ECDSA Plug-in for the E-learning Platform JCrypTool

Bachelor-Thesis
Veselin Marinov
March 2010

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Mathematik
Fachbereich Informatik
Cryptography and Computer Algebra
Prof. Dr. Johannes Buchmann

ECDSA Plug-in for the E-learning Platform JCrypTool

Vorgelegte Bachelor-Thesis von Veselin Marinov

1. Gutachten: Prof. Dr. Johannes Buchmann
2. Gutachten: Dr. Evangelos Karatsiolis

Tag der Einreichung:

Declaration on the bachelor’s thesis

I hereby certify to have written this bachelor thesis without using help of a third party. For the creation of this bachelor thesis I have only used the literature listed in the bibliography at its end. The places, where material from the bibliography has been used, have been clearly identified and marked as such. This bachelor thesis is not to be found in the same or similar form and is not already existent at any exam authority.

Darmstadt, 14th March 2010

(V. Marinov)

Contents

1 Abstract **1**

2 Mathematical Preliminaries **2**
- 2.1 SHA-1 Hash Function . . . 2
- 2.2 Finite Fields . . . 2
  - 2.2.1 The finite field $\mathbb{F}_p$ . . . 2
  - 2.2.2 The finite field $\mathbb{F}_{2^m}$ . . . 3
- 2.3 Elliptic Curves over Finite Fields . . . 3
  - 2.3.1 Elliptic curves over $\mathbb{F}_p$ . . . 3
  - 2.3.2 Elliptic curves over $\mathbb{F}_{2^m}$ . . . 3

3 Ecliptic Curve Digital Signature Algorithm - ECDSA **4**
- 3.1 Description of the algorithm . . . 4
  - 3.1.1 ECDSA Domain Parameters . . . 4
  - 3.1.2 ECDSA Key Pairs . . . 4
  - 3.1.3 ECDSA Signature Validation and Verification . . . 4

4 Architecture **6**
- 4.1 JCrypTool . . . 6
- 4.2 FlexiProvider . . . 6
- 4.3 Eclipse Plug-in Architecture . . . 6
- 4.4 Rich Client Platform . . . 7
- 4.5 RCP Product . . . 7
- 4.6 Extension Points and Extensions . . . 7
- 4.7 JCrypTool Architecture . . . 8

5 Design and Implementation **9**
- 5.1 Requested Functionality . . . 9
- 5.2 Intention . . . 9
- 5.3 Design and User Interface . . . 9
- 5.4 Implementation . . . 11
  - 5.4.1 Internationalization . . . 11
  - 5.4.2 Class Hierarchy . . . 11
  - 5.4.3 Technical Implementation . . . 12
- 5.5 Workflow . . . 12
  - 5.5.1 ECDSA Domain parameters . . . 12
  - 5.5.2 ECDSA Signature Generation . . . 13
  - 5.5.3 ECDSA Signature Verification . . . 13
- 5.6 Example of Valid Signatures . . . 13
  - 5.6.1 Example of Valid Signature for $\mathbb{F}_p$ . . . 14
  - 5.6.2 Example of Valid Signature for $\mathbb{F}_{2^m}$ . . . 16

6 Conclusion **17**
- 6.1 Summary . . . 17
- 6.2 Outlook . . . 17

1 Abstract

In this bachelor thesis we give a brief description of the ECDSA algorithm and the mathematical preliminaries that are needed for the description of the ECDSA algorithm (as defined in [1]). We also describe the JCrypTool and give a practical implementation of an E-Learning ECDSA plug-in for JCrypTool.

Furthermore we give brief motivation why ECDSA is very important for secure electronic communications and keeping our privacy in Internet. Business practice has changed with introduction of computers and internet related technologies. Many organizations and firms are substituting their paper-based transactions with electronic ones. Great amounts of money are transferred over electronic communication mechanisms. The higher values of above mentioned transactions comes with severe risk from accidental or deliberate malicious alternation or destruction of the data transferred. Most of the conventional controls like safety paper, handwritten signatures or embossed seals are not available in such environment. In the generally conservative world we have to provide at least the same degree of certainty of authenticity as in the traditional paper environment. In an electronic environment there is need to have adequate replacements of the conventional security methods.

These are some of the reasons why the need of strong security has led to one of the strongest standards for digital signatures: The Elliptic Curve Digital Signature Algorithm (ECDSA). The algorithm defines ways for uniquely generating and validating digital signatures. ECDSA is the elliptic curve analog of the Digital Signature Algorithm (DSA). ECDSA was first proposed in 1992 by Vanstone in response to NISTs (National Institute of Standards and Technology) (see [1]).

ECDSA is used to create digital signatures (mainly in conjunction with a hash algorithm). Proper implementations of the algorithm provide data integrity, non-repudiation of the data, and data origin authentication.

Due to the high level security which ECDSA provides it is used in many E-banking and government facilities. For example of the three allowed algorithms RSA, DSA and ECDSA for the German electronic passports, the ECDSA was chosen.

The wide use of ECDSA drives the need for better understanding of the inner mechanics for generating secure transactions. The E-learning platform JCrypTool has been developed to serve this purpose (for more details see [13]). JCrypTool is an open-source, user-friendly, platform independent tool providing the possibility to extend the already available functionality with a plug-in. It is based on easy, scalable, and flexible technologies like Java, Eclipse RCP, and FlexiProvider.

This bachelor thesis is furthermore describing the implementation of an E-learning ECDSA plug-in for the JCrypTool. This plug-in gives the user understanding of how Elliptic Curve Digital Signature Generation and Validation work, allowing him to interact with each step of the algorithm and have maximum flexibility.

2 Mathematical Preliminaries

In this chapter we present basic theory that is applied to the practical implementation described in Chapter 5. We begin with a discussion on hash function.

2.1 SHA-1 Hash Function

A hash function compute a fixed-length digital representation (known as a message digest) of an input data sequence (the message) of any length. SHA-1 is defined in Secure Hash Signature Standard (see [5]).

Definition 2.1.1 *A (cryptographic) hash function H is a function that maps bit strings of arbitrary lengths to bit strings of a fixed length t such that*

1. *H can be computed efficiently;*
2. *(Preimage resistance) For essentially all $y \in \{0,1\}^t$ it is computationally infeasible to find a bit string x such that $H(x) = y$ and:*
3. *(Collision resistance) It is computationally infeasible to find distinct bit strings x_1 and x_2 such that $H(x_1) = H(x_2)$.*

2.2 Finite Fields

Next we provide a brief introduction to finite fields.

Definition 2.2.1 *A finite field is a set of elements $\mathbb{F}$ together with two binary operations on $\mathbb{F}$, called addition and multiplication that satisfy certain arithmetic properties, such as addition, multiplication, and inversion.*

Definition 2.2.2 *The number of elements of a field $\mathbb{F}$ is called order of the field $\mathbb{F}$. We note that there exists a finite field of order q if and only if q is a prime power. If q is a prime power there is only one field of order q and we will denote it by $\mathbb{F}_q$.*

There are many ways to represent the elements of $\mathbb{F}_q$. Some of them may lead to a more sufficient implementations of field arithmetic in hardware and in software.

Definition 2.2.3 *Let $q = p^m$ where p is a prime number and m is a positive integer. The number p is called the characteristic of the finite field $\mathbb{F}_q$ and the number m is called the extension degree of the field $\mathbb{F}_q$.*

We note that most standards that specify the elliptic curve cryptographic techniques restrict the order of the underlying finite field to be an odd prime ($q = p$) or a power of 2 ($q = 2^m$). In the next subsections we briefly describe the finite fields $\mathbb{F}_q$ and $\mathbb{F}_{2^m}$.

2.2.1 The finite field $\mathbb{F}_p$

Definition 2.2.4 *Let p be a prime number. The finite field $\mathbb{F}_p$, called a prime field, if it is comprised of the set of integers $\{0, 1, 2, \ldots, p-1\}$ with the following arithmetic operations:*

1. *(ADDITION): If $a, b \in \mathbb{F}_p$, then $a + b = r \in \mathbb{F}_p$ where r is the* ***remainder*** *when $a + b$ is divided by p and $0 \leq r \leq p-1$. This is called* ***addition modulo*** *p.*
2. *(MULTIPLICATION): If $a, b \in \mathbb{F}_p$, then $a \cdot b = s \in \mathbb{F}_p$ where s is the* ***remainder*** *when $a \cdot b$ is divided by p and $0 \leq s \leq p-1$. This is called* ***multiplication modulo*** *p.*
3. *(INVERSION): If $a \neq 0$ and an element of $\mathbb{F}_p$, the* ***inverse*** *of a modulo p denoted by a^{-1}, is the unique integer $c \in \mathbb{F}_p$ for which $a \cdot c = 1$.*

2.2.2 The finite field $\mathbb{F}_{2^m}$

Definition 2.2.5 *The finite field $\mathbb{F}$ is called a* ***characteristic two finite field*** *or a* ***binary finite field****, and is denoted $\mathbb{F}_{2^m}$, if it is a vector space of dimension m over the finite field $\mathbb{F}_2$. The finite field $\mathbb{F}_2$ consists of the two elements 0 and 1. In other words, there exist m elements $\alpha_0, \alpha_1, \ldots, \alpha_{m-1}$ such that each element $\alpha \in \mathbb{F}_{2^m}$ can be written in the form*

$$\alpha = a_0\alpha_0 + a_1\alpha_1 + \cdots + a_{m-1}\alpha_{m-1}, \quad \text{where} \quad a_i \in \{0,1\} \tag{2.1}$$

This set $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ will be called a ***basis*** *of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. Given such a basis each element α can be represented as the bit string $(a_0a_1 \ldots a_{m-1})$. Addition of field elements is performed by bitwise XOR-ing the vector representations. ANSI X9.62 (see [4]) permits two kinds of bases:* ***polynomial bases*** *and* ***normal bases****.*

Polynomial Basis Representation

Definition 2.2.6 *Let $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_2x^2 + f_1x + f_0$ (where $f_i \in \{0,1\}$ for $i = 0,1,\ldots,m-1$). We call f irreducible polynomial of degree m over $\mathbb{F}_2$ if f cannot be factored as a product of two polynomials over $\mathbb{F}_2$, each of degree less than m.*

Each such polynomial f defines a polynomial basis representation of $\mathbb{F}_{2^m}$. Furthermore, we call f the **reduction polynomial**.

Normal Basis Representation

Definition 2.2.7 *A basis of the form $\{\beta, \beta^2, , \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$, where $\beta \in \mathbb{F}_{2^m}$ is called a normal basis of $\mathbb{F}_{2^m}$. Such a basis always exists and an element of it can be written as $a = \sum_{i=0}^{m-1} a_i\beta^{2^i}$, where $a_i \in \{0,1\}$. Such basis has the computational advantage that squaring an element can be done very efficiently. Multiplying distinct elements, on the other hand, can be cumbersome in general. For this reason, ANSI X9.62 specifies that Gaussian normal bases be used, because for them multiplication is both simpler and more efficient.*

2.3 Elliptic Curves over Finite Fields

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

Definition 2.3.1 *Let $p \geq 3$ be an odd prime. An elliptic curve E over $\mathbb{F}_p$ is defined by an equation of the form*

$$y^2 = x^3 + ax + b \tag{2.2}$$

where $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \not\equiv 0 \mod p$. The set $E(\mathbb{F}_p)$ consists of all points (x, y) such that $x, y \in \mathbb{F}_p$, which satisfy equation (2.2), together with a special point O called the point of infinity.

2.3.2 Elliptic curves over $\mathbb{F}_{2^m}$

Definition 2.3.2 *An elliptic curve E over $\mathbb{F}_{2^m}$ is defined by an equation of the form*

$$y^2 + xy = x^3 + ax^2 + b \tag{2.3}$$

where $a, b \in \mathbb{F}_{2^m}$ and $b \neq 0$. The set $E(\mathbb{F}_{2^m})$ contains all points (x, y) such that $x, y \in \mathbb{F}_{2^m}$, which satisfy equation (2.3), together with a special point O called the point of infinity.

3 Ecliptic Curve Digital Signature Algorithm - ECDSA

3.1 Description of the algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the Digital Signature Algorithm (DSA). It was accepted in 1999 as an ANSI standard and was accepted in 2000 as IEEE and NIST standards. It was also accepted in 1998 as an ISO standard and is under consideration for inclusion in some other ISO standards. In this chapter we will describe the ANSI X9.62 ECDSA, and discuss related implementation and security issues.

3.1.1 ECDSA Domain Parameters

The domain parameters for ECDSA consist of a suitably chosen elliptic curve E defined over a finite field $\mathbb{F}_q$ of characteristic p or 2^m and a base point $G \in (\mathbb{F}_q \times \mathbb{F}_q)$. More specifically elliptic curve domain parameters over $\mathbb{F}_q$ shall consist of the following:

- A field size $q = p$ or $q = 2^m$ which defines the underlying field $\mathbb{F}_q$, where $q \geq 3$ is a prime number. In the case $q = p$ the underlying finite field is $\mathbb{F}_p$, which is represented by the integers modulo p. In the case $q = 2^m$, the underlying finite field is $\mathbb{F}_{2^m}$ whose elements are represented with respect to a polynomial or a normal basis as described in Chapter 2.
- An indication FR (field representation) of the representation used for the elements of $\mathbb{F}_q$.
- A bit string seedE (optional).
- Two field elements a and b in $\mathbb{F}_q$ which define the equation of the elliptic curve over $\mathbb{F}_p$ (i.e. $E : y^2 = x^3 + ax + b$) or the equation of the elliptic curve over $\mathbb{F}_{2^m}$ (i.e. $E : y^2 + xy = x^3 + ax^2 + b$).
- Two field elements x_G and y_G in $\mathbb{F}_q$ which define a point $G = (x_G, y_G)$ of prime order n on E i.e. $nG = O$. We note that G is not the point of infinity O.
- The cofactor $h = \sharp E(\mathbb{F}_q)/n$ where $\sharp$ denotes the number of elements in $E(\mathbb{F}_q)$ (optional).

3.1.2 ECDSA Key Pairs

An ECDSA key pair is associated with a particular set of elliptic curve domain parameters. The public key is a random multiple of the base point, while the private key is the integer used to generate the multiple. We briefly describe a key pair generation procedure. An entity A's key pair is associated with a particular set of elliptic domain parameters $D = (q, FR, a, b, G, n, h)$. This association can be assured cryptographically (e.g. with certificates) or by context (e.g. all entities use the same domain parameters). Entity A must have the assurance that the domain parameters are valid.

ECDSA Key Pair Generation

1. Select a random or pseudorandom integer d in the interval $[n - 1]$.
2. Compute the product $Q = dG$.
3. A's public key is Q and A's private key is d.

3.1.3 ECDSA Signature Validation and Verification

This subsection describes the procedure for generating and verifying signatures using ECDSA.

ECDSA Signature Generation

To sign a message m an entity A with domain parameters D and associated key pair (d, Q) does the following steps (For the practical implementation of the steps see Chapter 5):

1. Select a random or pseudorandom integer k such that $1 \leq k \leq n-1$.
2. Compute $kG = (x_1, y_1)$ and convert x_1 to an integer $\overline{x}_1$. In order to do this we use a field element to integer conversion procedure. If $\alpha \in \mathbb{F}_q$ and q is a prime number p then no conversion is required. If $q = 2^m$ then α must be a bit string of length m bits. Let $s_1, ..., s_m$ be the bit string of α from leftmost to rightmost. Then we convert α to an integer satisfying $\overline{x}_1 = \sum_{i=1}^{m} 2^{a_{n-1}f} s_i$ (For more details see [4]).
3. Compute $r = \overline{x}_1 \bmod n$. If $r = 0$ then go to step 1.
4. Compute $k^{-1} \bmod n$.
5. Compute $SHA\text{-}1(m)$ (see 2.1) and convert this bit string to an integer e. In order to do this we use a procedure similar to the one described in Step 2.
6. Compute $s = k^{-1}(e + rd) \bmod n$. If $s = 0$ then go to step 1.
7. A's signature for the message m is (r, s).

Having done that in order for B to verify A's signature for the message m, B obtains an authentic copy of A's domain parameters D and the associated public key Q. It is recommended that B also validates D and Q before beginning to validate the signature through the following steps:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute $SHA\text{-}1(m)$ and convert this bit string to an integer e.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $X = u_1 G + u_2 Q$.
6. If $X = O$ the signature is not valid and should be rejected. Otherwise we continue by converting the x-coordinate x_1 of X to an integer $\overline{x}_1$ and computing $v = \overline{x}_1 \bmod n$.
7. Accept the signature if and only if $v = r$.

In the next chapter we continue by describing the architecture used for practical implementation of ECDSA signature generation and verification.

4 Architecture

4.1 JCrypTool

In this chapter we describe important architecture parts and underlying structures of ECDSA plug-in. At first we introduce basic knowledge about FlexiProvider- toolkit providing cryptographic functionality. After that we give basic facts about Eclipse and Rich Client Platform (RCP). At the end of the chapter we briefly describe JCrypTool plug-in architecture and the place of ECDSA plug-in in it. In some places we use verbatim citations from product portfolios and documentation in order to keep the descriptions short and informative.

The platform independent E-learning program JCrypTool supports users in experimenting with various cryptographic mechanisms like classic, symmetric and asymmetric encryption, digital signatures, hash values, and XML security. JCrypTool uses the Standard Widget Toolkit (SWT) for the GUI together with the Eclipse Rich Client Platform (RCP) for a modular framework. The core JCT framework delivers extension points and offers basic mechanisms like menus, editors, and action history. All these functions are accessible and extendable by plug-ins, so by serving these extension points, plug-in developers only have to concentrate on their plug-in topic.

JCrypTool aims to be a powerful multi-threaded, cross-platform cryptography software with easy to use GUI, implementing strong key-based encryption methods using FlexiProvider and BouncyCastle. For tests and power users an additional command line interface is available.

JCrypTool can be used for learning and teaching cryptography. The Eclipse RCP framework makes JCrypTool extremely extendable and enables other users to easily extend the core application with further cryptographic plug-ins. JCrypTool is developed by a group of open source developers.

More information about JCrypTool can be found on JCrypTool webpage `http://jcryptool.sourceforge.net`.

Most of the computational algorithms of JCryptTool are inherited from FlexiProvider, which is explained in more details in the next subsection.

4.2 FlexiProvider

The FlexiProvider is a powerful toolkit for the Java Cryptography Architecture (JCA/JCE). It provides cryptographic modules that can be plugged into every application that is built on top of the JCA.

The goal of the project is to supply fast and secure implementations of cryptographic algorithms which are easy to use even for developers who are not well-footed in the field of cryptography.

The FlexiProvider has been developed by the Theoretical Computer Science Research Group of Prof. Dr. Johannes Buchmann at the Departement of Computer Science at the Technical University Darmstadt, Germany.

More information about the FlexiProvider can be found on the FlexiProvider webpage (see [11]).

JCrypTool is based on the Eclipse Rich Client technology, which lies on top of Eclipse plug-in Architecture. We explain this in more details in the next subsection.

4.3 Eclipse Plug-in Architecture

An Eclipse Environment consists of many plug-ins. The most common variations of the plug-in interface are the User Interface Plug-in, Resources Plug-in and Variables Plug-in. Plug-ins are the smallest deployable and installable software components of Eclipse. Eclipse Plug-in Architecture is described in details at `http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`.

On top of the Eclipse Plug-in Architecture lies Rich Client Platform (RCP).

4.4 Rich Client Platform

The Rich Client Platform (RCP) is a way to build Java applications that can compete with native applications on any platform.

Because of its unique open source license, one can use the technologies that are included into Eclipse to create own commercial quality programs. Before version 3.0, this was possible but difficult, especially when one wanted to heavily customize the menus, layouts, and other user interface elements. That was because the "IDE-ness" of Eclipse was hard-wired into it. Version 3.0 introduced the Rich Client Platform (RCP), which is basically a refactoring of the fundamental parts of Eclipse's UI, allowing it to be used for non-IDE applications. Version 3.1 updated RCP with new capabilities and, most importantly, new tooling support to make it easier to create than before.

RCP applications are based on the familiar Eclipse plug-in architecture. Therefore, one will need to create a plug-in to be the main program. Eclipse's Plug-in Development Environment (PDE) provides a number of wizards and editors that take some of the drudgery out of the process. PDE is included with the Eclipse SDK download so that is the package one should be using. RCP is described in more detail at `http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html`.

Defining a product, as we see in the next subsection, makes the RCP application much more flexible and configurable.

4.5 RCP Product

In Eclipse terms a product is everything that goes with an application, including all the other plug-ins it depends on, a command to run the application (called the native launcher), and any branding (icons, etc.) that make an application distinctive. Although one can run a RCP application without defining a product, having one makes it a whole lot easier to run the application outside of Eclipse. This is one of the major innovations that Eclipse 3.1 brought to RCP development.

We are not going to get into details about how to create a product and manage it, because we have already one provided by JCrypTool. It is important to understand though that the product is the main configuration place for JCrypTool. Other two important terms involved in Rich Client Platform application development are extensions and extension points.

4.6 Extension Points and Extensions

The loosely coupling of different application components is one of the most characteristic features of Eclipse that should also be used by an RCP-developer. Especially because of the interfaces that are defined in Eclipse with the extension-framework, it is possible to deliver products that are flexible and easy to customize. Therefore Eclipse gives us all needed tools and formalism to cope with them.

If we are developing an application based on the Eclipse-framework, we are using primarily the extension-points provided by Eclipse itself. The concept of the "late binding" can be and shall be used by a developer of a new Eclipse application. Therefore the mechanism is really simple. We are defining an extension point with a minimalist interface and extending the framework with our interface-implementation.

Each plug-in can define extension-points which define possibilities for functionality contributions (code and non-code) by other plug-ins. Non-code functionality contributions are for example the provision of help content.

A plug-in can use extensions, e.g. provide functionality to these extension points. In general an extension point can be used several times (either by the same plug-in or by other plug-ins).

The basis for this architecture is the runtime environment of Eclipse which is based on the OSGI Alliance. Eclipse used the OSGI reference implementation Equinox to run upon. The plug-in concept of Eclipse is the same as the bundle concept of OSGI. Generally speaking an OSGI bundle equals a plug-in and vice-versa.

The used extensions and the provided extension-points are described in the file plugin.xml. This file is an .xml file which can be edited via the PDE (Plug-in Development Environment) which provides a nice user interface for editing this xml file.

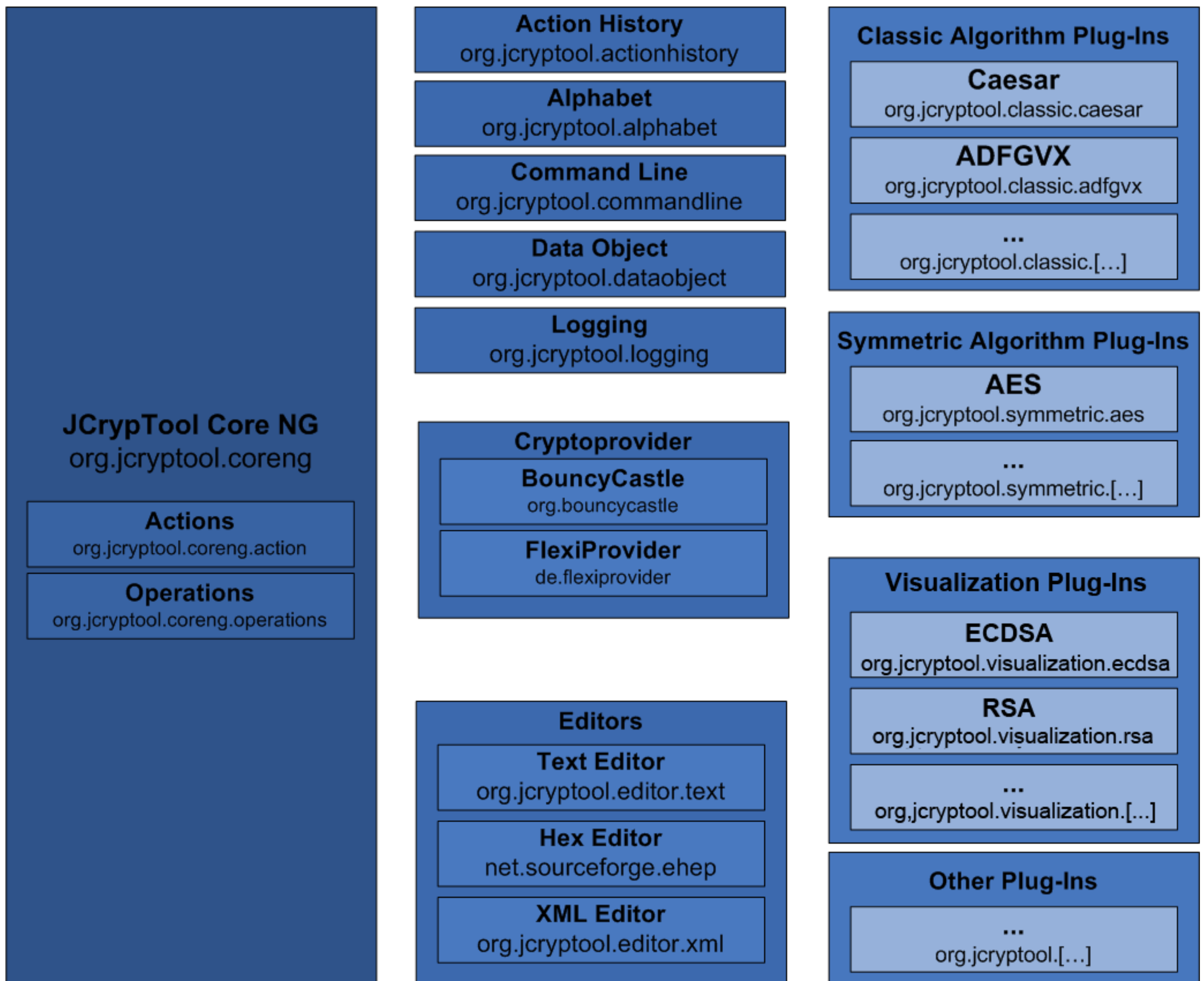

Figure 4.1: JCrypTool plug-ins Architecture and our plug-in placed in it

Further information on Extension Points and Extensions can be found in Eclipse Extension Points and Extensions Tutorial by Lars Vogel at `http://www.vogella.de/articles/EclipseExtensionPoint/article.html`.

In the next subsection we will explain more about JCrypTool Architecture.

4.7 JCrypTool Architecture

To understand how to develop plug-ins for JCryptTool we need to get familiar with the JCrypTool architecture.

The org.jcryptool.coreng plug-ins build the core framework and contain only the bare minimum to run JCrypTool. Those plug-ins would be enough for an empty runtime. However in order for the crypto plug-ins to work, crypto providers such as FlexiProvider as well as editors such as EHEP (see [16]) are required as well.

We are going to develop a plug-in for ECDSA. According to the JCrypTool Architecture and naming conventions this plug-in takes the name org.jcryptool.visual.ecdsa. This means it is a visualization plug-in with the name "ecdsa".

Figure 4.1 illustrates JCrypTool plug-ins Architecture and the position of our plug-in in it.

5 Design and Implementation

5.1 Requested Functionality

This bachelor thesis develops an E-learning plug-in for JCrypTool that explains the generation and verification of ECDSA signatures. It is user-friendly and self explaining. Users should be able to understand how the algorithm works and be able to play with different variables and see their effect on different steps of the process of generation and verification of signature. A multi-language environment should also be considered.

5.2 Intention

The idea of the plug-in is to provide users with ability to use elliptic curve digital signature generation and verification and to give them access to the inner workings and details of different steps of the algorithm. To allow this, some of the real world restrictions on the algorithm are removed and others are simplified. As an end effect the user is allowed to generate signatures that do not confirm to the actual specifications. This is done in order to have computational tasks simple enough that the normal user can keep track of the calculations without the need of special computation devices.

5.3 Design and User Interface

User interface is one of the crucial points of an E-learning application. Users should be able to easily grasp the whole process and not be distracted by unnecessary or unneeded information. At the same time the application should be able to assist the user with hints and tell him if he is doing something not in the way he is supposed to.

Taking that into account after starting the plug-in only the inputs needed for the first step of the generation are enabled. All the unneeded for the current step inputs and fields are grayed out to keep the user focused on the current task.

ECDSA is a complex algorithm and consists of many steps. Because most of the steps interact with each other the plug-in user interface consists of a single page represented by a tab in the JCrypTool program. This ensures more space for the plug-in. Also having all the algorithm steps in one place allows the user to keep track of what has changed in previous steps of the algorithm. Furthermore, it enables the user to see how start parameters combine and which variables are derived by which start parameters and other variables.

The plug-in is intended to be used in two general ways:

- Computation mode: The user enters start parameters and follows the algorithms until it's finished.
- Challenge mode: The user is able to interact with every step of the algorithm. For every step the user can try to compute alone the result, type it in the input field and check if his own computations were right.

To give the user maximum freedom to play with different values of the variables, ECDSA plug-in allows user to type any value in the input fields. For every input field there is a hint popup window which gives instructions how to obtain right values or for example what are the restrictions for this parameter. If wrong values are entered the user gets a message about which exact restriction is not fulfilled. Here, it has to be noted that due to the E-learning nature of the plug-in some of the real-world restrictions are simplified or totaly removed to allow values that can be computed by a human being in reasonable time without the use of special computational devices.

User interface is divided into three major sections (see Figure 5.1):

- Domain Parameters: Here all parameters needed to generate ECDSA signature are being selected.
- Elliptic Curve Digital Signature Generation: Here the signature generation algorithm is divided into small user-friendly and easy to be calculated steps.
- Elliptic Curve Digital Signature Verification: Analog to the generation part here all the steps for verification of an ECDSA signature are included.

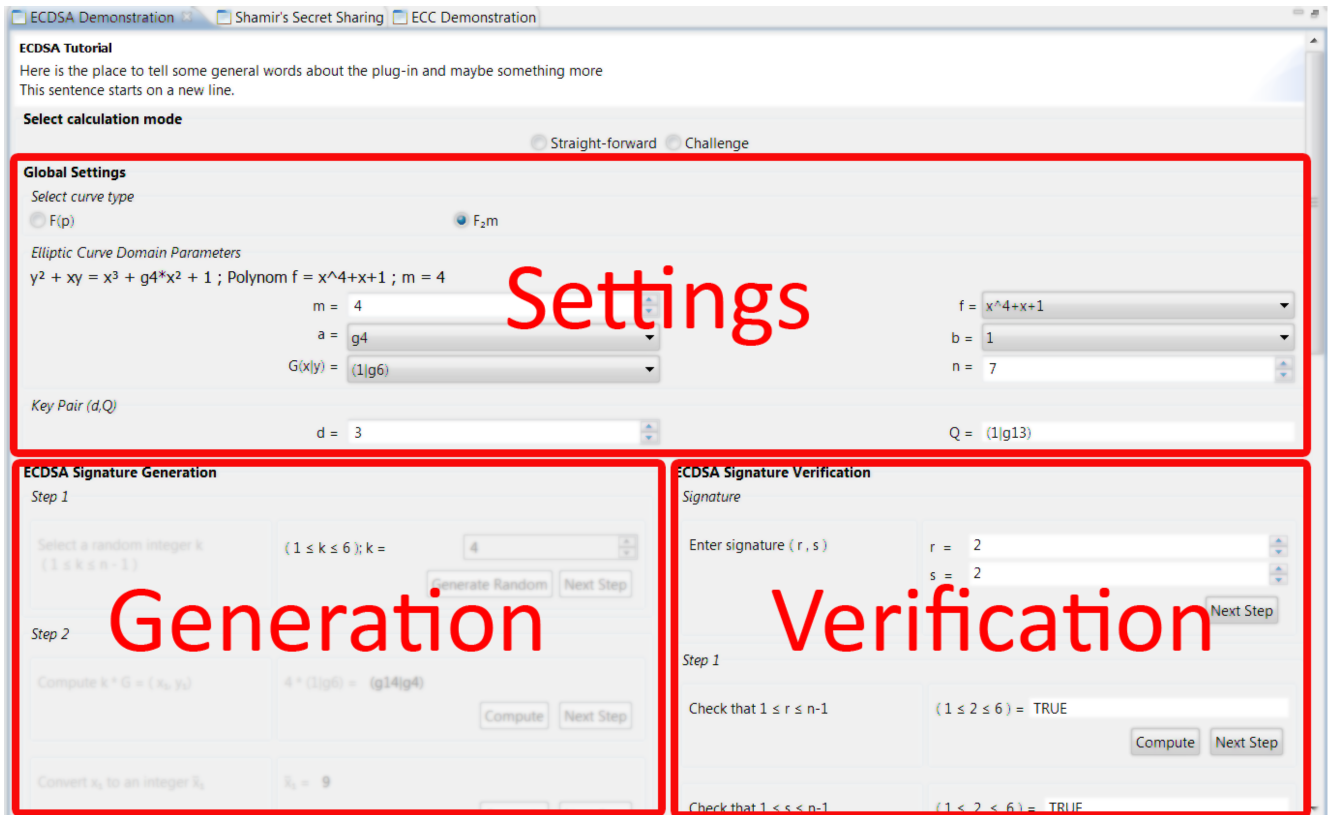

Figure 5.1: ECDSA plug-in main GUI sections

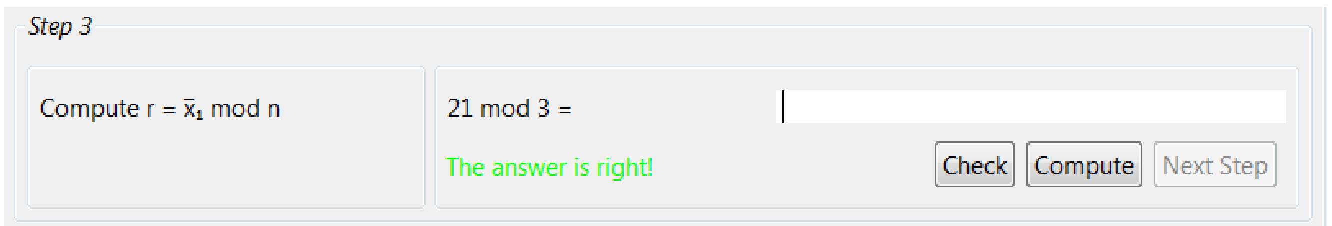

Figure 5.2: Single step overview

We will consider each of these parts later in this document.

All major parts of the plug-in are easy to distinguish thanks to the use of different font styles. Major sections are bolded and subsections are italic. Only one font type is considered so that the interface does not get overcrowded and thus non-readable.

ECDSA is defined over the fields $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$. There is a radio switch at the top of the user interface to toggle between these two fields. We will explain more about these issues in Section 5.4.

The generation and verification algorithms are broken into small steps. Every step is divided into three major parts (see Figure 5.2).

- Explanation part: Here is written (usually in words) what exactly is happening in this step. Variables are denoted by their alphabetic names. The user sees the general condition or equation that has to be solved in order to proceed to the next step.
- Calculation part: Here the equation from the explanation part is rewritten with concrete values. Thus, the calculation steps are made tractable and more transparent to the user at calculation time.

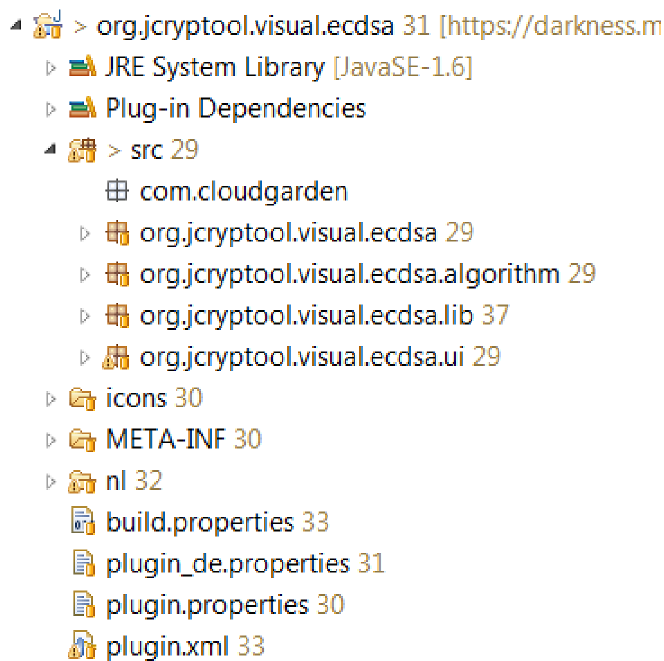

Figure 5.3: Class structure overview

- Challenge mode part: A blank field is available for the user to input his own solution of the step's equation. It is checked against the real answer and evaluated. After that the user becomes a message whether the entered answer is right or wrong.

Once a value of a variable is known it is automatically put into the calculation part of every step it is used in. Before it is calculated the user sees just the alphabetical name of the variable.

At a certain point in time only a single step is active. All other steps are grayed out to focus the attention of the user to the current task. Only once the correct answer for the current step is calculated the user is presented with the possibility to go to the next step and the already completed step is grayed out. At the same time the result is populated among future steps that are making use of it.

An important feature of the ECDSA plug-in is also the ability to verify an ECDSA signature without having to generate it first. This means that the user can use the plug-in starting either from the verification or from the generation part.

Some of the computational parts of the plug-in are based on an elliptic curve visualization plug-in already available in JCrypTool. They are extended and parts of them are rewritten to suit the needs of the ECDSA E-learning plug-in.

5.4 Implementation

5.4.1 Internationalization

Providing a software product solely in English is no longer acceptable from a usability, quality, and marketing standpoint. That is why JCrypTool utilizes Eclipse Plug-in Internationalization. The ECDSA plug-in as part of the JCrypTool is also written taking into account internationalization and all the user interface strings are externalized in configuration files. More on internationalization can be found under (`http://www.eclipse.org/articles/Article-Internationalization/how2I18n.html`).

5.4.2 Class Hierarchy

The source code is divided into four packages. (see Figure 5.3)

- org.jcryptool.visual.ecdsa: Contains standard plug-in classes for initializing and starting of the plug-in.

- org.jcryptool.visual.ecdsa.algorithm: The classes dealing with computation in different fields are included here. Furthermore the different field member objects are defined here.
- org.jcryptool.visual.ecdsa.lib: Includes small helper classes not directly connected to the algorithm.
- org.jcryptool.visual.ecdsa.ui: Here all GUI related classes are included (building of different parts of the GUI, transition between different steps etc.).

There are separate classes for each of the fields where elliptic curve points lie. In our case these include the class ECFp.java (representing points in $(\mathbb{F}_p \times \mathbb{F}_p)$) and the class ECFm.java (representing points in $(\mathbb{F}_{2^m} \times \mathbb{F}_{2^m})$). These classes define arithmetics over these fields, such as addition, multiplication, and finding an inverse. Both of them are based on the arithmetics for elliptic curves in the real numbers (represented by the class EC.java).

Two different objects are representing points in $(\mathbb{F}_p \times \mathbb{F}_p)$ and in $(\mathbb{F}_{2^m} \times \mathbb{F}_{2^m})$. They provide methods for getting, setting, and representing the point in a human readable format. As an example they are implementing the $toString()$ method that outputs a point in the format (x, y).

GUI classes are also inherited from a main GUI class to make use of the object oriented paradigm of Java and to ensure unique implementation of most methods and GUI elements among derived classes. For example the class ECDSAForFm.java extends the class ECDSAForFp.java.

5.4.3 Technical Implementation

Because of the nature of elliptic curves over $(\mathbb{F}_{2^m} \times \mathbb{F}_{2^m})$ the GUI for choosing domain parameters and key pair variables is implemented with different controls. For example to give the user the ability to choose domain parameters easy in $\mathbb{F}_{2^m}$ the possible polynomials are precalculated and offered as a selection from a drop-down box. Another reason is that in $\mathbb{F}_p$ most of the parameters are integers and in $\mathbb{F}_{2^m}$ are polynomials. The same base GUI class methods are used for building the interface for both $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ and only the input elements that handle polynomials in $\mathbb{F}_{2^m}$ are overwritten by the GUI class responsible for ECDSA in $\mathbb{F}_{2^m}$.

In the process of generation or verification of the ECDSA signature we are using many special mathematical symbols to denote different variables or operations. This include sub- and superscripts, use of greek letters and others. For example: $\mathbb{F}_{2^m}$, x_G, x^3 etc. Because Java Swing supports Unicode characters, all of these special symbols are represented by Unicode character combinations, for example "x\u00b2" for x^2. The full list of special mathematical characters can be found in the latest version of the technical report "Unicode Support for Mathematics" [14] and [15].

The process of digitally signing an electronic message involves the use of hash algorithms. Traditionally SHA-1 is used. Additionally a simplified fast hash algorithm is implemented specially for use with the ECDSA plug-in. It is designed to return small hash values to enable more transparent calculations of the resulting hash value. The user has the ability to choose which one of the two hashing methods to use. If SHA-1 is used only the last two numbers of the resulting hash value are taken into account.

5.5 Workflow

The first thing for a user to choose is whether to go into a challenge mode or simple calculation mode. According to this choice the challenge input fields for every step are made visible or hidden. Using this technique building a special interface for the challenge mode has been spared.

After choosing the mode, the user is presented the opportunity to choose the underlying field $\mathbb{F}_q$ ($\mathbb{F}_{2^m}$ or $\mathbb{F}_p$) of the elliptic curve $E(\mathbb{F}_q)$. The user makes his choice by invoking one of the radio buttons at the top of the window for $\mathbb{F}_{2^m}$ or $\mathbb{F}_p$ respectively.

5.5.1 ECDSA Domain parameters

The next section in the "Elliptic Curve Domain parameters" (see Figure 5.4) is modified according to user preference. Some fields are made visible, others that are unused are hidden.

Elliptic Curve Domain Parameters

y² mod 23 = (x³ + 10x + 15) mod 23

p = 23

a = 0 b = 0

G(x|y) = (1|7) n = 13

Key Pair (d,Q)

d = 3 Q = (16 , 4)

Figure 5.4: Domain parameters selection box for $\mathbb{F}_p$

Elliptic Curve Domain Parameters

y² + xy = x³ + g4*x² + 1 ; Polynom f = x^4+x+1 ; m = 4

m = 4 f = x^4+x+1

a = g4 b = 1

G(x|y) = (1|g6) n = 7

Figure 5.5: Domain parameters selection box for $\mathbb{F}_{2^m}$

In the case of $\mathbb{F}_{2^m}$ domain parameters (see Figure 5.5) selection of the power parameter m is restricted to maximum 6. This is done in order to keep the simplicity of calculations. This also enables the predefining of the field representation for the domain parameters. Once all the parameters needed to define an elliptic curve are gathered, all the points on the curve are automatically generated. At this moment the user can play with different parameters of the elliptic curve equation and see which points are part of the curve, how the exact equation looks like, etc. Next step is to select a private key pair d. A public key Q is then calculated and a key pair (d, Q) is presented.

There are two possible ways to continue using the plug-in. One way is to start the digital signature generation and the other is to start the digital signature verification.

5.5.2 ECDSA Signature Generation

If the user chooses to continue with a digital signature generation the algorithm starts from Generation step 1. The workflow continues as described in Figure 5.6. An important point in the algorithm is Step 2. Before proceeding with the algorithm a field element-string conversion is performed (as described in [4, Sec. 4.3]). In step 6 the user can choose which hashing algorithm to use. For differences between the two hash algorithms and their implementation see Section 5.4.3.

In some steps, as for example Step 4 and Step 8, there is a decision making procedure which decides whether to continue with the algorithm or start all over, based on an equality check.

After finishing the generation the user sees the resulting values for the digital signature (r, s) of the message. This signature can be plugged as an input in the second part of the plug-in "ECDSA Signature Verification". As already mentioned this part is independent of "ECDSA Signature Generation" and can be invoked on its own.

5.5.3 ECDSA Signature Verification

Analogically to ECDSA Signature Generation part of the plug-in the user follows step by step the algorithm as represented graphically in Figure 5.7

5.6 Example of Valid Signatures

In this section we give an example of two possible valid signature sets of parameters for both underlying fields $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$.

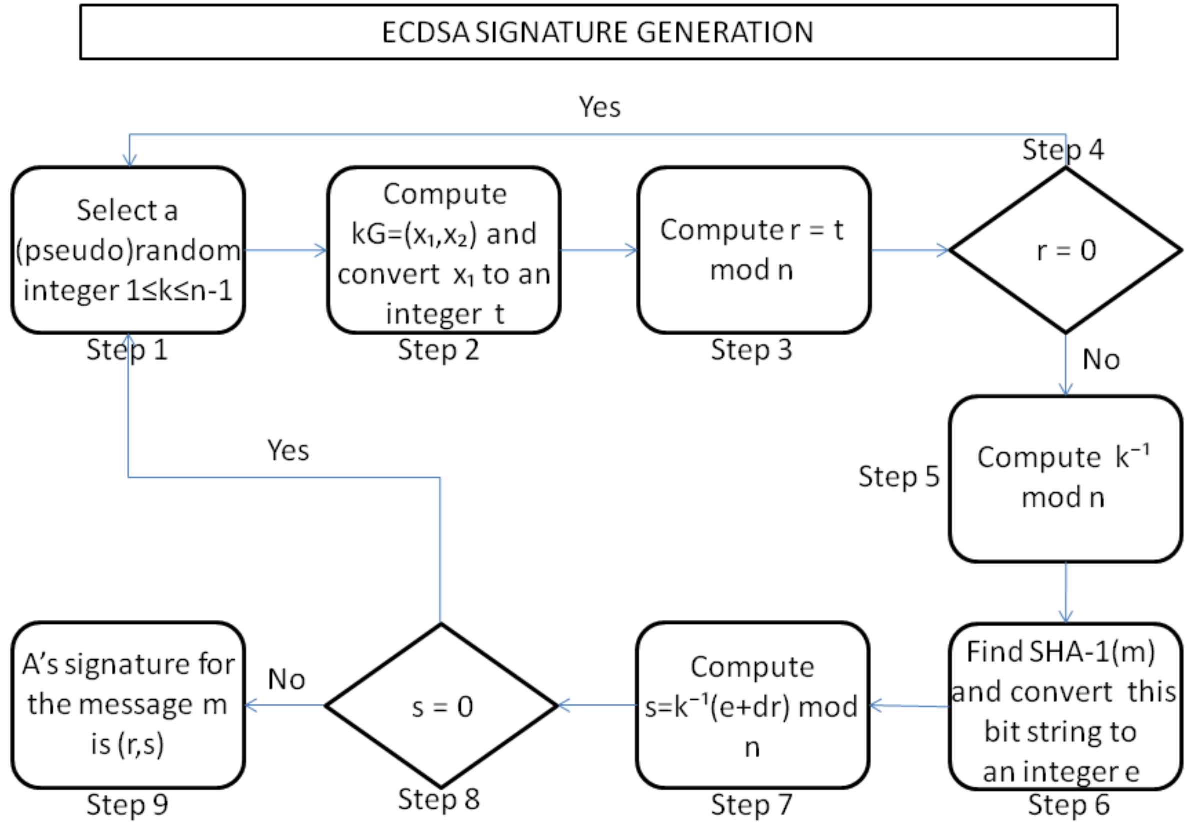

Figure 5.6: Workflow ECDSA Signature Generation for domain parameters (q, FR, a, b, G, n, H) and a key pair (d, Q)

5.6.1 Example of Valid Signature for $\mathbb{F}_p$

Let the message be "RESULT". Select domain parameters $D = (q, FR, a, b, G, n, h)$ such that $q = p = 23$, $a = 10$, $b = 15$, $G = (1, 7)$, $n = 13$. We note that p is a prime number greater than 2 and the point G has an order $n = 13$ that is $nG = O$. Furthermore $n = 13$ divides the number of elements $\sharp E(\mathbb{F}_p) = 26$ on the curve $y^2 = x^3 + 10x + 15 \bmod 23$. We choose an integer $d = 13$ for the private key and compute the public key $Q = (16, 4)$. These domain parameters are valid and generate a valid ECDSA Signature $(r, s) = (7, 5)$.

1. Choose "randomly" $k = 7$.
2. Compute $kG = (x_1, y_1) = (20, 2)$.
3. Convert x_1 to an integer 20.
4. Compute $r = x_1 \bmod n$ and get $r = 7$.
5. Check that 7 does not equal 0.
6. Compute $k^{-1} \bmod n$ and get as a result 2.
7. Compute hash of the message using simple hash and get result 79.
8. Compute $s = k^{-1}(e + rd) \bmod n$ and get result $s = 5$.

By choosing these domain parameters the ECDSA signature of the message is $(r, s) = (7, 5)$.

Now we have to verify the signature.

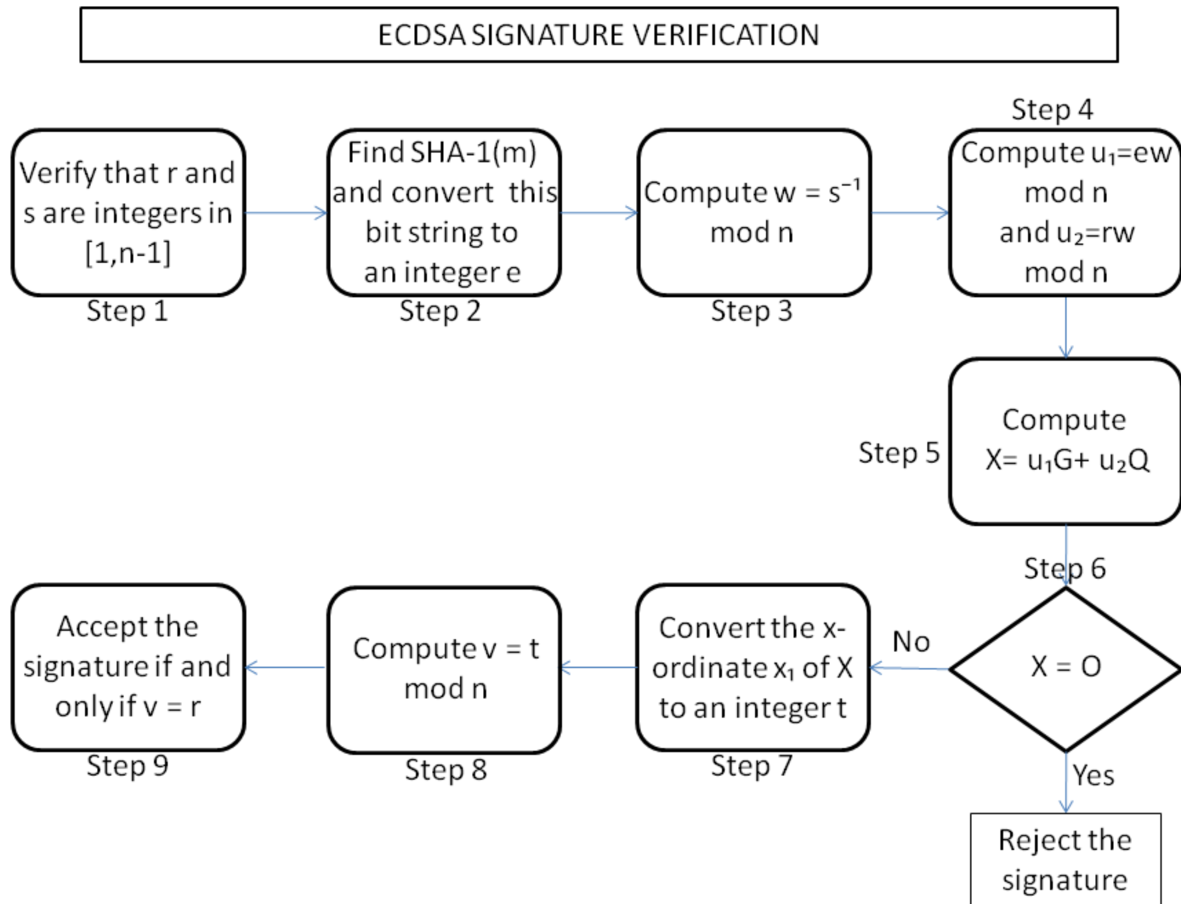

Figure 5.7: Workflow ECDSA Signature (r, s) on message m Verification for domain parameters (q, FR, a, b, G, n, H) and a public key Q

1. Verify that $r = 7$ and $s = 5$ are integers in the interval $[1, 12]$. Correct.
2. Compute hash of the message using simple hash and get result 79.
3. Convert this bit string to an integer $e = 79$.
4. Compute $w = s^{-1} \bmod n$ and get result 8.
5. Compute $u_1 = ew \bmod n$ and get result $u_1 = 4$.
6. Compute $u_2 = rw \bmod n$ and get result $u_2 = 1$.
7. Compute $X = u_1 G + u_2 Q$ and get result for $X = (20, 2)$.
8. Check that $X = O$ is false.
9. Convert the x-coordinate x_1 of X to an integer $\overline{x}_1$ and get result 20.
10. Compute $v = \overline{x}_1 \bmod n$ and get result 7.
11. Accept the signature because $v = 7 = r$.

5.6.2 Example of Valid Signature for $\mathbb{F}_{2^m}$

Let the message be "RESULT". Select domain parameters $D = (q, FR, a, b, G, n, h)$ such that $m = 4$, $f = x^4 + x + 1$, $a = g13$ $b = g6$, $G = (g3, g2)$, $n = 7$. Here g denotes the element which generates all elements representing the elliptic curve E modulo f. We note that p is a prime number greater than 2 and the point G has an order $n = 7$ that is $nG = O$. Furthermore $n = 7$ divides the number of elements $\sharp E(\mathbb{F}_p) = 14$ on the curve $y^2 + xy = x^3 + (g13)x^2 + g6$ with polynom $f = x^4 + x + 1$. We choose an integer $d = 3$ for the private key and compute the public key as $Q = (g14, g9)$. These domain parameters are valid and generate a valid ECDSA Signature $(r, s) = (2, 2)$.

1. Choose "randomly" $k = 4$.
2. Compute $kG = (x_1, y_1) = (g14, g4)$.
3. Convert x_1 to an integer 9.
4. Compute $r = x_1 \bmod n$ and get $r = 2$.
5. Check that 7 does not equal 0.
6. Compute $k^{-1} \bmod n$ and get as a result 2.
7. Compute hash of the message using simple hash and get result 79.
8. Compute $s = k^{-1}(e + rd) \bmod n$ and get result $s = 2$.

By choosing these domain parameters we get ECDSA signature of the message $(r, s) = (2, 2)$.

We do the following steps in order to verify the signature.

1. Verify that $r = 2$ and $s = 2$ are integers in the interval $[1, 6]$. Correct.
2. Compute hash of the message using simple hash and get result 79.
3. Convert this bit string to an integer $e = 79$.
4. Compute $w = s^{-1} \bmod n$ and get result 4.
5. Compute $u_1 = ew \bmod n$ and get result $u_1 = 1$.
6. Compute $u_2 = rw \bmod n$ and get result $u_2 = 1$.
7. Compute $X = u_1 G + u_2 Q$ and get result for $X = (g14, g4)$.
8. Check that $X = O$ is false.
9. Convert the x-coordinate x_1 of X to an integer $\overline{x}_1$ and get result 9.
10. Compute $v = \overline{x}_1 \bmod n$ and get result for $v = 2$.
11. Accept the signature because $v = 2 = r$.

6 Conclusion

This thesis concludes with a summary of the central point and an outlook of future developments.

6.1 Summary

This thesis described the Elliptic Curve Digital Signature Generation and Verification Algorithm. An ECDSA plug-in has been developed and a description of the E-learning platform has been written. Guidelines for implementing ECDSA Algorithm in an E-learning environment have been presented. Simplified versions of hashing algorithms have been implemented to conform with the E-learning nature of the plug-in. Implementation of ECDSA Generation and Verification Algorithm in both $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ is written. A frame for extending the plug-in is made available.

6.2 Outlook

The plug-in has laid the foundation for future development of similar E-learning plug-ins. Future developers have the ability to extend the plug-in to real world cases, for example extending the spectrum of the domain parameters. Key-pair generation is also another point for future extensions. In general this plug-in can be viewed as an example for future implementations of JCrypTool plug-ins.

Bibliography

[1] [Johnson, Don; Menezes, Alfred; Vanstone, Scott] The elliptic curve digital signature algorithm, Proceedings of the 7th conference on USENIX Security Symposium - Volume 7, 1998

[2] [Smart, Nigel] Introduction to cryptography, London Mathematical Society Lecture Note Series 317, Cambridge University Press, 2005

[3] [Blake, I.] Elliptic curves in the cryptography, London Mathematical Society Lecture Note Series 265, Cambridge Univeristy Press, 2002

[4] [Working Draft, AMERICAN NATIONAL STANDARD] X9.62-1998 Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)

[5] [National Institute of Standards and Technology] Secure Hash Signature Standard (SHS) (FIPS PUB 180-2).

[6] [Bassham, L.] Representation of elliptic curve digital signature algorithm keys and signatures, Internet draft, June 1999

[7] [Lopez, J.] An Overview of Elliptic Curve Cryptography, Technical Report IC-00-10, State University of Campinas, 2000

[8] [Hankerson, Darrel; Menezes, Alfred; Vanstone, Scott] Guide to Elliptic Curve Cryptography, Springer, 2004

[9] [JCrypTool] `http://jcryptool.sourceforge.net/`

[10] [Eclipse RCP] `http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html`

[11] [Flexiprovider] `http://www.cdc.informatik.tu-darmstadt.de/flexiprovider`

[12] [Eclipse Internationalization] `http://www.eclipse.org/articles/Article-Internationalization/how2I18n.html`

[13] [ANSI X9.62-1998] `http://www.securitytechnet.com/resource/crypto/standard/`

[14] [Unicode Support for Mathematics] `http://www.unicode.org/reports/tr25`

[15] [The Unicode Standard 5.2] `http://unicode.org/charts/PDF/U2070.pdf`

[16] [Eclipse Hex Editor Plug-in] `http://ehep.sourceforge.net`

[17] [Representation of elliptic curve digital signature algorithm keys and signatures] `http://www.ietf.org`