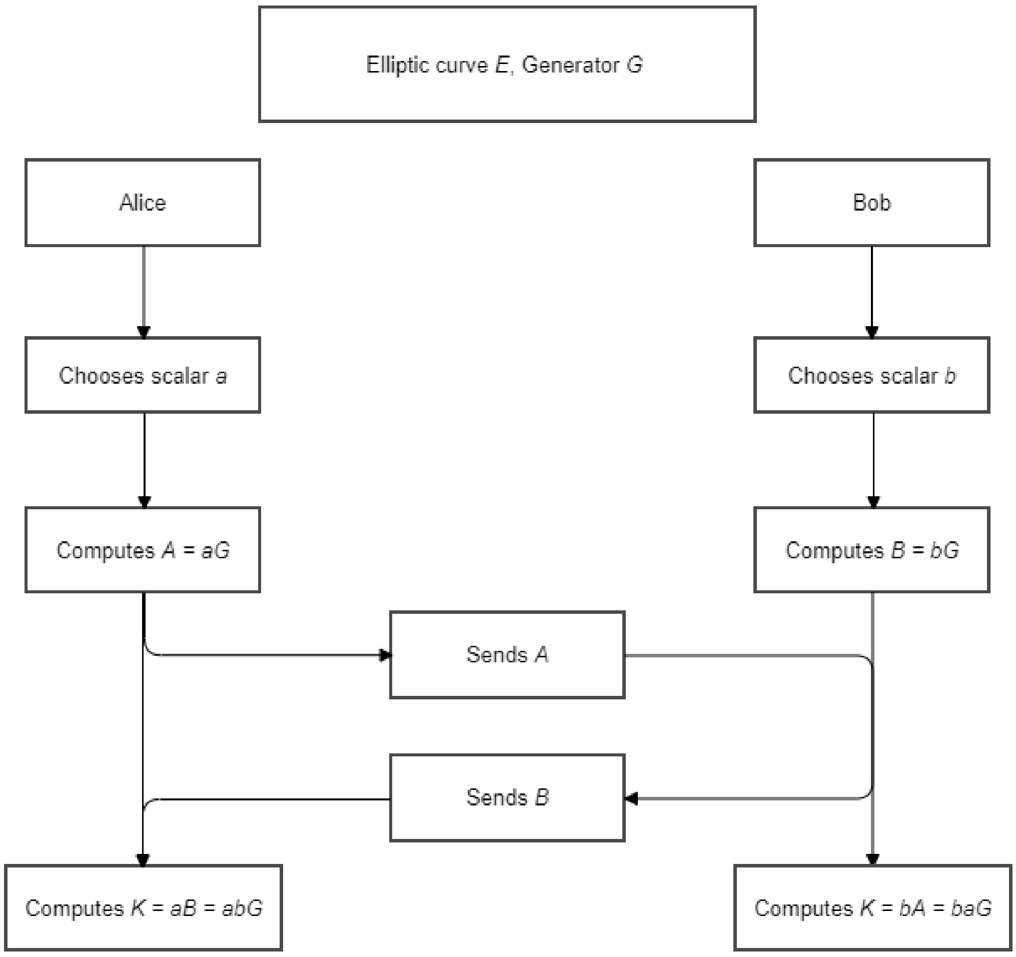


Fig. 1.5: Elliptic curve Diffie-Hellman protocol.

The relation between Elliptic Curve Diffie-Hellman problem and the Elliptic Curve Discrete Logarithm Problem is identical to the relation of the Diffie-Hellman problem and the Discrete Logarithm Problem in $\mathbb{Z}_p^*$. The Elliptic Curve Diffie-Hellman problem can be informally stated as: given points aG and bG, compute point abG. Once again, the solution of the Elliptic Curve Diffie-Hellman problem would be trivial, if it was possible to solve Elliptic Curve Discrete Logarithm Problem – compute a from aG or b from bG.

1.9 Elliptic Curve Digital Signature Algorithm

Elliptic Curve Digital Signature Algorithm (ECDSA) [8] is the elliptic curve version of the well-known Digital Signature Algorithm (DSA). The ECDSA offers better performance compared to its predecessor, as the discrete logarithm problem, which ensures the safety of both of these algorithms, is harder to solve in a cyclic group which represents the points of an elliptic curve over finite field than it is in the multiplicative groups, which are used in DSA. This implies that the relation between ECDSA and DSA is exactly the same as ECDH and DH.

The ECDSA is a public-key cryptography protocol used for generation and verification of digital signatures. These signatures ensure the **authenticity**, **integrity** and **non-repudiation** of the messages they are provided with.

1. Authenticity – message was sent by verified sender,
2. integrity – message was not changed in any way since it was signed,
3. non-repudiation – sender can not dispute authorship of a message signed with his signature.

1.9.1 Key generation

The given domain parameters of ECDSA protocol consist of elliptic curve E defined over prime field $\mathbb{F}_p$ and base point G with sufficiently large **prime** order n. User A can generate his own key pair associated with these domain parameters by following these steps:

1. choose a random integer d, $d \in [1, n-1]$,
2. compute point $Q = dG$,
3. d and Q are A's private and public keys respectively.

1.9.2 Signature generation

In order to sign a message m, user A with key pair (d, Q) does the following:

1. generate a random integer k, $k \in [1, n-1]$,
2. compute point $kG = (x, y)$,
3. compute $r = x \bmod n$. If $r = 0$, go to step 1.
4. Compute $k^{-1} \bmod n$,
5. compute hash of the message $h(m)$ and convert it to integer e,
6. compute $s = k^{-1}(e + dr) \bmod n$. If $s = 0$, go to step 1.
7. A's signature for the message m is pair (r, s).

1.9.3 Public key verification

To verify the validity of user A's public key Q, user B needs to obtain the same domain parameters and do the following:

1. check that $Q \neq P_\infty$,
2. check that coordinates of Q are properly represented elements of $\mathbb{F}_p$,
3. check that Q lies on an elliptic curve E,
4. check that $nQ = P_\infty$.

If any check fails, the public key (and any signature associated with it) is not valid.

1.9.4 Signature verification

To verify A's signature for the message m, user B needs to obtain A's public key Q and the same domain parameters A used for the generation of his signature. User B can verify the signature (r, s) attached to message m by following these steps:

1. verify A's public key Q,
2. check that $r, s \in [1, n-1]$,
3. compute hash of the message $h(m)$ and convert it to integer e,
4. compute $w = s^{-1} \bmod n$,
5. compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$,
6. compute point $X = u_1G + u_2Q$,
7. If $X = P_\infty$, reject the signature. Otherwise compute $v = x_1 \bmod n$, where x_1 is x-coordinate of point X.
8. Accept the signature if and only if $v = r \bmod n$.

1.10 Summary

This chapter covered several subjects from algebra and number theory, which all find its use in elliptic curve cryptography. **Groups** were generally described, because the elliptic curve can be viewed as one. Very important attribute of a group is its **generator**, which in case of public-key cryptosystems is one of the main parameters. **Finite fields** were introduced, as they are the structure the elliptic curves are defined over. Then the elliptic curves themselves with their group law **point addition** were covered. Special cases of point addition – **point doubling** and **scalar multiplication** were covered separately and in great detail. Then the elliptic curve discrete logarithm problem, which ensures the safety of elliptic curve cryptosystems, was introduced. To show an example of use of the elliptic curve cryptography, two popular protocols were described–**Elliptic Curve Diffie-Hellman** key exchange and **Elliptic Curve Digital Signature Algorithm**. Safety of both of these protocols is built upon Elliptic Curve Discrete Logarithm Problem. Both ECDH and ECDSA are implemented in the practical part of this thesis.

2 Software Background

This chapter introduces the software tools which were researched and tested during the **implementation** of the web application. Main goal of this chapter is to propose software candidates, which are best suited for implementation of the web application, and describe them from theoretical point of view. Chapter 3 covers author's comments on what was achieved with these tools. Note that these chapters cover even tools, which were not used in the final implementation of the web application, because they were deemed suboptimal during the testing in Chapter 3.

2.1 Goals of the web application

Before researching of any tools is started, it is mandatory to state what is needed to achieve with them. The web application should mainly be used for **education** of students. Application's purpose is to allow students to quickly check whether or not their pen-and-paper calculations (and therefore their understanding of the ECC) are correct. The web application should be capable of **fast and convenient computation** of basic EC operations (point addition, point doubling, and scalar multiplication) described in Section 1.6 on **user-defined curves of multiple forms**, as well as commonly used cryptographic protocols on elliptic curves, such as ECDH (Section 1.8) and ECDSA (Section 1.9). There is great emphasis on the application being **user-friendly** since this quality is not present in popular software solutions which are capable of performing operations on elliptic curves. In other words, the main contribution of this application should be providing a way to use powerful mathematical software without the need to first study the software itself for an extensive amount of time.

2.2 Java

Java is high level object-oriented programming language. This properties make it perfect for creating interactive web applications. Another great quality of Java is its active community, which has created considerable amount of helpful freely available content. In this thesis, Java is used in:

- experimental implementation of the mathematical back-end engine of the web application, covered in Section 3.1,
- implementation of test web application, whose purpose is to test the compatibility between *Spring* web and *SageMath* server, covered in Section 3.5,
- implementation of the final web application, covered in Section 3.6.

2.3 EC Library candidates

One of the viable approaches for designing the mathematical back-end is to use a **library**, which is able to perform operations on elliptic curves. A research on the state-of-the-art EC libraries was conducted. Library *PARI/GP (Pascal ARIthmetic/ Great Programmable calculator)* [10] is best suited for implementing the mathematical back-end of the web application, as it supports GW curves. Full results of the research are covered in Section 3.2 together with author's comments.

2.4 *SageMath*

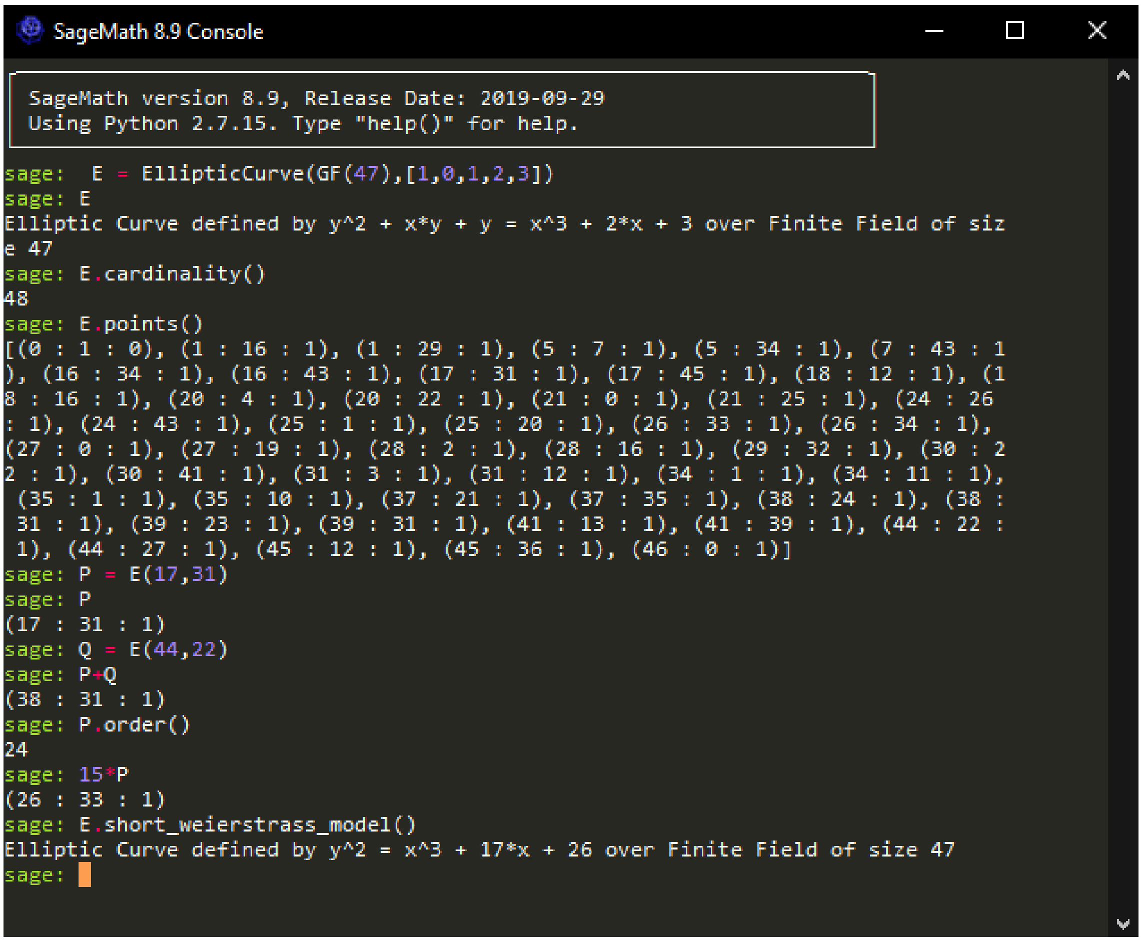
SageMath [11] is a Python-based **mathematics software system** which serves as an interface to numerous open-source mathematical packages. *SageMath's* main goal is providing a software that can be used to explore and experiment with mathematical constructions in algebra, geometry, number theory, calculus, numerical computation etc. to students, teachers and researchers. *SageMath* aims to be efficient and fast, as it uses highly-optimized mature software like *GMP (GNU Multiple Precision arithmetic library), PARI/GP, GAP (Groups, Algorithms, Programming)* and *NTL (Number Theory Library)*. *SageMath*'s components are also commonly used in software engineering practise. *SageMath*'s source code is **freely available** and readable, so users can understand what the system is really doing and more easily extend it. *SageMath* is also flexible enough to run on Windows, Linux and macOS alike.

After installation, there are four possible ways to use *SageMath* from our computer:

1. *SageMath* Console.
2. *SageMath* Notebook.
3. *SageMath* Shell.
4. By writing interpreted and compiled programs in *Sage* language.

2.4.1 *SageMath* Console

Running *SageMath* Console starts a customized version of the IPython shell, and imports many functions and classes, so they are ready to use from the command prompt. After that, Console is ready to receive user's input. Example of *SageMath* Console session is depicted in Figure 2.1. This Figure depicts defining GW elliptic curve E over field $\mathbb{F}_{47}$, printing its order and coordinates of all points, addition of two points, scalar multiplication of a point and transforming the curve to SW form. Another student [12] tried to create a pipe that would pass the input to the console from his Java desktop application, but he was not successful. He provided good explanation why it did not work, so it was decided to not use the console as well.



```
 SageMath version 8.9, Release Date: 2019-09-29
 Using Python 2.7.15. Type "help()" for help.

sage:  E = EllipticCurve(GF(47),[1,0,1,2,3])
sage: E
Elliptic Curve defined by y^2 + x*y + y = x^3 + 2*x + 3 over Finite Field of siz
e 47
sage: E.cardinality()
48
sage: E.points()
[(0 : 1 : 0), (1 : 16 : 1), (1 : 29 : 1), (5 : 7 : 1), (5 : 34 : 1), (7 : 43 : 1
), (16 : 34 : 1), (16 : 43 : 1), (17 : 31 : 1), (17 : 45 : 1), (18 : 12 : 1), (1
8 : 16 : 1), (20 : 4 : 1), (20 : 22 : 1), (21 : 0 : 1), (21 : 25 : 1), (24 : 26
: 1), (24 : 43 : 1), (25 : 1 : 1), (25 : 20 : 1), (26 : 33 : 1), (26 : 34 : 1),
(27 : 0 : 1), (27 : 19 : 1), (28 : 2 : 1), (28 : 16 : 1), (29 : 32 : 1), (30 : 2
2 : 1), (30 : 41 : 1), (31 : 3 : 1), (31 : 12 : 1), (34 : 1 : 1), (34 : 11 : 1),
 (35 : 1 : 1), (35 : 10 : 1), (37 : 21 : 1), (37 : 35 : 1), (38 : 24 : 1), (38 :
 31 : 1), (39 : 23 : 1), (39 : 31 : 1), (41 : 13 : 1), (41 : 39 : 1), (44 : 22 :
 1), (44 : 27 : 1), (45 : 12 : 1), (45 : 36 : 1), (46 : 0 : 1)]
sage: P = E(17,31)
sage: P
(17 : 31 : 1)
sage: Q = E(44,22)
sage: P+Q
(38 : 31 : 1)
sage: P.order()
24
sage: 15*P
(26 : 33 : 1)
sage: E.short_weierstrass_model()
Elliptic Curve defined by y^2 = x^3 + 17*x + 26 over Finite Field of size 47
sage: 
```

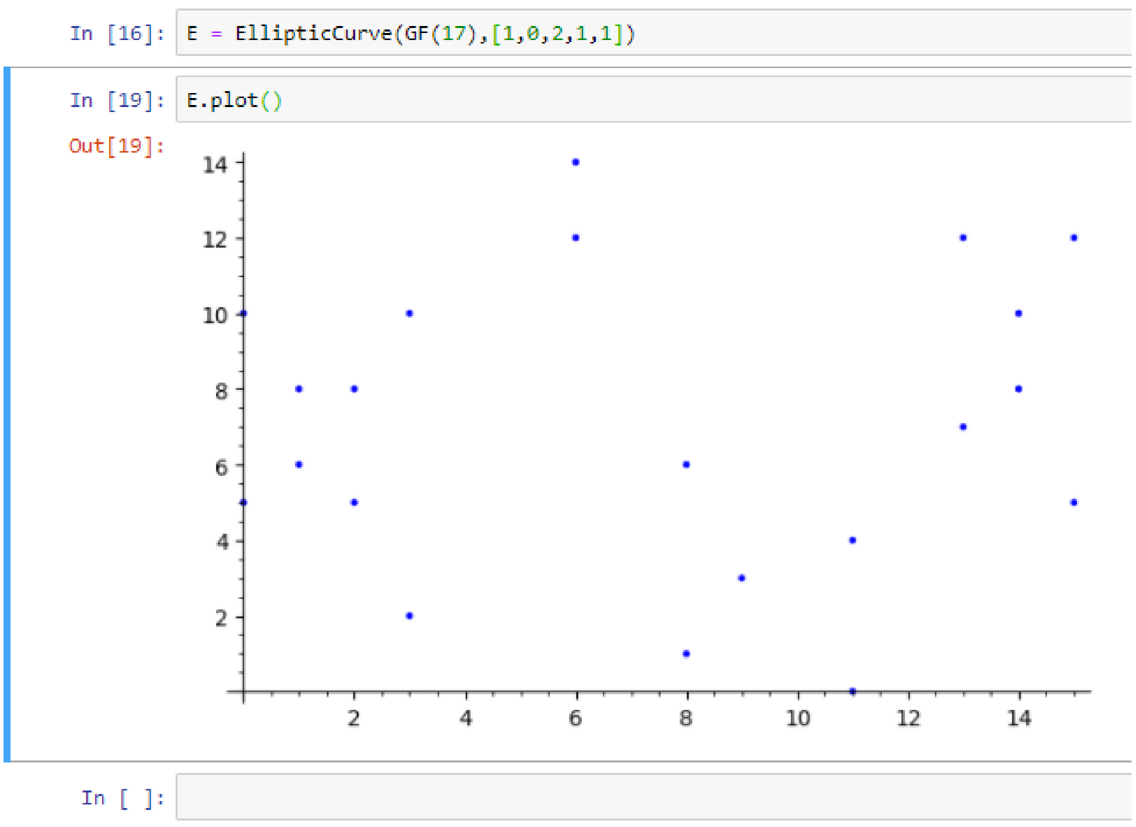
Fig. 2.1: Example of *SageMath* Console session.

2.4.2 *SageMath* Notebook

This option launches Jupyter Notebook on *localhost:8888*. The main advantages [13] of this Notebook are:

1. *In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.*
2. *The ability to execute code from the browser, with the results of computations attached to the code which generated them.*
3. *Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc.*

Figure 2.2 depicts Graphical User Interface of Jupyter Notebook. Notebook allows users for example to conveniently **plot an elliptic curve** over finite field.



```
In [16]: E = EllipticCurve(GF(17),[1,0,2,1,1])

In [19]: E.plot()
Out[19]:
```

```
In [ ]:
```

Fig. 2.2: *SageMath* in Jupyter Notebook.

2.4.3 *SageMath* Shell

This option starts a *UNIX*-like terminal capable of running Python scripts. The Shell is only present in Windows version of *SageMath*, because Linux and macOS are able to run the scripts directly from the Terminal.

2.4.4 *SageMathCell*

There are also several ways to use *SageMath* even without actually having to install it locally. Examples of these services are *CoCalc* [11], which is basically an online version of Jupyter Notebook, and *SageMathCell* [14]. *SageMathCell* can be used to embed *SageMath* computations into HTML (Hyper Text Markup Language) code of a web page. The web page would then send the generated input through the internet to be executed in *SageMathCell* server. This seemed to be exactly something that was needed. There is unfortunately major issue that was discovered during the testing described in Section 3.4.

2.4.5 *SageMath* server connection

In the previous thesis [12] a Python script for *SageMath* Shell is used for starting a *SageMath* server on *localhost.* The implemented Java desktop application then generates an input based on user's actions, sends the input to the *SageMath* server and collects the results. The script was provided by one of the users [16] of the *SageMath* forum. The assignment of this thesis suggests to use the same approach.

The *SageMath* server created by the script on *localhost* is capable of:

1. listening for *SageMath* input on given port,
2. evaluating the input
3. returning the output back to the client

Adopting this solution allows designing a web application which would operate in following steps:

1. user submits his input via front-end of the web application,
2. input is formatted to *SageMath* code in back-end of the web application,
3. formatted input is sent to the *SageMath* server created by the python script,
4. input is executed and evaluated in *SageMath*,
5. *SageMath* output is sent back to the back-end of the web application,
6. output is parsed to user-friendly format and displayed via the front-end.

Sections 3.5 and 3.6 cover reimplementation of this connection in the web application environment and building the web application itself.

2.5 *Spring* framework

Method, that could be used for connecting the web application to *SageMath* server, was covered in previous section. The server can serve as the mathematical back-end of the web application. Framework *Spring*, which will be used in implementation of the web application itself, is introduced in this section.

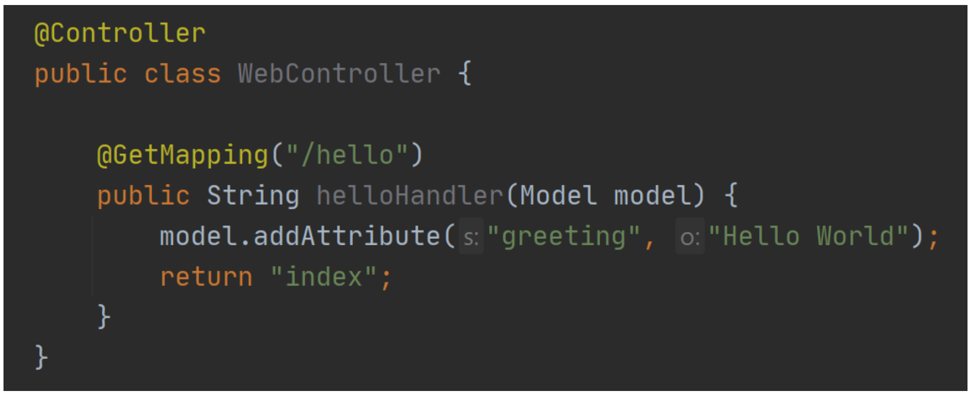
2.5.1 *Spring* framework generally

Spring framework's [17, 18, 19] maturity, power and high flexibility make it the most popular web application development framework for enterprise Java. One of the main advantages of the *Spring* framework is its layered architecture, which allows users to select which of its **modules** they want to use. Only 2 out of the very high amount of various available modules are used in terms of this thesis – *Spring Web* and *Thymeleaf*. Blank *Spring* project can be initialized online at `https://start.spring.io/`, where it is possible to declare which of the modules are needed and automatically generate the Maven or Gradle dependencies. Blank project can be imported to most popular Java IDEs (Integrated Development Environments). It is possible to start a Tomcat server with the web application by pressing a single button in the IDE. This makes testing the web application very convenient.

2.5.2 *Spring* Web

Spring Web [17, 20] is a module of the *Spring* framework which deals with the **Model-View-Controller** pattern. It provides annotation-based programming approach. Classes with annotation *@Controller* are the core of the *Spring Web* application, their purpose is preparing a model map with data and selecting a view to be rendered. Its functionality is best showed on an example.

Figure 2.3 depicts the Controller of Hello World web application written in *Spring Web*. The controller consists of one handler method, which is listening on address */hello*. After visiting this address, handler method called *helloHandler* will add String *greeting* = "Hello World" to the Model of the application. Afterwards, template file, from which the View is rendered, *index.html* is returned and the View is rendered and displayed by user's web browser. The attribute *greeting* can be accessed inside index.html via *Thymeleaf* to print out its value ("Hello World") in the web browser.



```
@Controller
public class WebController {

    @GetMapping("/hello")
    public String helloHandler(Model model) {
        model.addAttribute( s: "greeting",  o: "Hello World");
        return "index";
    }
}
```

Fig. 2.3: Controller of simple *Spring Web* application.

2.5.3 *Thymeleaf*

Thymeleaf [20] is a modern server-side Java template engine for both web and standalone environments. It makes the HTML (Hyper Text Markup Language) templates more **dynamic**, for example by accessing variables from the model of the web application or sending user input to the model. It can even be used for implementing loops and if-statements inside the HTML code.

2.6 HTML5 and CSS

Hyper Text Markup Language 5 is the newest version of standard **markup** language for creating web pages. HTML code consists of tags surrounded with "<>", each tag represents an **element** on a web page. These tags are processed by web browsers in order to render the web page to the user. Cascading Style Sheets (CSS) are used for styling of said HTML files, which usually only define the structure of the web page. In other words, HTML defines the content of web pages, while CSS determine the appearance of the page.

2.7 *Bootstrap*

Bootstrap [21] is one of the most popular frameworks for designing responsive and clean **front-ends** of websites. It includes set of CSS style sheets and several JavaScript libraries. Features like the grid system make the websites powered by *Bootstrap* very clear and user-friendly. Another great quality of *Bootstrap* is the fact that is is open-source and freely available under very permissive MIT license (named after Massachusetts Institute of Technology).

3 Web Application

This chapter covers author's commentary on what was achieved with the tools that were introduced in the previous chapter. Note that not every tool has proven to be useful. Three final sections cover the implementation of the web application, which is the main goal of this thesis.

First major decision that had to be made concerned approach to designing **back-end mathematical engine** for the application. The mathematical engine needed to be capable of several crucial tasks:

1. Creating a curve based on user-defined parameters.
2. Finding points of such curve.
3. Performing point addition, doubling and scalar multiplication.
4. Running in reasonable runtime.

There were three possible approaches for implementing the mathematical engine with varied level of viability:

1. Implementation without dependence on existing EC libraries.
2. Implementation using EC library.
3. *SageMath* software system.

3.1 Implementation without dependence on existing libraries

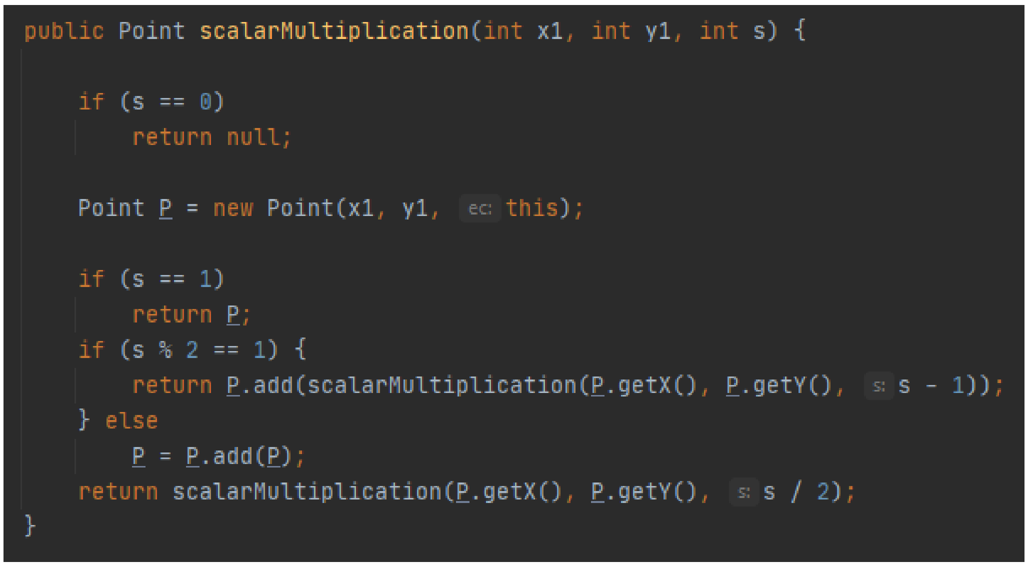
First possible approach involved implementing the back-end in one of the available programming languages without the use of external library. This was without question the least optimal and fast approach, because solid implementation of elliptic curve operations would require a lot of low-level programming in order to make it competitive with already existing implementations mentioned in Table 3.1. This however is out of the field of this thesis, as its main goal is to create user-friendly web application that would operate only in numbers of relatively small size.

That being said, it was decided to shortly explore this approach and implement basic functionalities of ECC anyway. The main reason for this was the fact that there were no existing libraries which would allow user to define his own parameters for curves other than those of SW and GW forms. For more details see Section 3.2. The language of choice is Java, as its high-level nature makes designing web applications more convenient than C++ for example, which would possibly be used if the goal was to make the computations as fast as possible. Author was able to implement methods for:

1. Finding and printing points of SW GW and BN curves with user-defined parameters,
2. Checking whether or not is the curve singular,
3. Point addition and doubling based on formulas described in Section (1.6),
4. Scalar multiplication based on the recursive double-and-add algorithm [5].

3.1.1 Double-and-add

Figure 3.1 depicts the implementation of the double-and-add algorithm, described in Section 1.6.5. The depicted method uses the **recursive** version of said algorithm. Depending on the value of scalar, method either decides the multiplication is done and returns resulting point, calls point addition and itself with the scalar decremented by 1 (if the scalar is **odd**) or calls only itself with halved scalar (if the scalar is **even**).



```java
public Point scalarMultiplication(int x1, int y1, int s) {

    if (s == 0)
        return null;

    Point P = new Point(x1, y1,  ec: this);

    if (s == 1)
        return P;
    if (s % 2 == 1) {
        return P.add(scalarMultiplication(P.getX(), P.getY(),  s: s - 1));
    } else
        P = P.add(P);
    return scalarMultiplication(P.getX(), P.getY(),  s: s / 2);
}
```

Fig. 3.1: Recursive double-and-add algorithm.

3.1.2 Summary

Author's implementation has proven to be fully functional only for the SW and BN curves, operations on GW curves behaved in an **inconsistent** manner for some specific inputs. Author was however able to deepen his understanding of the operations on elliptic curves. This approach was dropped in favour of the other two.

3.2 Implementation using library

In this section, the state-of-the-art ECC **libraries** are researched. Based on the results of this research, it is decided which one would serve the purpose of this thesis the best. Note that the main qualities in terms of this thesis are:

1. Support for curves of various forms with user-defined parameters.
2. Printing points of a curve.
3. Performing point addition/multiplication.

The results of conducted research are displayed in Table 3.1. As expected, every researched library is capable of performing basic points operations. Lack of this feature would make usefulness of such library questionable. However, none of the researched libraries is able to find and display the points of an elliptic curve. The purpose of these libraries is to function in **practical environments**, where the order of the curves must be big enough to ensure security of the encryption. Finding all points of a curve with safe parameters is a difficult problem, which however does not need to be solved in practical deployment.

Tab. 3.1: Library candidates. "User-defined" stands for curves whose parameters can be explicitly defined by user. "Pre-defined" stands for forms of built-in curves whose parameters are given by various standards. Ed stands for Edwards curves.

Library	Language	User–defined	Pre–defined
libTomCrypt[22]	C	SW	SW
Crypto++[23]	C++	SW	SW, Montgomery
Bouncy Castle[24]	Java	SW	SW, Montgomery
AMCL[25]	C	None	SW, Ed, Montgomery
TEPLA[26]	C	None	BN
libecc[27]	C	SW	SW
OpenSSL[9]	C	SW	SW, Montgomery
PARI/GP[10]	C	GW	GW
Relic[28]	C	None	SW, Ed, BN

These facts make the ability to process **user-defined** curves main deciding factor. Based on the information in Table 3.1, it is evident that majority of libraries are only capable of handling user-defined curves in SW form or do not support user-defined curves at all. *PARI/GP* is an exception, as it works with curves defined in GW form. This could prove useful, since all the other forms of elliptic curves can be **expressed** in GW form. It is also the library *SageMath* uses for its calculations on elliptic curves.

3.3 *SageMath*

It was decided to use *SageMath* as the back-end engine, because it allows the application to access functionalities of *PARI/GP* and is more suited for deployment on a web application written in Java, compared to working with C library directly. As a bonus, *SageMath* already has a function which can find and **display all points** on an elliptic curve.

There were two possible methods to use *SageMath* as the back-end engine of the web application:

- embedding *SageMathCell* to HTML code, described in Section 2.4.4,
- reimplementing the *SageMath* server connection, described in Section 2.4.5.

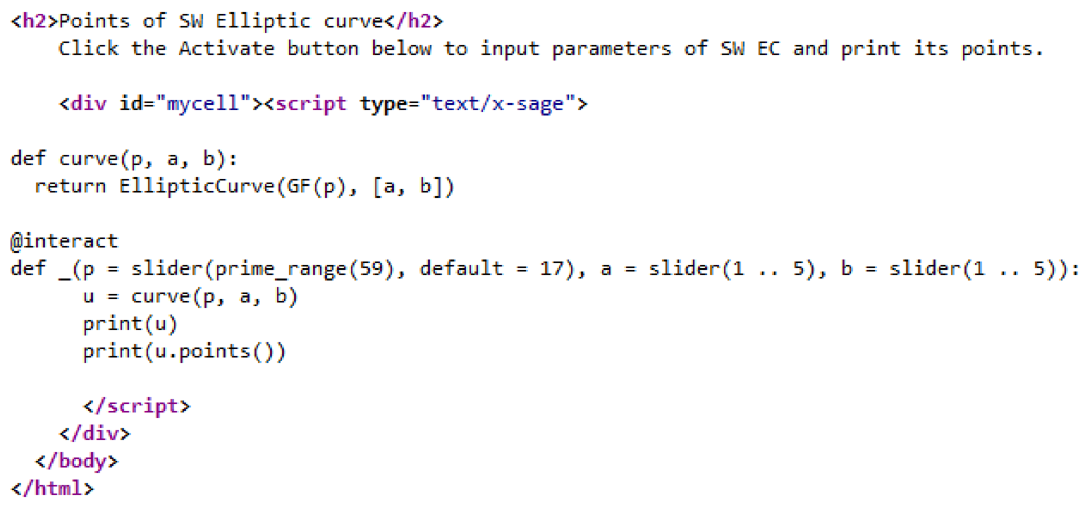
3.4 *SageMathCell*

In order to test the *SageMathCell*, it was embedded on a simple HTML page. The *SageMathCell* included a Python code, that would create an elliptic curve in SW form based on users parameters and print out its points. The parameters were inserted via sliders, which allowed the web page to restrict the user from inputting invalid parameters, such as non-prime field.

Figure 3.2 depicts a source code of the *Cell*. The figure does not show the declarations in HTML head, which are needed for the *SageMathCell* to work. The annotation *@interact* is used to specify which method is handling users inputs. The code is otherwise pretty self-explanatory: method *curve* creates SW elliptic curve based on its 3 parameters. Method with annotation *@interact* collects these 3 parameters as an input from the user. The elliptic curve is then stored in variable *u*. Variable *u* can then be used to access *SageMath*'s various functions for elliptic curves, such as *points()*, which prints the points of an elliptic curve.

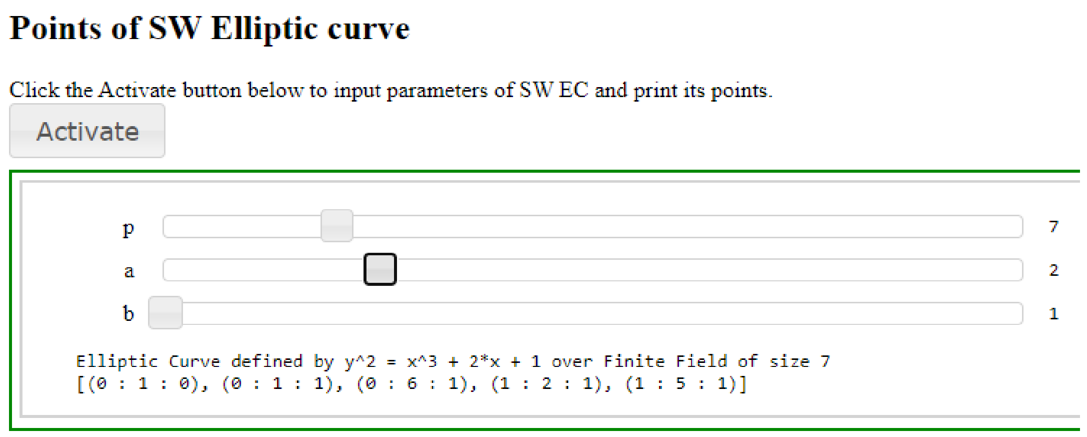
Figure 3.3 shows what does the *Cell* look like in a web browser. The results are automatically reevaluated every time the user changes his input via sliders.

There is however a major issue, which will now be briefly described. According to manual covering *SageMathCell*s included in *SageMath* GitHub repository it is possible to link more *SageMathCell*s together [15]. This should make them share the same *SageMath* Kernel, and therefore a variable declared in one *Cell* should be accessible in another linked *Cell*. This does not seem to be the case for variables handled inside the method with annotation *@interact*. For example, it was not possible to access variable u, which stored the elliptic curve defined by users parameters, in another *SageMathCell* which could serve for point addition on said curve. Because of this, *SageMathCell* can not be used as the back-end engine of the application that would function one step at time (choosing form of the curve $\rightarrow$ inputting the parameters of the curve $\rightarrow$ choosing an operation etc. The necessity to input everything at once could overwhelm less experienced users.



```
<h2>Points of SW Elliptic curve</h2>
    Click the Activate button below to input parameters of SW EC and print its points.

    <div id="mycell"><script type="text/x-sage">

def curve(p, a, b):
  return EllipticCurve(GF(p), [a, b])

@interact
def _(p = slider(prime_range(59), default = 17), a = slider(1 .. 5), b = slider(1 .. 5)):
      u = curve(p, a, b)
      print(u)
      print(u.points())

      </script>
    </div>
  </body>
</html>
```

Fig. 3.2: Source code of *SageMathCell* capable of printing points of elliptic curve.



Points of SW Elliptic curve

Click the Activate button below to input parameters of SW EC and print its points.

```
Elliptic Curve defined by y^2 = x^3 + 2*x + 1 over Finite Field of size 7
[(0 : 1 : 0), (0 : 1 : 1), (0 : 6 : 1), (1 : 2 : 1), (1 : 5 : 1)]
```

Fig. 3.3: *SageMathCell* capable of printing points of elliptic curve.

Author tried to fix this problem using **Spring Web** and **Thymeleaf**. In short, another HTML page with form in it was created, that would pass the user-given parameters to the original HTML with the embedded *SageMathCell*. *SageMathCell* would then create an elliptic curve based on those parameters and then the method with the annotation *@interact* could be used to pick points for addition instead. Unfortunately, *SageMathCell* seemed to not be compatible with Spring Web.

Therefore, it was decided to adopt the *SageMath* **server** option for implementing the mathematical back-end, which was explored and documented in [12].

3.5 *SageMath* server connection

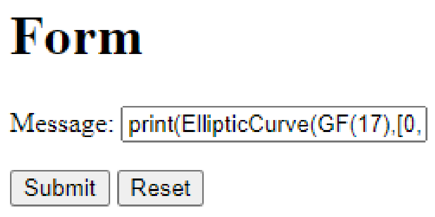
This section covers reimplementation of the *SageMath* server connection described in Section 2.4.5. Another student [12] used this method to connect his JavaFX desktop application with *SageMath* server running on *localhost*.

One of the main challenges of this thesis was to adopt this method and deploy it on a web application. Accomplishing this would eliminate the major downside of previous student's application–the necessity to first install *SageMath* and the desktop application itself on user's computer. The web application developed as the practical part of this thesis requires only connection to the Internet.

The web application also offers more functionalities, for example support of **various forms** of curves and ability to simulate ECDSA protocol.

Before the implementation of the main web application started, it was first needed to test if it is even possible to connect the *Spring* web application with the *SageMath* server in the same way as the desktop application. This was tested by creating a very simple *Spring* web application with single HTML form. This HTML form would take raw unformatted input, send it to the *SageMath* server for evaluation and show server's output. Note that the server has to be built first by running *sage-daemon.py* [16] from *SageMath* Shell. After examination of the predecessor's source code, it was discovered that he has implemented method called *interactWithSage*, which does exactly that − sends a string to the *SageMath* server via socket and returns *SageMath* server's output as another string. This method was adopted unchanged, because there seems to be no other way of implementing this method. HTML form on the test *Spring* web application was connected with the *interactWithSage* method, by adding the method to the Controller of the web application. The test web application that was created was working similarly to *CoCalc*, online *SageMath* notebook. Form for submitting *SageMath* code of this test web application is depicted on Figure 3.4.

The connection between *SageMath* server and *Spring* web application was working properly. Next task was creating the main web application and deploying the connection there. The main web application should allow users to submit their input in **user-friendly format**, the internal logic of the web application would then translate the input to *SageMath* code and send it to the *SageMath* server for evaluation.


Form

Message: print(EllipticCurve(GF(17),[0,

Fig. 3.4: Test *Spring* web application capable of evaluating *SageMath* input.

3.6 Implementation of the Web Application

This section covers the implementation of the final **web application**, describes its functionality and shows its appearance via numerous screenshots. The final application is user-friendly and can be used as an education tool, which is its main purpose. The application in its final state is capable of:

- handling three forms of user-defined elliptic curves (SW, GW and BN),
- finding points and displaying them to the user,
- finding orders of all points on the curve and displaying them to the user,
- calculating the order of the curve,
- performing addition of user-picked points,
- performing doubling of user-picked point,
- performing scalar multiplication of user-picked point,
- simulating ECDH key exchange protocol,
- generating ECDSA signature,
- verifying ECDSA signature.

3.6.1 Initialization of *Spring* project

The first step of implementation was generating a blank *Spring* project at `https://start.spring.io/`. The project was generated with these settings:

- Gradle as the build automation tool,
- *Spring* version 2.4.0,
- Java 11,
- *Spring Web* and *Thymeleaf* modules dependencies.

Spring Web and *Thymeleaf* are crucial for making the web application work as intended. These two modules were introduced in previous chapter. Other settings are based on personal preference, the implementation of the web application would be the same even if for example some other version of *Spring* was used.

3.6.2 General scheme of the application

The web application operates in **cycles** based on the **Model-View-Controller pattern**. Simplified diagram of such cycle is depicted in Figure 3.5. The general diagram will be explained in next few paragraphs, next two sections describe process of evaluating point addition and ECDSA, where the cycles will be explained in practise.

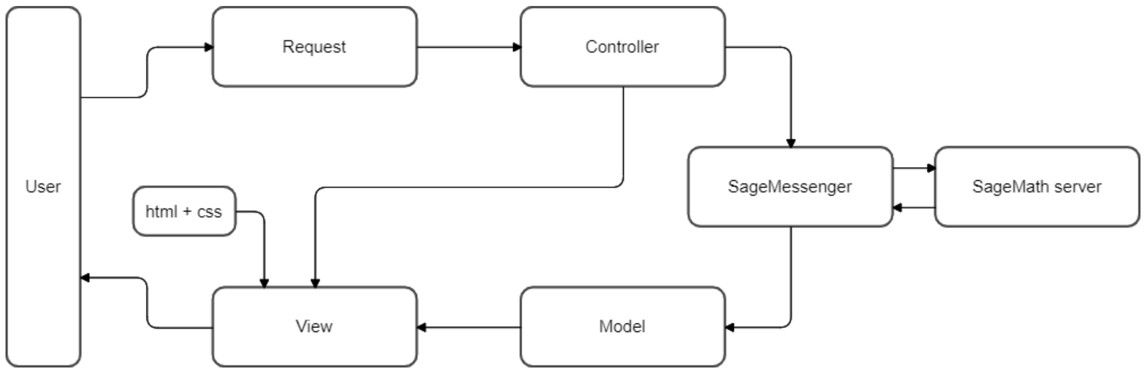


Fig. 3.5: Simplified diagram of the application.

The cycle starts with user making a **request**. This request carries information based on the user's previous actions, such as variables he put in via forms, and address of his next destination on the website.

Controller

Class annotated *@Controller* is the core of the *Spring* web application. It consists of **handler methods**, each mapped on different address. The Controller reads the address in the request and invokes the corresponding handler method. Handler method then sends instructions to the *SageMessenger* object, which executes all the computations, based on the variables stored in the request. *SageMessenger* object can directly make requests to the *SageMath* server, created by the script. After the computations, *SageMessenger* object (which also stores the results) is added to the Model, so the View can be rendered based on *SageMessenger*'s contents. Handler method then returns name of the HTML template, which will be used for rendering the next View.

SageMessenger class

Everything math-related is handled inside this class. It includes a method called *interactWithSage*, which was introduced in previous section. Purpose of this method is to send input to *SageMath* server and store its response to a String. Note that the *SageMath* server has to be launched first by running the script *sage-daemon.py* in *SageMath* Shell.

SageMessenger class also contains several variables (curve parameters, lists of points, scalar for multiplication, etc.), which are used for storing user input as well as storing results. The results returned by *SageMath* server are not in an ideal format, so they have to be processed by several formatting methods first. For example method *makeSagePointLookPrettier* transforms projective point representation of *SageMath* (which is "(x : y : z)") to affine representation of "[x, y]", which is more user-friendly.

Another part of the class are methods [29, 30, 31] for solving several mathematical problems, which do not require help from the *SageMath* server. For example method *findDivisors* is used for finding all divisors of a number. This is needed for sorting points of an elliptic curve to lists based on their order. Finding the divisors of the order of the curve is equal to finding all possible orders of the points.

Last part of the *SageMessenger* class are methods used for generating input for *interactWithSage* method. These methods take the user-defined parameters and translate them to the SageMath syntax. These methods handle the computations on elliptic curves, which can not be handled by Java alone.

HTML templates

Last part of the web application is a collection of HTML templates (one for each handler method in Controller), which are used for rendering Views. *Thymeleaf* engine enables these HTML templates to access the *SageMessenger* object stored in the Model and read the variables inside. *Thymeleaf* even allows embedding some basic algorithms into the HTML files, like cycling through a list or a hash map. This is for example used for rendering a table of points from a **hash map** called *pointsByOrder*, which is stored in the *SageMessenger* object. This hash map uses possible orders of the points as a key and list of points of that exact order as a value. This allows the website to render a table of points sorted by their orders.

The layout of the web page generally consists of 4 components:

1. navigation bar at the top, which allows user to either return to home page or start computations on a new curve,
2. information about the curve, which is currently being processed,
3. table of points of said curve,
4. the computations part, where user can either put in his input via forms or see the results.

The forms for user input are designed in a way that allows as little user error as possible. For example coordinates of points, which will be used for computations, are selected from a dropdown menu of viable points. This prevents the user from choosing a point that is not located on the currently processed curve.

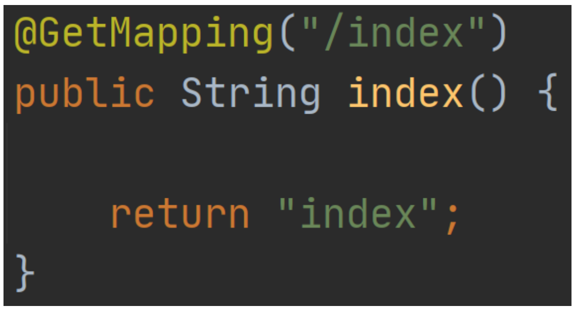
The HTML templates are styled by *Bootstrap*´s CSS style sheets. The overall appearance of the web application is very simple and minimalistic, which complements its educative purpose.

3.6.3 Point addition

This section covers all the processes that lead to evaluation of point addition. The purpose of this section is to explain functionality of the application to the reader on practical example, with screenshots of the application and samples of the source code. Note that the other operations function very similarly, the basic principles are the same across the application.

Homepage

Homepage of the application can be found on address */index*. After visiting this address, a very first request is made and sent to the Controller. Controller finds the corresponding handler method for address */index* and calls it. This handler method is depicted in Figure 3.6. The handler just returns name of the template, which is *index.html*, there is no further internal logic involved. This is due to the fact that homepage of the web application is always the same, there are no variables to be computed or displayed. Template *index.html* is then used to render a View of the homepage in user's web browser.

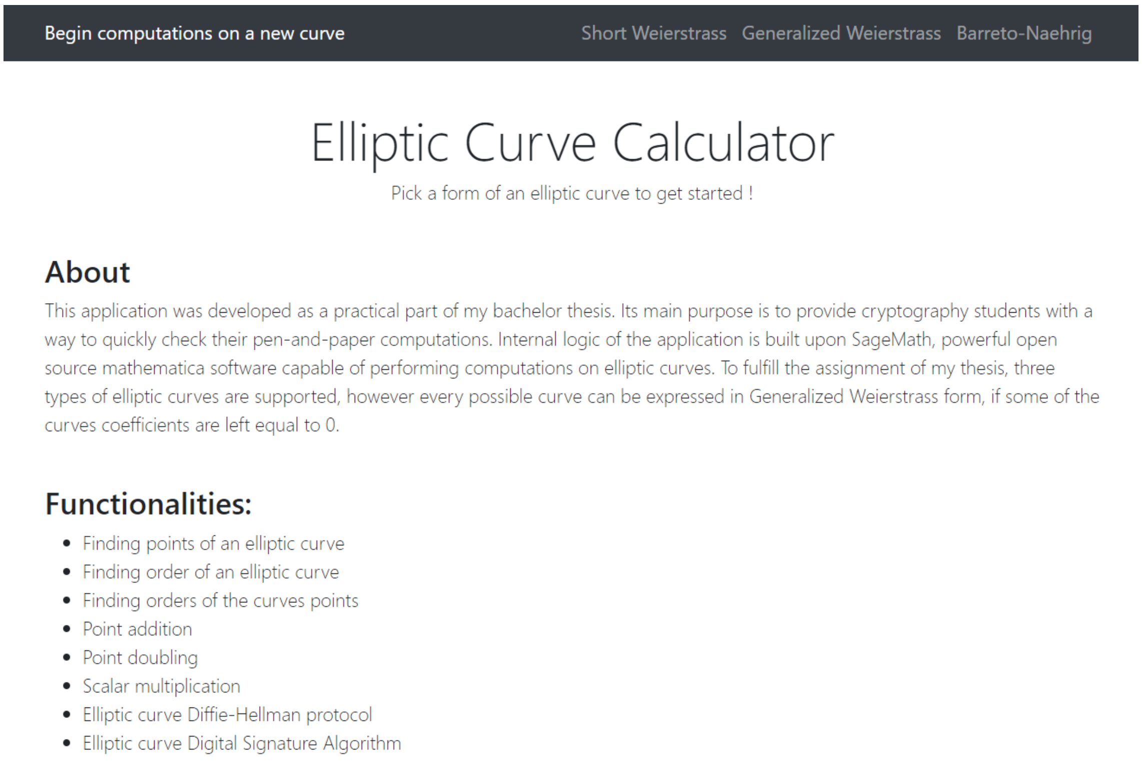


```
@GetMapping("/index")
public String index() {

    return "index";
}
```

Fig. 3.6: Handler method mapped on */index*.

The template *index.html* consists of navigation bar, which is also present on every other page of the application, big main header and short info about the application. Homepage of the application is depicted in Figure 3.7.

User can pick a form of the curve via the navigation bar at the top. The navigation bar can also be used later to reset all the computations and start a new computation on a new curve. Figure 3.8 depicts the source code of the navigation


Begin computations on a new curve

Short Weierstrass Generalized Weierstrass Barreto-Naehrig

Elliptic Curve Calculator

Pick a form of an elliptic curve to get started !

About

This application was developed as a practical part of my bachelor thesis. Its main purpose is to provide cryptography students with a way to quickly check their pen-and-paper computations. Internal logic of the application is built upon SageMath, powerful open source mathematica software capable of performing computations on elliptic curves. To fulfill the assignment of my thesis, three types of elliptic curves are supported, however every possible curve can be expressed in Generalized Weierstrass form, if some of the curves coefficients are left equal to 0.

Functionalities:

- Finding points of an elliptic curve
- Finding order of an elliptic curve
- Finding orders of the curves points
- Point addition
- Point doubling
- Scalar multiplication
- Elliptic curve Diffie-Hellman protocol
- Elliptic curve Digital Signature Algorithm

Fig. 3.7: View rendered from *index.html* in browser.

bar. As shown in the figure, the user can be redirected to 3 different addresses from the navigation bar.

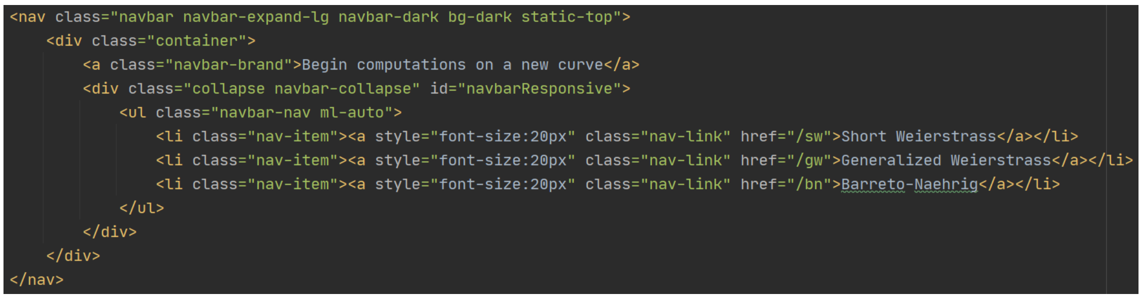

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
    <div class="container">
        <a class="navbar-brand">Begin computations on a new curve</a>
        <div class="collapse navbar-collapse" id="navbarResponsive">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item"><a style="font-size:20px" class="nav-link" href="/sw">Short Weierstrass</a></li>
                <li class="nav-item"><a style="font-size:20px" class="nav-link" href="/gw">Generalized Weierstrass</a></li>
                <li class="nav-item"><a style="font-size:20px" class="nav-link" href="/bn">Barreto-Naehrig</a></li>
            </ul>
        </div>
    </div>
</nav>
```

Fig. 3.8: Source code of the navigation bar, styled by *Bootstrap* CSS.

Choosing parameters of SW Curve

Let us assume the user has chosen Short Weierstrass form for his new computation. He is redirected to */sw*, another request is made and sent to the Controller and handled by the corresponding handler method. Finally, the template *swparam.html* is returned by the handler method and new View is rendered from it. The View rendered from *swparam.html* is depicted on Figure 3.9. As shown in the figure, purpose of this page is to collect the parameters of the curve user would like to perform computations on. In case the parameters provided by the user are not valid, error page is invoked and user is notified about this flaw.

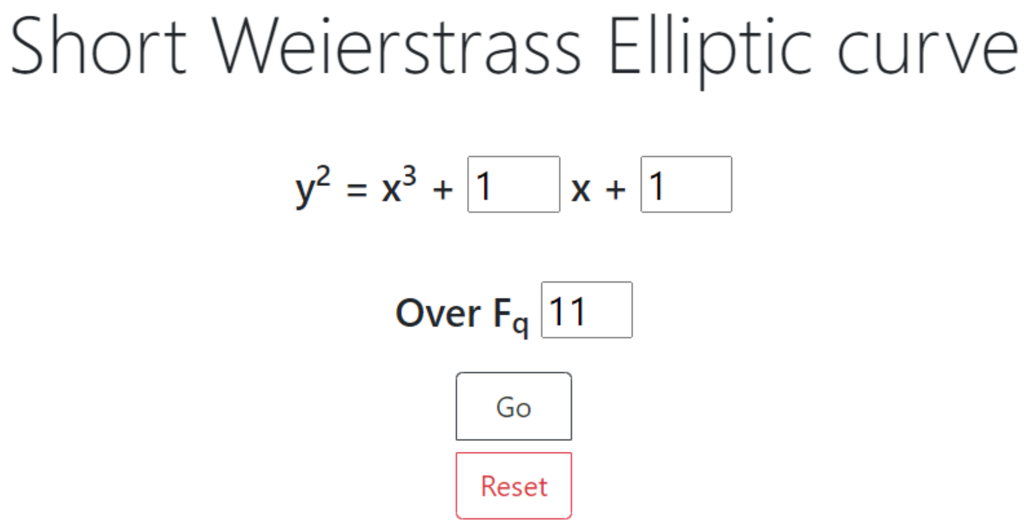


Fig. 3.9: View rendered from *swparam.html*. The navigation bar at the top is cropped off.

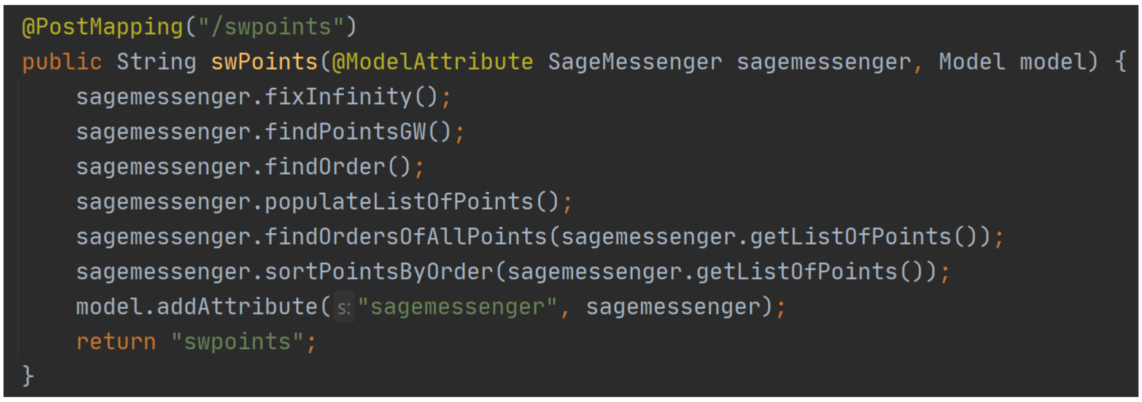
Finding points of the curve

After pressing "Go", user is redirected to */swpoints* and another request is made. This time the request carries the parameters, which the user typed in. Handler method mapped on */swpoints* is called. Source code of this method is depicted in Figure 3.10. The method initializes object of the *SageMessenger* class and calls various methods within the object:

1. method *fixInfinity* declares a variable O inside the *SageMath* server. This allows us to refer to point at infinity as O, instead of *SageMath*'s default $(0 : 1 : 0)$.
2. Method *findPointsGW* generates the input readable by *SageMath* based on the user-provided parameters of the curve and sends the request to *SageMath* server via *interactWithSage* method. The server's response (unformatted string of all the points of the curve in *SageMath*'s default representation) is then stored in the variable *points*. Source code of this method is depicted in Figure 3.11.
3. Method *findOrder* sends request to the *SageMath* server to find order of an elliptic curve and stores it in one of the *SageMessenger*'s variables.
4. Method *populateListOfPoints* creates new object of *Point* class for each point stored in String *points* (see method *findPointsGW*) and stores them in an

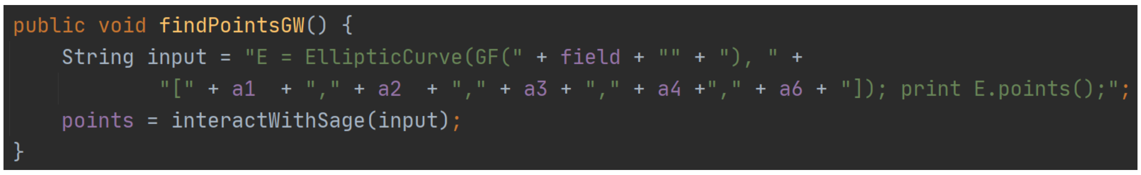
array list called *listOfPoints*. Class *Point* has three integer variables inside – x-coordinate, y-coordinate and the order of the point.

5. Method *findOrdersOfAllPoints* sends input to *SageMath* server to find order of each point stored in *listOfPoints*.
6. Method *sortPointsByOrder* first finds divisors of the curve's order and then fills hash map *pointsByOrder* with array lists and keys equal to the pre-computed divisors. Lists inside the hash map then get filled with *Point* objects from the original array list *listOfPoints*. This results in each list inside the hash map *pointsByOrder* storing points of one exact order. Method *sortPointsByOrder* is depicted in Figure 3.12.



```java
@PostMapping("/swpoints")
public String swPoints(@ModelAttribute SageMessenger sagemessenger, Model model) {
    sagemessenger.fixInfinity();
    sagemessenger.findPointsGW();
    sagemessenger.findOrder();
    sagemessenger.populateListOfPoints();
    sagemessenger.findOrdersOfAllPoints(sagemessenger.getListOfPoints());
    sagemessenger.sortPointsByOrder(sagemessenger.getListOfPoints());
    model.addAttribute( s: "sagemessenger", sagemessenger);
    return "swpoints";
}
```

Fig. 3.10: Source code of the handler method mapped on */swpoints*.



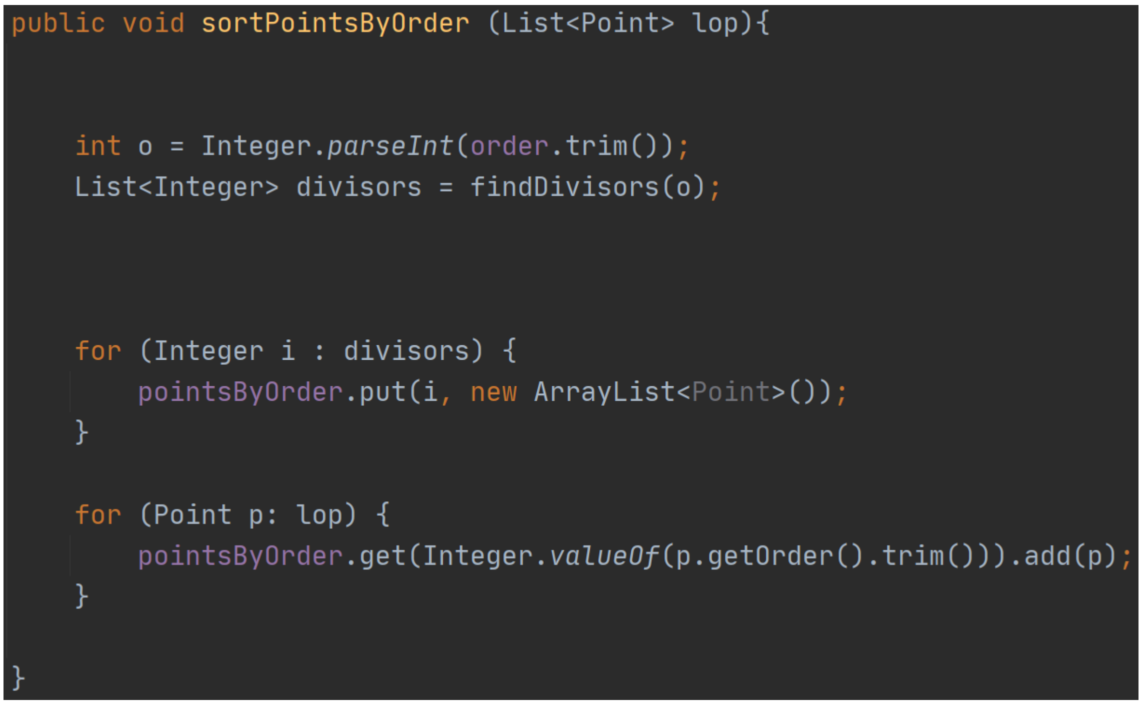
```java
public void findPointsGW() {
    String input = "E = EllipticCurve(GF(" + field + "" + "), " +
            "[" + a1  + "," + a2  + "," + a3 + "," + a4 +"," + a6 + "]); print E.points();";
    points = interactWithSage(input);
}
```

Fig. 3.11: Source code of the *SageMessenger*'s method used for finding points on a user-defined elliptic curve.

Finally, the handler method mapped on */swpoints* adds object *SageMessenger* to the Model of the application, so the variables inside it can be accessed from template *swpoints.html*, which is used for rendering the next Model.

Template *swpoints.html* as well as all the next templates consists of 5 parts:

1. the navigation bar at the top,
2. parameters of the user-defined elliptic curve,
3. order of the curve,
4. table of points sorted by their order (generated from the *pointsByOrder* hash map,



```
public void sortPointsByOrder (List<Point> lop){


    int o = Integer.parseInt(order.trim());
    List<Integer> divisors = findDivisors(o);



    for (Integer i : divisors) {
        pointsByOrder.put(i, new ArrayList<Point>());
    }

    for (Point p: lop) {
        pointsByOrder.get(Integer.valueOf(p.getOrder().trim())).add(p);
    }

}
```

Fig. 3.12: Source code of the *SageMessenger*'s method used for filling hash map *pointsByOrder* with lists of points based on their order.

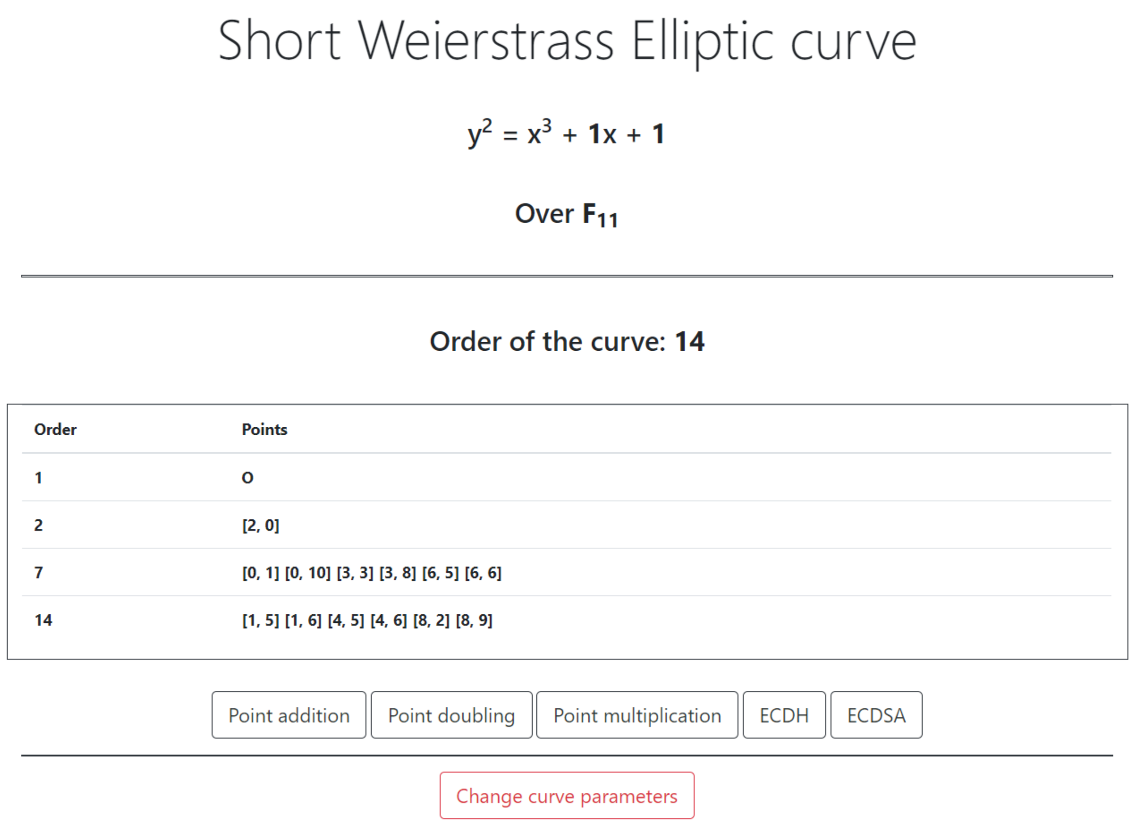
5. buttons used for choosing desired operation.

View rendered from the *swpoints.html* template is depicted in Figure 3.13. Source code of the table is depicted in Figure 3.14. Tags "th:" are processed by *Thymeleaf*.

Choosing points for addition

Let us assume the user wants to compute point addition on the curve he has submitted. Clicking the "Point addition" button redirects him to */swadd* and corresponding handler method, which returns template *swadd.html*, is called. The already known parameters $(a, b, field\dots)$ are stored in the request as hidden form parameters. However, this method is able to store only strings and integers in the request – hash map *pointsByOrder* can not be stored in the request. Instead, the points of the curve are stored in the request as the string variable *points* (unformatted *SageMath* output, see Figure 3.11). The hash map is then filled again from the contents of this variable. This is definitely not the most optimal solution, however, filling the hash map *pointsByOrder* is not too computationally demanding, compared to finding the coordinates of the points (which is done by the *SageMath* server) for example.

Template *swadd.html* consists of several parts, most of them are the same as in the previous template:


Short Weierstrass Elliptic curve

$y^2 = x^3 + 1x + 1$

Over F_{11}

Order of the curve: **14**

Order	Points
1	O
2	[2, 0]
7	[0, 1] [0, 10] [3, 3] [3, 8] [6, 5] [6, 6]
14	[1, 5] [1, 6] [4, 5] [4, 6] [8, 2] [8, 9]

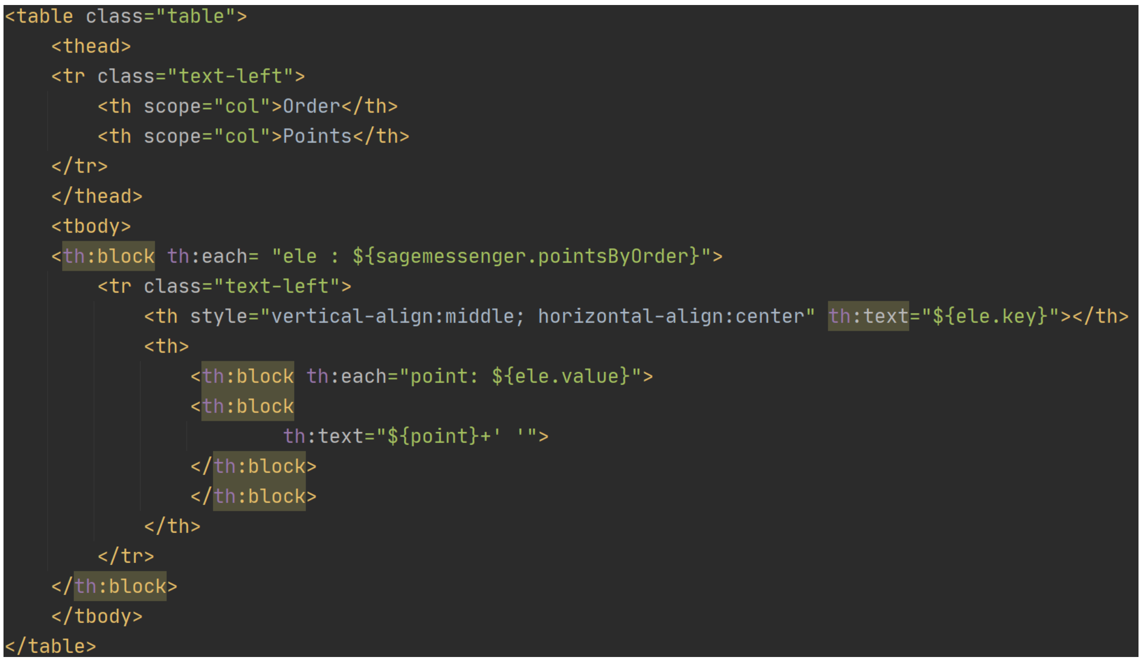
Fig. 3.13: View rendered from template *swpoints.html*. The navigation bar is cropped.

1. the navigation bar at the top,
2. parameters of the user-defined elliptic curve,
3. order of the curve,
4. table of points sorted by their order (generated from the hash map),
5. two dropdown menus, where the user can select the desired points for addition.

Figure 3.15 depicts the dropdown menus, which are used for selecting points for points addition. Note that the menus are situated below the table of points.

Displaying results of the addition

Clicking "Go" redirects the user to */swaddres* (short Weierstrass addition result). User-picked points are stored in the request. Object *SageMessenger* then creates an input for the *SageMath* server based on the coordinates of the points user has picked. *SageMath* server then returns result of the addition and it is stored inside one of the many variables of the *SageMessenger* object. Template *swaddres.html* is used to render the View with the result of the addition. This View is depicted on Figure 3.16. User can choose his next destination on the application via the provided buttons.



```
<table class="table">
    <thead>
    <tr class="text-left">
        <th scope="col">Order</th>
        <th scope="col">Points</th>
    </tr>
    </thead>
    <tbody>
    <th:block th:each= "ele : ${sagemessenger.pointsByOrder}">
        <tr class="text-left">
            <th style="vertical-align:middle; horizontal-align:center" th:text="${ele.key}"></th>
            <th>
                <th:block th:each="point: ${ele.value}">
                <th:block
                        th:text="${point}+' '">
                </th:block>
                </th:block>
            </th>
        </tr>
    </th:block>
    </tbody>
</table>
```

Fig. 3.14: Source code of the table. *Thymeleaf* engine is used to cycle through the *pointsByOrder* hash map.

3.6.4 ECDSA

This section showcases the way the web application computes ECDSA protocol. After successfully loading a curve and choosing the ECDSA via the corresponding button (see Figure 3.13), the user is redirected to */swecdsa*. The bottom part of the View he sees is depicted in Figure 3.17. As shown in the figure, user can choose whether he wants to generate a signature or verify a signature. Either way, he has to choose a generator G of prime order n and put in the integer e, which represents the hash of the message. The dropdown menu, from which the user chooses the generator, is filled only with points of prime order. User is asked to input the hash directly, because the purpose of the application is to allow students to check their pen-and-paper computations. The possible assignment of student's exercise on ECDSA would certainly provide the message in this format, because computing hash only on paper is very challenging task.

Let us suppose the user has chosen signature generation. He is redirected to */swecdsasign*, where he is asked to input 2 additional parameters, which are specific only to the generation of signature. These parameters are his private key d and random integer k. The prime order n of the generator G is displayed to the user, because both private key d and random integer k have to be in $[1, n-1]$. The bottom part of the View rendered from template *swecdsasign.html* is depicted in Figure 3.18.

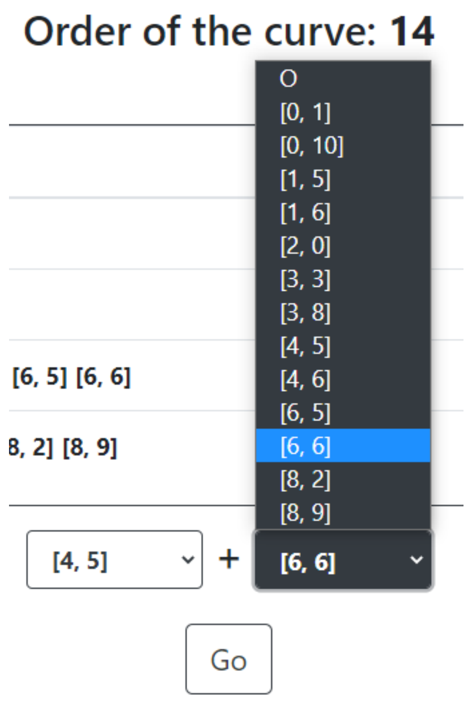


Fig. 3.15: Dropdown menus of the View rendered from *swadd.html*, used for picking points which are supposed to be added.

After clicking "Go", user is redirected to */swecdsasignres*, where the results of the computations described in Section 1.9.2 are displayed. These computations are depicted in Figure 3.19. In case the user inputs invalid d or k or the generated signature is not valid (r or s is equal to 0), the green success bar turns red and warns the user about these flaws.

Signature verification operates in a very similar manner, the application can distinguish between three results:

1. signature, which the user put in, is valid,
2. public key of the signer is not valid,
3. public key of the signer is valid, but the signature is not.

All three of these results alter the success/warning bar accordingly, so the user can know exactly where the issue is. Example of such warning bar is depicted in Figure 3.20.

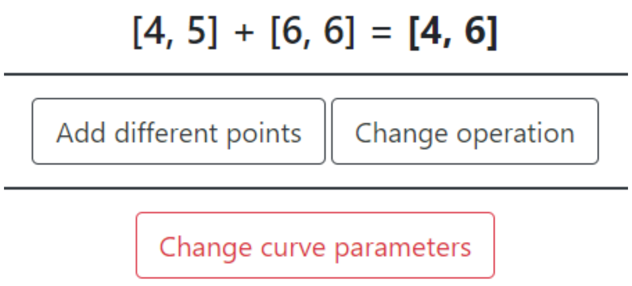
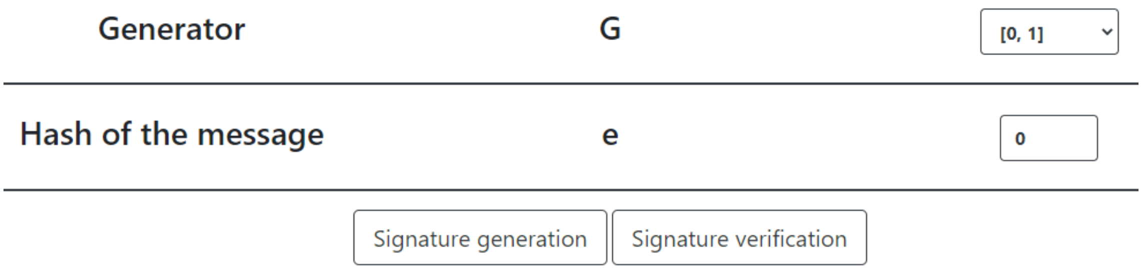
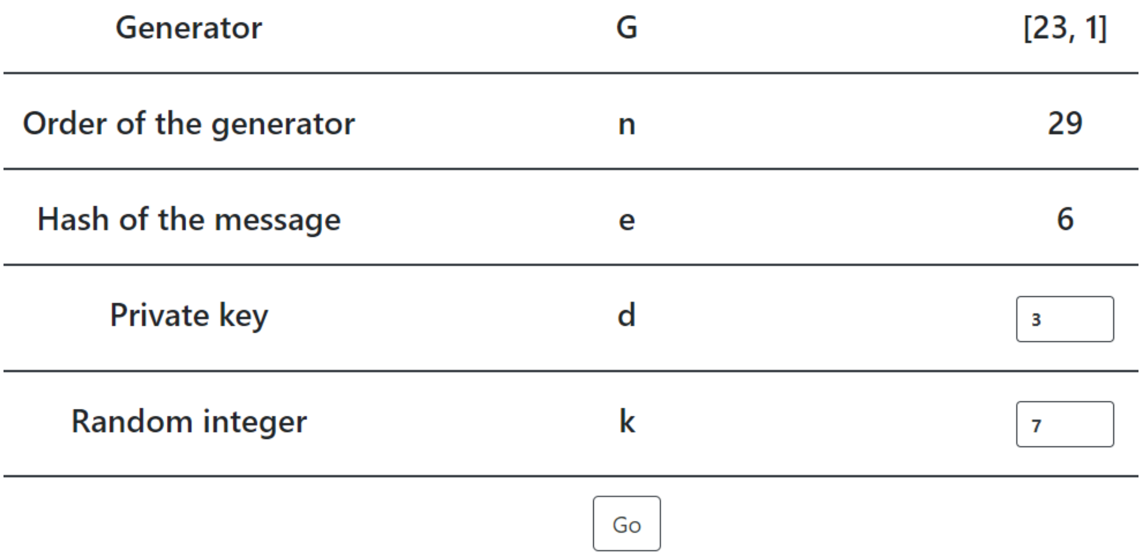

[4, 5] + [6, 6] = **[4, 6]**

Fig. 3.16: Result of the point addition.



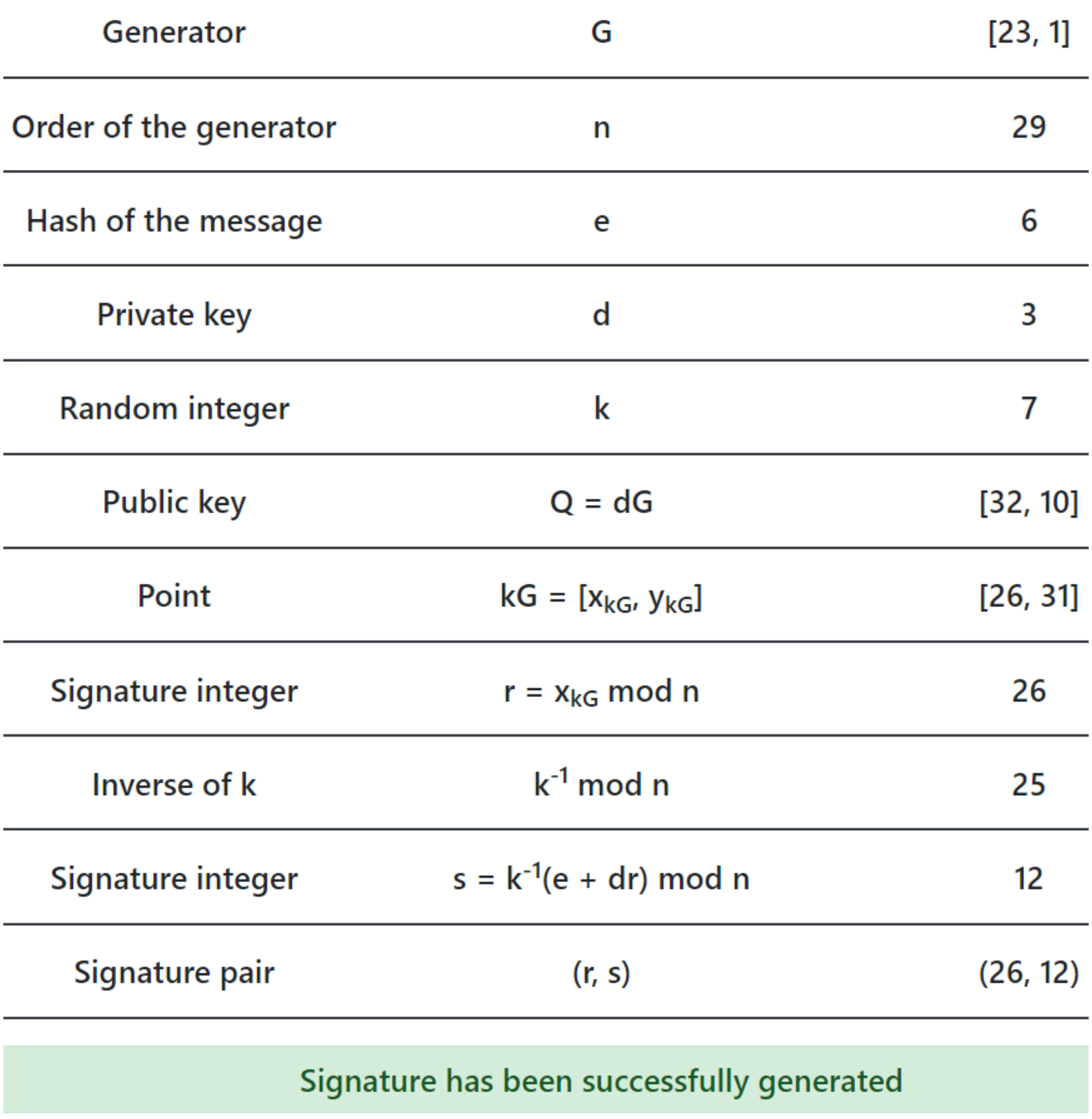
Generator	G	[0, 1]
Hash of the message	e	0

Fig. 3.17: View rendered from *swecdsa.html*.



Generator	G	[23, 1]
Order of the generator	n	29
Hash of the message	e	6
Private key	d	3
Random integer	k	7

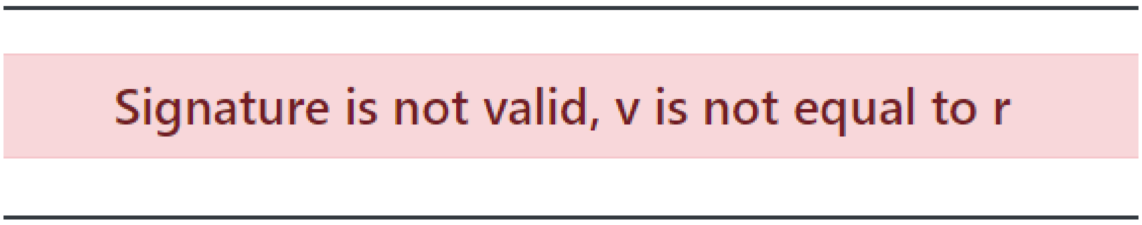
Fig. 3.18: View rendered from *swecdsasign.html*.



Generator	G	[23, 1]
Order of the generator	n	29
Hash of the message	e	6
Private key	d	3
Random integer	k	7
Public key	Q = dG	[32, 10]
Point	kG = [x_{kG}, y_{kG}]	[26, 31]
Signature integer	r = x_{kG} mod n	26
Inverse of k	k^{-1} mod n	25
Signature integer	s = k^{-1}(e + dr) mod n	12
Signature pair	(r, s)	(26, 12)

Signature has been successfully generated

Fig. 3.19: Computations of signature pair on View rendered from *swecdsasign-res.html*.



Signature is not valid, v is not equal to r

Fig. 3.20: Warning bar on View rendered from *swecdsacheckres.html*, caused by inputting valid signer's public key, but invalid signature.

Conclusion

The main contribution of this thesis is building web application, which provides students with user-friendly interface to powerful mathematical software, which is by design difficult to work with.

Three possible approaches for designing the mathematical back-end engine of the web application were proposed. Using *SageMath* has proven to be best option, because it enables using *PARI/GP* library, which is best suited for the cause. *SageMath* is also much more convenient to work with in the web application environment, compared to using a library written in C directly.

All the possibilities *SageMath* provides were carefully explored. It was decided that the best way to use *SageMath* is to build a server, which would process the requests sent by the web application.

The application that was built using *Spring* framework is capable of performing point addition, doubling and multiplication on GW, SW and BN elliptic curves of custom parameters. Two very popular and used protocols are also implemented. Thus, the requirements of the assignment seem to be fulfilled.

The application's appearance is minimalistic and clean, thanks to CSS styles of the *Bootstrap* framework. The web application is very easy to navigate, only basic understanding of the ECC is required. The mechanisms, which restrict the user from inputting invalid parameters, enhance the user-friendliness of the application even further. The application appears to run smoothly with little to no bugs. However, the code of the application could be more clean and better optimized, if the author was more experienced in implementing web applications. Another downside of the application is the lack of some sort of Wiki, which would explain the very basics of the ECC. However, the user can always resort to the first chapter of this thesis.

The application is currently not deployed on a server connected to the Internet, but it can be launched from a provided virtual machine. Manual on how to use the virtual machine is enclosed in the appendices of this thesis.

Bibliography

[1] STALLINGS, William. *Cryptography and network security principles and practices, 4th edition.* ISBN: 978-0-13-187319-3. Prentice Hall, 2005.

[2] MENEZES, Alfred J.; VAN OORSCHOT, Paul C.; VANSTONE, Scott A. *Handbook of applied cryptography.* ISBN: 978-0-84-938523-0. CRC press, 2018.

[3] COHEN, Henri, et al. *Handbook of elliptic and hyperelliptic curve cryptography.* ISBN: 978-1-58488-518-4. CRC press, 2005.

[4] SILVERMAN, Joseph H. An Introduction to the Theory of Elliptic Curves. In: *Computational Number Theory and Applications to Cryptography* [online]. 2006. Available at `https://www.math.brown.edu/johsilve/Presentations/WyomingEllipticCurve.pdf`

[5] PANTŮČEK. Dominik. Elliptic curves: double and add. In: *trustica.cz* [online]. May 3, 2018. [cit. 2020-10-26]. Available at `https://trustica.cz/en/2018/05/03/elliptic-curves-double-and-add/`.

[6] FAN, Junfeng, et al. State-of-the-art of secure ECC implementations: a survey on known side-channel attacks and countermeasures. In: *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST).* IEEE, 2010. p. 76-87.

[7] HANKERSON, Darrel.; MENEZES, Alfred.; VANSTONE, Scott. *Guide to Elliptic Curve Cryptography.* ISBN 0-387-95273-X. Springer–Verlag New York, Inc., 2004.

[8] JOHNSON, Don.; MENEZES, Alfred.; VANSTONE, Scott. The elliptic curve digital signature algorithm (ECDSA). In: *International journal of information security.* January 1, 2001. p. 36-63.

[9] Elliptic Curve Cryptography. In *OpenSSL Wiki.* Available at `https://wiki.openssl.org/index.php/Elliptic_Curve_Cryptography`.

[10] Elliptic curves. In *Catalogue of Functions for the GP/PARI CALCULATOR Version 2.12.1.* Available at `https://pari.math.u-bordeaux.fr/dochtml/html/Elliptic_curves.html`.

[11] *SageMath Documentation.* The Sage Development Team, 2020 [cit. 2020-11-10]. Available at `https://doc.sagemath.org/`.

[12] JANOUT, Vladimír. *Application for Elliptic Curve Cryptography.* Brno, 2020. Bachelor's Thesis. Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Advised by M.Sc. Sara Ricci, Ph.D.

[13] *Jupyter Notebook Documentation.* Jupyter Team, 2015 [cit. 2020-11-16]. Available at `https://jupyter-notebook.readthedocs.io/en/latest/notebook.html`.

[14] GROUT, Jason.; HANSON, Ira.; JOHNSON, Steven.; KRAMER, Alex.; NOVOSELTSEV, Andrey.; STEIN, William. *About SageMathCell.* Available at `https://sagecell.sagemath.org/static/about.html`.

[15] *Sage Mathematical Software System*, GitHub Repository. The Sage Development Team, 2020 [cit. 2020-11-17]. Available at `https://github.com/sagemath`.

[16] WOODGNOME. Running Sage from other languages with high(er) performance?. In: *ask.sagemath.org* [online]. July 21, 2014. [cit. 2020-11-18]. Available at `https://ask.sagemath.org/question/23431/running-sage-from-other-languages-with-higher-performance/`

[17] *Spring tutorial.* Baeldung, 2020 [cit. 2020-11-18]. Available at `https://www.baeldung.com/spring-tutorial`.

[18] *Spring framework tutorial.* Java2Blog, 2020 [cit. 2020-11-18]. Available at `https://java2blog.com/introduction-to-spring-framework/`.

[19] *Spring tutorial.* Tutorialspoint, 2020 [cit. 2020-11-18]. Available at `https://www.tutorialspoint.com/spring/index.htm`.

[20] *Thymeleaf documentation.* The Thymeleaf Team, 2020 [cit. 2020-12-01]. Available at `https://www.thymeleaf.org/documentation.html`.

[21] *Bootstrap documentation.* The Bootstrap team, 2021 [cit. 2020-05-24]. Available at `https://getbootstrap.com/docs/5.0/getting-started/introduction/`.

[22] ST DENNIS, Tom. *LibTomCrypt Developer Manual.* LibTom Projects. Available at `https://www.co.tt/files/libTomCryptDoc.pdf`.

[23] Elliptic Curve Cryptography. In *Crypto++ Wiki.* Available at `https://www.cryptopp.com/wiki/Elliptic_Curve_Cryptography`.

[24] *Bouncy Castle 1.66 API Specification.* Available at `https://javadoc.io/doc/org.bouncycastle/bcprov-jdk15on/latest/overview-summary.html`.

[25] *Apache Milagro Crypto Library Overview.* Available at `https://milagro.apache.org/docs/amcl-overview.html`.

[26] *TEPLA Manual.* Laboratory of Cryptography and Information Security, University of Tsukuba, 2013. Available at `http://www.cipher.risk.tsukuba.ac.jp/tepla/doc/tepladoc1_0_0e.pdf`.

[27] BENADJILA, Ryad.; EBALARD, Arnaud.; FLORI, Jean-Pierre. *Libecc Project Readme.* Available at `https://github.com/ANSSI-FR/libecc`.

[28] ARANHA, D. F.; GOUVÊA, C. P. L.; MARKMANN T.; WAHBY, R. S.; LIAO, K. *RELIC Readme.* Available at `https://github.com/relic-toolkit/relic`.

[29] ANKUR. Modular multiplicative inverse. In: *geeksforgeeks.org* [online]. April 21, 2021. [cit. 2021-05-20]. Available at `https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/`.

[30] Check If a Number Is Prime in Java. In: *baeldung.com* [online]. January 15, 2020. [cit. 2021-05-19]. Available at `https://www.baeldung.com/java-prime-numbers`.

[31] Find Factors of a Number and Save to an Array in Java. In: *stackoverflow.com* [online]. November 16, 2015. [cit. 2021-05-19]. Available at `https://stackoverflow.com/questions/33744408/find-factors-of-a-number-and-save-to-an-array-in-java`.

List of symbols, quantities and abbreviations

ECC	Elliptic–Curve Cryptography
GW	Generalized Weierstrass Curve
SW	Short Weierstrass Curve
BN	Barreto-Naehrig Curve
ECDH	Elliptic Curve Diffie-Hellman
TLS	Transport Layer Security
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheets
PARI/GP	Pascal ARIthmetic/Great Programmable calculator
GMP	GNU Multiple Precision arithmetic library
GAP	Groups, Algorithms, Programming
NTL	Number Theory Library
IDE	Integrated Development Environment
MIT	Massachusetts Institute of Technology
Ed	Edwards Curve

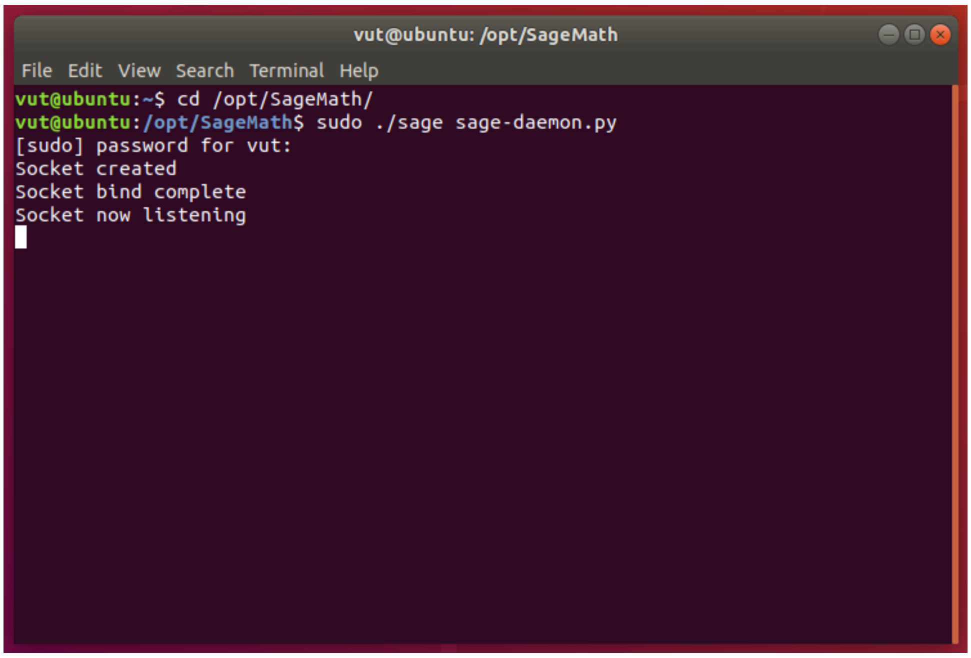
A File contents

```
xstark04BT
├── ECC_Web_App_test ..................... source code of the web application
├── sage-daemon.py ............................... launches SageMath server
└── manual.pdf ............... manual on how to run the application from VM
```

B Manual – Running the Application from provided VM

Running the application is not trivial, because a lot of prerequisites need to be present (*SageMath*, Java, etc.). Ubuntu virtual machine with all the prerequisites ready can be downloaded there:
`https://drive.google.com/file/d/1k6zKwsdY3JFvcJhwz0leoroghGrD5Fl0/view?usp=sharing`. In order to run the app and test it yourself, please:

1. download the VM,
2. import the VM to VMware Workstation Player,
3. run the VM,
   - user name: vut
   - password: vut
4. launch the *SageMath* server:
   - open Terminal (Ctrl + Alt + T),
   - run command *cd /opt/SageMath/* ,
   - run command *sudo ./sage sage-daemon.py* (password: vut),
   - successful launch of the *SageMath* server is depicted on Figure B.1.
   - Do **not** close the Terminal!
5. Launch the Tomcat server with the application on *localhost:8080*:
   - open IntelliJ Idea as in Figure B.2,
   - click on the green play button in top-right corner, as in Figure B.3,
   - starting the server takes a few seconds, see console at the bottom for info.
   - Do **not** close the IntelliJ Idea!
6. Open Google Chrome (Firefox is also an option, but the arrows for number inputs are disabled there),
   - the same way IntelliJ Idea was opened.
7. Visit *http://localhost:8080* .
   - Navigating through the application is simple, but it is recommended to read the Chapter 1 of this thesis (especially for ECDSA).


vut@ubuntu: /opt/SageMath
File Edit View Search Terminal Help

```
vut@ubuntu:~$ cd /opt/SageMath/
vut@ubuntu:/opt/SageMath$ sudo ./sage sage-daemon.py
[sudo] password for vut:
Socket created
Socket bind complete
Socket now listening
```

Fig. B.1: *SageMath* server is ready.

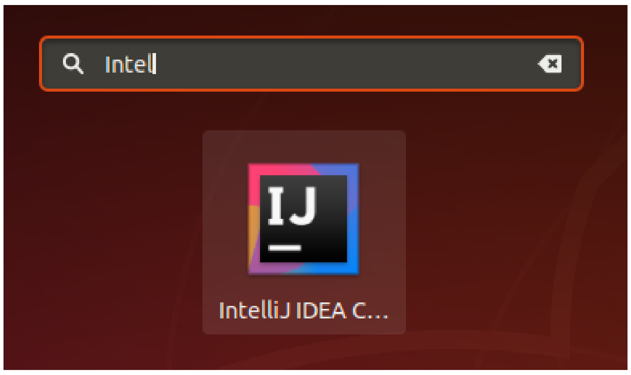


Fig. B.2: Launching *IntelliJ Idea*.

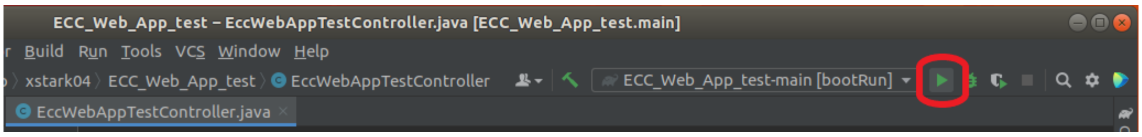

ECC_Web_App_test – EccWebAppTestController.java [ECC_Web_App_test.main]
r Build Run Tools VCS Window Help
xstark04 ECC_Web_App_test EccWebAppTestController
ECC_Web_App_test-main [bootRun]
EccWebAppTestController.java

Fig. B.3: Launching the application.