Design and Implementation of Plug-ins for JCrypTool

Visualization of the Merkle-Hellman Algorithm and of Hash Sensitivity
Diploma thesis by Ferit Dogan
March 2015

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Theoretische Informatik
Kryptographie und Computeralgebra
Prof. Dr. Johannes Buchmann

TECHNISCHE UNIVERSITÄT DARMSTADT

JCrypTool
The cryptography e-learning platform.

Design and Implementation of Plug-ins for JCrypTool
Visualization of the Merkle-Hellman Algorithm and of Hash Sensitivity

Diploma thesis by Ferit Dogan

1. Consultant: Prof. Dr. Johannes Buchmann
2. Consultant: Dr. Mohamed Saied Emam Mohamed

Day of submission:

Acknowledgements

I'd like to say thanks to everybody supporting me during my study and during this thesis.

A very special thank goes to Prof. Bernhard Esslinger, the head of the open-source project CrypTool, who was always responsive and gave valuable feedback concerning the design of the two visualizations.

Also, I'd like to thank Dominik Schadow, head of the JCrypTool team, who helped solving some technical problems.

And last but not least, I'd like to thank Dr. Mohamed Saied Emam Mohamed for looking after my thesis. And Prof. Dr. Johannes Buchmann for allowing to write my thesis at his faculty and for his support for open-source software.

Declaration

I hereby declare that this Diploma thesis is my original work and it has been written by me in its entirety. I have acknowledged all the sources of information which have been used in the thesis.

Darmstadt, 10th March 2015

(Ferit Dogan)

Conventions

For a better understanding, there are some typographical conventions that are followed in this Diploma thesis.

- `Courier` - Code examples, class, method and file names
- **Bold** - Name of the user interface elements such as menus, buttons, input boxes, window titles and plug-in names.
- *Italic* - Formulas and mathematical constructs.

This diploma thesis was written entirely in LaTeX. For the external layout the corporate design of the **Technical University of Darmstadt** was chosen. Served for the internal structure of the thesis the following literature: [MG00], [Kop00] and [Kna04]. For the design of Eclipse plug-ins we used the books [EC09] and [DG03].

Abbreviations

- GUI = Graphical User Interface
- IDE = Integrated Development Environment
- JCT = JCrypTool
- OSGi = Open Service Gateway initiative
- RCP = Rich Client Platform
- UI = User Interface

Abstract

JCrypTool is the Java and Eclipse Rich-Client-Platform (RCP) based cryptography e-learning platform. Developed as an open-source project, it enables students, teachers, developers, and anyone else interested in cryptography to apply and analyze cryptographic algorithms in a modern, easy-to-use application. Its aim is to create a new form of e-learning by not just encouraging users to learn about cryptography and apply the algorithms themselves, but also to develop their own cryptographic plug-ins and extend the JCrypTool platform in new directions. JCrypTool already includes a wealth of cryptographic mechanisms including classic, symmetric, and asymmetric encryption, hash functions, analysis tools, visualizations, and crypto games[1].

This diploma thesis describes the design and the implementation of two plug-ins, **Visualization of the Merkle-Hellman Algorithm and of Hash Sensitivity** for the JCrypTool framework. The Merkle-Hellman algorithm is an asymmetric encryption scheme based on the knapsack problem, developed by Ralph Merkle and Martin Hellman in 1978. Hash Sensitivity considers some hash functions such as MD5, SHA etc. The aim of this thesis is to implement suitable visualizations for both topics.

[1] https://www.cryptool.org/en/jcryptool-en

Contents

1 Introduction

This diploma thesis describes the design and the implementation of two plug-ins, **Merkle-Hellman** and **Hash Sensitivity**, for the e-learning platform **JCrypTool (JCT)**[1]. JCrypTool is a cryptographic e-learning platform, which offers visualizations of various cryptographic algorithms. It is used in schools and at universities by students, teachers and cryptography interested persons to understand the functionality of cryptographic algorithms. It is an open-source project that bases on Eclipse Rich-Client-Platform[2]. JCrypTool also provides the opportunity to implement new plug-ins to add new functionalities to the JCrypTool platform. The head of this project is Prof. Bernhard Esslinger from the University of Siegen.
The goal of this diploma thesis is to implement two adequate visualizations of the above named algorithms for JCrypTool, so that the users are supported in understanding these cryptographic methods better, and can do self-studies and experiments with the according algorithms and their intensions. Both plug-ins and their online documentation are implemented in both English and German.

In this chapter we explain the functionality of RCP and show how you can create/develop a plug-in. In the chapters 2 - 3 the plug-ins are demonstrated and in chapter 4 we finish our thesis with an outlook.

1.1 The JCrypTool Project

With the help of JCrypTool users can get a better overview and faster understanding of cryptographic functions such as symmetric/asymmetric cryptographic systems, (public/private) key generation, encryption/decryption, hash functions, and cryptanalysis of systems. They even can implement their own games at this project. The goal of the JCrypTool project is to make users aware of how cryptography can help against network security threats and to explain the underlying concepts of cryptology. The software JCT is available in English and German. It runs on the operating systems Windows, Linux and MacOS.

The JCrypTool consists of two parts: **JCrypTool Core**[3] and **JCrypTool Crypto**[4]. The Core project contains the editors (Hex and text), the views (Crypto-Explorer view and Action view), the logging, the help, cryptographic functions and crypto providers (FlexiProvider and Bouncy Castle). The Crypto project contains new cryptographic plug-ins in the categories algorithms, analyses, games, and visualizations.

1.2 Eclipse Rich Client Platform

Eclipse RCP is a platform for software developments, which was created in 2004 from the Eclipse IDE[5] and additionally includes common ingredients. The components of the backbone include the workbench structure, user interface, extensible plug-in system, the aid component and the update manager. These modules enable developers to implement their own application functions in a simple way. Since June 26th, 2013 Eclipse is available in version 4.3 as Kepler release. For the implementation of the plug-ins for this diploma thesis the Kepler release was used, but as a target platform version 3.7 was used which serves as the basis for the current version JCrypTool. The creation of an RCP application consists of the following basic components:

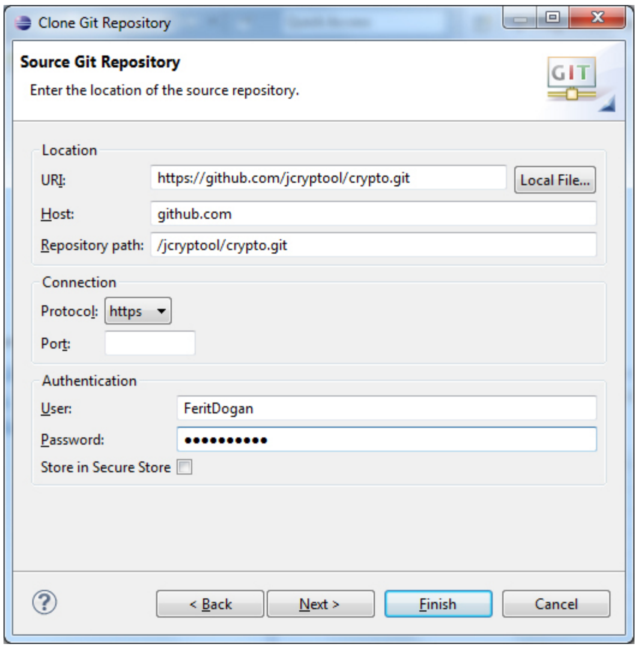
- OSGi[6] specifies a Java runtime environment that enables the execution of modules (so-called bundles or plug-ins). Eclipse implements this specification with Eclipse Equinox. Eclipse applications consist of a variety of plug-ins that run in Equinox.
- Eclipse Core Runtime provides the not UI[7] related functionalities for Eclipse applications. Among others, the life cycle of Eclipse applications is managed so that it is responsible for the start and initialization of applications.
- Standard Widget Toolkit (SWT) is the UI toolkit developed by the Eclipse platform and provides the native UI widgets such as layouts and components (buttons, text entry fields, tables, ...) of the operating system for the construction of an application with a graphical user interface.

[1] JCT: JCrypTool – http://www.cryptool.org/de/jcryptool
[2] RCP: Rich-Client-Platform – http://www.eclipse.org/
[3] https://github.com/jcryptool/core
[4] https://github.com/jcryptool/crypto
[5] IDE: integrated development environment
[6] OSGi: Open Service Gateway initiative
[7] UI: User Interface

- JFace provides the functionalities that are needed to fill the UI widgets that are provided by SWT with data from Java model objects.

1.3 Setting up the development environment

JCrypTool was originally hosted at Sourceforge[8] as Subversion[9] repository. Due to the increasing popularity of the version management program GIT[10] it was decided to migrate to GIT. Thus, since August 11th, 2011 the source code of JCrypTool is available on GitHub[11] as a GIT repository. GIT, in comparison to SVN, has the advantage that you can also check in your changes offline. In order to integrate JCrypTool into the development environment you have to check out both the Core and the Crypto project of JCrypTool. Because this diploma thesis deals with the visualization of algorithms you can check out the Core project as read-only. For the Crypto project you must have permission for this repository to check in your changes; this is in turn controlled by the JCrypTool team. To check out the source code from GitHub now you can go into Eclipse Git Repository Exploring perspective and select `Clone a Git repository`. Now you can select the URL and switch with `Next` to the next screen, where you can get your login information, as shown in Figure 1.1 and enter them to check out the Master branch of the repository.


Clone Git Repository

Source Git Repository
Enter the location of the source repository.

Location
URI: https://github.com/jcryptool/crypto.git
Host: github.com
Repository path: /jcryptool/crypto.git

Connection
Protocol: https
Port:

Authentication
User: FeritDogan
Password: ••••••••••
Store in Secure Store ☐

Figure 1.1: Check out the Crypto project of JCrypTool

After the checkout is completed, you must still import the projects into the Eclipse workspace. For this, you click with the right mouse button on the repository and select `Import Projects...` With `Import existing projects` you can integrate all existing projects of the repository into the workspace. In order to check in/out changes flexibly you should create a new branch with `Switch To - New Branch...` This allows for a simpler handling of the commits when you merge into the master branch later. The same procedure must be done for the JCrypTool Core project to set up all the necessary resources locally. In Conclusion you should still include the target platform, which is included in the project org.jcryptool.product, so that you can directly develop against it. For this you go into the Eclipse Preferences and change the target platform by selecting the file jcryptool.target and confirm with `OK`. The file `jcryptool.product` which is in the project `org.jcryptool.product` can now be executed to start JCrypTool.

8 http://sourceforge.net
9 Subversion (SVN) version control software – http://subversion.apache.org/
10 http://git-scm.com
11 GitHub is a web hosting service for software development projects.

2 Merkle-Hellman plug-in

1976 Whitfield Diffie and Martin Hellman presented a new concept in cryptography, the so-called public-key cryptography. Two years later, 1978, Ralph Merkle and Martin Hellman released their cryptosystem: The Merkle-Hellman algorithm is an asymmetric cryptosystem which bases on the **knapsack problem/subset sum problem** [RM78]. It was also an alternative idea to RSA. 1983 Adi Shamir and Richard Zippel broke this cryptosystem [Sha84].

2.1 Merkle-Hellman algorithm

As the Merkle-Hellman algorithm is based on the subset sum problem it is helpful to bring some definitions from the subset sum problem. For this problem first we define the term *knapsack vector*. A *knapsack vector* consists of a set A which contains positive and pairwise different integers: $A = (a_1, a_2, ..., a_n)$, $a_i \neq a_j$ if $i \neq j$. For our algorithm we need a superincreasing vector which requires that the set A consists of positive and pairwise different integers.

In the following, we explain what the *subset sum problem* is: The subset sum problem is an important problem in cryptography which is also described as a special case of the knapsack problem. It is NP-complete this means there is no polynomial algorithm to solve it. We denote here two aspects for the following description: Known and Unknown.

Known: A knapsack vector $A = (a_1, a_2, \ldots, a_n)$ and the sum s that is a positive integer.

Unknown: Is there a subset A' of A with $\sum$ a' in A'(a') = s, or equivalent: Does there exist a vector $X = (x_1, x_2, ..., x_n)$, x_i in $(0, 1)$, with $A * X = s$?

$A * X$ is defined as: $a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n$

The problem described in Unkown is called decision problem. To get a solution the following task must be computed: $X = (x_1, x_2, ..., x_n)$, x_i in (0,1) such that $A * X$ = s The functional problem of a knapsack vector A together with a sum s is given as (A, s).

Now we give a small example for the subset sum problem: A is the set with the following elements A = (8, 7, 20, 32, 13) and the sum s is 40. A possible solution for (A, 40) is illustrated by the vector X = (1, 0, 0, 1, 0) then $A*X = a_1 + a_4 = 8 + 32 = 40$. The corresponding subset is $A' = (8, 32)$. This solution is not the only one for our example, there may be several solutions for (A, 40) as X = (0, 1, 1, 0, 1) with the subset A' = (7, 20, 13).

2.1.1 Key generation

In general for the communication of this algorithm we need two keys, the public key only for encryption and the private key only for decryption. In Merkle-Hellman, the keys are two knapsack vectors. The public key is a `hard` knapsack A, and the private key is an `easy`, or superincreasing, knapsack B. At first we start to fix the elements of A with the parameter n which defines the size of the message to be encrypted. B has the same length as A. Then we pick a superincreasing set $A = (a_1, \ldots, a_n)$. With the term superincreasing we mean that each element of the set is greater than the sum of all the elements before it. In next step the modulus M gets chosen that is an integer with the property M > $\sum_{i=1}^{n} a_i$. Now the multiplier W is chosen so that gcd(M, W) = 1 and 1 <= W < M. W effects the computation of the inverse element as U: $U * W$ = 1 (mod M). Then the elements of the public key B are calculated by the following formula: $B_i = (A_i * W) \, mod \, M$ for i = 1,..., n. At last we get the public key B and the private key (A, M, W) as a tuple.

2.1.2 Encryption

Before we can start to encrypt a message whose length is defined by the parameter n it can be necessary that a large message m must be divided into n-bit blocks. Now $m = (m_1, m_2, \ldots, m_n)$ is the binary message that we encrypt. We get the ciphertext c by calculating as follows: $c = b_1 * m_1 + b_2 * m_2 + ... + b_n * m_n = m * B$. If the i-th bit of m is 1 c takes the i-th element of B.

2.1.3 Decryption

For the decryption of the ciphertext c two steps must be done. The first step is to compute the equation $c' = U * c mod M = W^{-1} * c mod M$. As next step (A, c') is to solve. It is a required condition that A must be a superincreasing set for the solution.

2.2 Mathematical description

Here, we go more into the details of the mathematical description. After that, we will present an example to understand the algorithm better.

2.2.1 Key generation

To encrypt n-bit messages, first we fix a knapsack vector A which contains positive and pairwise different integers.

$$A = (a_1, a_2, ..., a_n)$$

Then a random integer M is chosen as follows

$$M > \sum_{i=1}^{n} a_i,$$

A random integer W is picked, such that gcd(W,M) = 1 (i.e. W and M are coprime). This side condition is important because coprimeness guarantees the existence of B that allows the decryption.

As next step we compute the sequence B

$$B = (b_1, b_2, ..., b_n)$$

where

$$B_i = W * a_i \; mod \; M$$

The public key is B and the private key is (A, M, W).

2.2.2 Encryption

To encrypt an n-bit message m

$$m = (m_1, m_2, \ldots, m_n)$$

where m_i is the i-th bit of the message and m_i ∈ (0, 1), compute

$$c = \sum_{i_1}^{n} m_i b_i$$

The ciphertext is c.

2.2.3 Decryption

To decrypt a ciphertext c, we firstly calculate c'as follows:

$$c' = U * c \ mod \ M = W^{-1} * c \ mod \ M$$

Here we need the inverse element U which is:

$$W * U = 1 \ mod \ M$$

We use Extended Euclidean algorithm to get U.

After that we start to solve (A, c'):

$$c' = \sum_{i=1}^{n} m_i a_i \pmod{M}$$

Now we pick the largest element of A denoted by a_l. If $a_l > c'$ then m = 0, if $a_l \leq c'$ then m = 1. Then we subtract lm_l from c' till we get 0.

2.2.4 Example

To demonstrate the functionality of the Merkle-Hellman algorithm we need first a superincreasing sequence A for the private key (A, M, W). We choose the following sequence

$$A = \{24, 34, 65, 141, 308\}.$$

Now we calculate the sum of the superincreasing sequence.

$$\sum A = 572$$

Then we choose a number M that is greater than the sum.

$$M = 900$$

Next step is to pick a number W that is in the range $[1, M)$ and is coprime to M.

$$W = 541$$

The private key contains the elements M, A and W.

For the calculation of the public key we multiply each element in A by W mod M. So we get the following sequence

$$B = \{384, 394, 65, 681, 128\}$$

because of

$$\begin{aligned}
b_1 &:= a_1 * W \ mod \ M = 24 * 541 \ mod \ 900 \equiv 384 \\
b_2 &:= a_2 * W \ mod \ M = 34 * 541 \ mod \ 900 \equiv 394 \\
b_3 &:= a_3 * W \ mod \ M = 65 * 541 \ mod \ 900 \equiv 65 \\
b_4 &:= a_4 * W \ mod \ M = 141 * 541 \ mod \ 900 \equiv 681 \\
b_5 &:= a_5 * W \ mod \ M = 308 * 541 \ mod \ 900 \equiv 128
\end{aligned}$$

Our public key is now the sequence B.

For the encryption we choose the message $m = 22$ with the condition to be in the range $[1, 32)$. The open upper bound for m depends on A which is part of the private key. This upper bound UB is defined by the number of elements of A: $UB < 2^{|A|}$. In our example A has five elements, so m can be between 0 and 31. The binary representation of m is 10110. Now we multiply each respective bit by the corresponding number in B. If the bit is 1 then we add the previous value in the iteration otherwise not.

$$\begin{aligned}
m_1 &:= 1 => 1 * 384 + 0 = 384 \\
m_2 &:= 0 => 0 * 394 + 384 = 384 \\
m_3 &:= 1 => 1 * 65 + 384 = 449 \\
m_4 &:= 1 => 1 * 681 + 449 = 1130 \\
m_5 &:= 0 => 0 * 128 + 1130 = 1130
\end{aligned}$$

We get the cipher $c = 1130$ for the message $m = 22$.

To decrypt the cipher $c = 1130$ we first have to calculate $c' = c * W^{-1} mod M$. We use the Extended Euclidean algorithm for the inverse of W. In our example the inverse of W is 361. Now we can insert the values into the equation and get the following

$$c' = c * W^{-1} \; mod \; M = 1130 * 361 \; mod \; 900 = 230$$

For the decryption we use the private key A. We decompose 230 by selecting the largest element in A which is less than or equal to 230. Then selecting the next largest element less than or equal to the difference, until the difference is 0. If it is not 0 then the text is trying to decrypt with a wrong key.

$$230 := 230 - 141 - 64 - 24 = 0$$

1	0	1	1	0	
24	34	65	141	308	
24	0	65	141	0	sum: 230

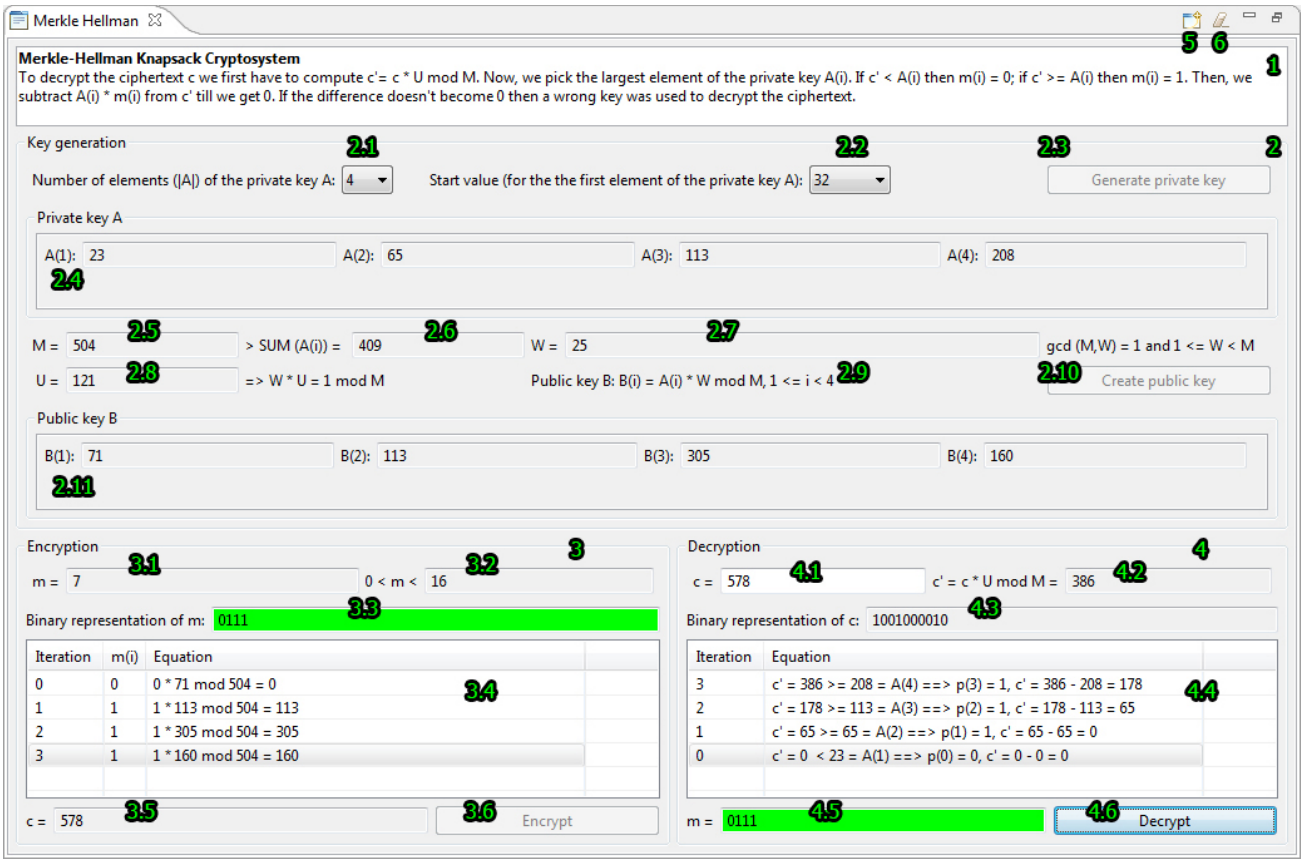
The elements which we select from our private key correspond to the 1 bits in the message 10110. When we convert from binary to decimal we get the plaintext 22.

2.3 The design of the plug-in

In this section we want to go into the design of the Merkle-Hellman plug-in. The user interface of the plug-in is created according to the GUI Guideline [JCT12] of JCrypTool. Thereby, the usability of the plug-in is important because it is our goal to guide the user through the algorithm and not simply to present the final result. To achieve our aim we select suitable presentation forms for the individual steps of the algorithm. The end result and the used lines of each table are highlighted. This action is helpful for a better overview.

For those who do not know the algorithm some sample values have been pre-assigned, so the users can start directly with the algorithm. The **Merkle-Hellman** plug-in and its components are schematically shown in figure 2.1. The numbers in this figure are related to the steps a user has to perform in the GUI. These numbers from the GUI and the according attributes are described in table 2.1 and 2.2:

- **Type**: The `SWT-Widget` element.
- **Initial state**: The initial state of the `Widget` (enabled or disabled).
- **Text**: The name of the used `Widget`.
- **Description**: Description of the `Widget`.



Merkle-Hellman Knapsack Cryptosystem
To decrypt the ciphertext c we first have to compute c'= c * U mod M. Now, we pick the largest element of the private key A(i). If c' < A(i) then m(i) = 0; if c' >= A(i) then m(i) = 1. Then, we subtract A(i) * m(i) from c' till we get 0. If the difference doesn't become 0 then a wrong key was used to decrypt the ciphertext.

Key generation

Number of elements (|A|) of the private key A: 4 Start value (for the the first element of the private key A): 32 Generate private key

Private key A

A(1): 23 A(2): 65 A(3): 113 A(4): 208

M = 504 > SUM (A(i)) = 409 W = 25 gcd (M,W) = 1 and 1 <= W < M

U = 121 => W * U = 1 mod M Public key B: B(i) = A(i) * W mod M, 1 <= i < 4 Create public key

Public key B

B(1): 71 B(2): 113 B(3): 305 B(4): 160

Encryption

m = 7 0 < m < 16

Binary representation of m: 0111

Iteration	m(i)	Equation
0	0	0 * 71 mod 504 = 0
1	1	1 * 113 mod 504 = 113
2	1	1 * 305 mod 504 = 305
3	1	1 * 160 mod 504 = 160

c = 578 Encrypt

Decryption

c = 578 c' = c * U mod M = 386

Binary representation of c: 1001000010

Iteration	Equation
3	c' = 386 >= 208 = A(4) ==> p(3) = 1, c' = 386 - 208 = 178
2	c' = 178 >= 113 = A(3) ==> p(2) = 1, c' = 178 - 113 = 65
1	c' = 65 >= 65 = A(2) ==> p(1) = 1, c' = 65 - 65 = 0
0	c' = 0 < 23 = A(1) ==> p(0) = 0, c' = 0 - 0 = 0

m = 0111 Decrypt

Figure 2.1: Merkle-Hellman plug-in and the numbers of the behavioral description

#	Type	Initial state	Text	Description
1	StyledText	disabled	see figure 2.1	Description of the algorithm (dynamic).
2	Group	enabled	Key generation	Includes entry fields for user input of private and public keys.
2.1	Combo	enabled	4	In this place the user can choose the number of elements (|A|) of the private key A.
2.2	Combo	enabled	32	In this place the user can choose the start value.
2.3	Button	enabled	Generate private key	To generate a private key. By clicking on the button, the generated values will be shown in the entry fields under the name **Private key A**.
2.4	Group	enabled	see figure 2.1	In this place the private key parts will be shown as entry fields. The values in the fields can be changed by the user. The number of elements of B is independent of the value in 2.1. By outsize the scroll bar appears.
2.5	Textfield	enabled	the value(M)	This value is either generated by clicking on the **Generate private key** button or can be entered by user. The condition is that M > SUM(A(i)).
2.6	Textfield	disabled	the value of SUM(A(i))	This value will be calculated based on the private key parts in 2.4, and cannot be changed via user entry.
2.7	Textfield	enabled	W	This value ist either generated by clicking on the **Generate private key** button or can be entered by user. The condition is that the ggT(M, W)= 1 and 1 <= W < M.
2.8	Textfield	disabled	the value of U	This value will be calculated based on the condition W * U = 1 mod M and cannot be changed via user entry
2.9	Label	enabled		This shows the condition for the public key parts (B(i)) based on the value of the number of elements (|A|) of the private key A.
2.10	Button	enabled	Generate public key	To generate a public key. By clicking on the button, the generated values will be shown in the disabled entry fields under the name **Public key B**. The group **Key generation** will be disabled and **Encryption** enabled.
2.11	Group	disabled	see figure 2.1	In this field the public key parts will be shown as disabled entry fields. The number of the parts are independent of the value in 2.1. By outsize the scroll bar appears.

Table 2.1: The behavioral description of the **Merkle-Hellman** plug-in – part 1

#	Type	Initial state	Text	Description
3	Group	disabled	Encryption	Includes entry field and the table for the encryption iterations.
3.1	Textfield	disabled	empty	This field will be enabled by clicking on the **Generate public key** button. In this place the user enters the message (m). The condition is that the value is between 0 and 2^n number of the elements (|A|) of the private key A.
3.2	Textfield	disabled	see figure 2.1	The value for the condition of the message. This will be calculated based on the number of elements (|A|) of the private key A. After the value is entered, the button **Encrypt** will be enabled.
3.3	Textfield	disabled	empty	This field visualizes the binary code of the message(m) dynamically.
3.4	Table	disabled	empty	The table consists of 3 columns. They shows the Iterations, the m(i)s and the equations of calculation of the ciphertext.
3.5	Textfield	disabled	empty	In this place the ciphertext will be shown after the calculation
3.6	Button	disabled	Encrypt	By clicking on the button the ciphertext will be calculated and the iterations of calculation will be shown in the table.
4	Group	disabled	Decryption	Includes entry field and the table for the decryption iterations.
4.1	Textfield	disabled	empty	This field will be enabled by clicking on the **Encrypt** button. The ciphertext will be taken in the field or user can enter a ciphertext.
4.2	Textfield	disabled	M	After the decryption of the message(m), in this field M gets shown.
4.3	Textfield	disabled	empty	This field visualizes the binary code of the ciphertext(c) dynamically.
4.4	Table	disabled	empty	The table consists of 2 columns. They shows the Iterations and the equations of calculation of the message.
4.5	Textfield	disabled	empty	In this field the message gets shown after the calculation
4.6	Button	disabled	Decrypt	By clicking on the button the message will be calculated and the iterations of calculation will be shown in the table.
5	Icon	enabled	see figure 2.1	**Restart** icon: By clicking on this button, all user input and results will be deleted. The initial state of the plug-in will be restored. If you move the mouse over the icon, a tooltip **Restart** will be shown.
6	Icon	enabled	see figure 2.1	**Reset** icon: By clicking on this button, the input fields for the private key will be enabled so that the user can enter new values for the private key. If you move the mouse over the icon, a tooltip **Reset** will be shown.

Table 2.2: The behavioral description of the **Merkle-Hellman** plug-in – part 2

2.4 The implementation of the plug-in

The plug-in is grouped in four packages:

- `org.jcryptool.visual.merkleHellman`
- `org.jcryptool.visual.merkleHellman.algorithm`
- `org.jcryptool.visual.merkleHellman.handlers`
- `org.jcryptool.visual.merkleHellman.views`

The package `org.jcryptool.visual.merkleHellman` contains the activator class `MerkleHellmanPlugin` which controls the whole lifecycle of the plug-in. This class is generated by the eclipse wizard. In the package `org.jcryptool.visual.merkleHellman.handlers` there are two classes `RestartHandler` and `ResetHandler` that implement the functionality in the menu bar. The class `MerkleHellmanView` is implemented in the package `org.jcryptool.visual.merkleHellman.views` which represents all the visualization of the plug-in. We would like first to describe the algorithm. The algorithm is implemented in the class `MerkleHellman` that is in the package `org.jcryptool.visual.merkleHellman.algorithm`. The class has the following attributes:

- `int dim` – number of the elements for the keys
- `BigInteger[] privateKey` – contains the private key elements
- `BigInteger[] publicKey` – contains the public key elements
- `BigInteger M` – modulus for the calculation
- `BigInteger W` – weight for the calculation
- `BigIntger U` – is the inverse of W mod M

The constructor of the class is shown in listing 2.1.

```java
public MerkleHellman(int n) {
   this.dim = n;
   this.privateKey = new BigInteger[n];
   this.publicKey = new BigInteger[n];

   for (int i = 0; i < this.dim; i++) {
      this.privateKey[i] = BigInteger.ZERO;
      this.publicKey[i] = BigInteger.ZERO;
   }
}
```

Listing 2.1: Constructor of the class `MerkleHellman`

As parameter you hand over the number of the elements for the private and public key. The private and public keys are an array of type `BigInteger` and they are initialized with zero. The class has a static method `public static BigInteger randomNumber(BigInteger min, BigInteger max)` which returns a random number in the range from `min` to `max`. The implementation of the method is shown in listing 2.2. First we check if the `max` value is smaller then `min` value. If so we switch the `max` and `min` values. If both values are equal then we return the `min` value. We determine the bit length of the range and generate a `BigInteger result` of these length. If this value is out of the range so if the value of `result` is bigger then the bit length of the `range` then we generate a new value and assign it to `result`. At last we add the `min` value and return `result`.

```java
public static BigInteger randomNumber(BigInteger min, BigInteger max) {
   if (max.compareTo(min) < 0) {
      BigInteger tmp = min;
      min = max;
      max = tmp;
```

```
    } else if (max.compareTo(min) == 0) {
        return min;
    }
    max = max.add(BigInteger.ONE);
    BigInteger range = max.subtract(min);
    int length = range.bitLength();
    BigInteger result = new BigInteger(length, new Random());
    while (result.compareTo(range) >= 0) {
        result = new BigInteger(length, new Random());
    }
    result = result.add(min);
    return result;
}
```

Listing 2.2: Generation of a random number

To make the encryption and decryption work we need a superincreasing vector for the private key. To do that we use the method `public void getSuperInc(BigInteger start)` which generates us a superincreasing vector and store it in the class attribute array `privateKey`. The implementation of this method is shown in listing 2.3. It takes as parameter a `start` value which is only allowed for non negative `BigInteger` value. The `start` value is the first element of the sequence. First we determine the bit length of the `start` value. Then we shift left the number one by the bit length of the start value and store it in the variable `t`. For the other key elements we iterate over the number of the key elements, save the sum of the key elements in the variable `sum` and generate each key element using the method `randomNumber(min, max)` which we have explained before. As `min` parameter we use the maximum of the variable `t` and the `sum` of the previous key elements incremented by one. The `max` parameter is the bit length left shifted by one and substracted by one. In each iteration we shift left the variable `t` by one. So we guarantee that the private key elements are superincreasing.

```
public void getSuperInc(BigInteger start) {
    if (start.compareTo(BigInteger.ZERO) <= 0)
        return;

    /* t is the bit length of start */
    BigInteger t = BigInteger.valueOf(start.bitLength());
    t = BigInteger.ONE.shiftLeft(t.intValue());
    this.privateKey[0] = start;
    BigInteger sum = getPrivateKeyElement(0);

    /* choose set[i] to be a (t+i)-bit number so that set[i] > sum */
    for (int i = 1; i < dim; i++) {
        this.privateKey[i] = randomNumber(sum.add(BigInteger.ONE).max(t),
                (t.shiftLeft(1).subtract(BigInteger.ONE)));
        sum = sum.add(this.getPrivateKeyElement(i));

        t = t.shiftLeft(1);
    }
}
```

Listing 2.3: Generation of a superincreasing vector

The method `public void setM()` which generates a modulus for a given private key is shown in listing 2.4. First, we determine the bit length of the last key element and store it in the variable `t`. Then we shift left a number one by this bit length to get the next bigger potency of two and store it in `t` again. This value we use to calculate the maximum of this value and the sum of the private key elements incremented by one as `min` parameter for the method `randomNumber`. As `max` parameter we shift left the value `t` subtracted by one. So we guarantee that the modulus `M` is a random number and is bigger than the sum of the private key elements.

```
public void setM() {
    /* t is the bit length of last element in super increasing sequence */
    BigInteger t = BigInteger.valueOf(this.privateKey[dim - 1].bitLength());
    t = BigInteger.ONE.shiftLeft(t.intValue());

    /* choose M to be a 2^t bit number with M > SUM(Ai) */
    this.M = randomNumber(getSum().add(BigInteger.ONE).max(t),
        (t.shiftLeft(1).subtract(BigInteger.ONE)));
}
```

Listing 2.4: Generation of a modulus for a given private key

The method `public void setW()` is calculating the weight `W` and the variable `U` which we need for the decryption. First, we check if the modulus `M` is zero. If so then we have to generate a new modulus with `setM()`. After that we generate a random number `w` in the range from two to modulus `M` minus two. The greatest common divisor is calculated with `d = w.gcd(this.M)` and stored in `d`. Now we iterate in a loop until the condition `d.compareTo(BigInteger.ONE) != 0` becomes true. In the loop we divide `w` by `d` and calculate `d` again. Finally we determine the weight `W` by dividing `w` by the gcd(w,M) with the statement `this.W = w.divide(w.gcd(this.M))`. The variable `U` is at last the multiplicative inverse of `W` modulus `M` with the statement `this.U = w.modInverse(this.M)`. The source code of the method is shown in listing 2.5.

```
public void setW() {
    if (this.M.compareTo(BigInteger.ZERO) == 0)
        setM();

    BigInteger w = randomNumber(new BigInteger("2"), this.M.subtract(new BigInteger("2")
        ));
    BigInteger d = w.gcd(this.M);

    while (d.compareTo(BigInteger.ONE) != 0) {
        w = w.divide(d);
        d = w.gcd(this.M);
    }

    this.W = w.divide(w.gcd(this.M));
    this.U = w.modInverse(this.M);
}
```

Listing 2.5: Generation of the weight

The method `public void createPublicKeys()` is used to generate the public key. Each public key element is calculated in which each private key element is multiplied by the weight `W` modulus `M`. The listing is shown in 2.6.

```
public void createPublicKeys() {
    for (int i = 0; i < dim; i++) {
        setPublicKeyElement(i, this.W.multiply(this.privateKey[i]).mod(this.M));
    }
}
```

Listing 2.6: Generation of the public key

For the encryption we use the method `public BigInteger encrypt(BigInteger plain)` that is shown in listing 2.7. The method takes a plaintext of type `BigInteger` and returns the corresponding ciphertext `chiffre`. First, we initialize `chiffre` with zeros. We store the number of key elements in the variable `n`. Then, we use another variable `s`, initialize it with one and shift it left by the variable `n` to multiply it with `n * 2`.

Now we iterate in a loop to calculate for each bit in the plaintext to add the corresponding public key elements to the `chiffre` text. We add the public key element only if the bit in the plaintext is one otherwise we skip the addition. To determine if the bit in the plaintext is one we use the `add` method with the variable `s` which we shift right in every iteration.

```
public BigInteger encrypt(BigInteger plain) {
    BigInteger chiffre = BigInteger.ZERO;
    int n = this.dim - 1;
    BigInteger s = BigInteger.ONE;
    s = s.shiftLeft(n);

    boolean b;
    for (int i = 0; i < this.dim; i++) {
        b = false;

        if (plain.and(s).compareTo(BigInteger.ZERO) != 0)
            b = true;

        if (b) {
            chiffre = chiffre.add(this.getPublicKeyElement(i));
        }
        s = s.shiftRight(1);
    }
    return chiffre;
}
```

Listing 2.7: Encryption of a plaintext

In the package `org.jcryptool.visual.merkleHellman.views` we have the class `MerkleHellmanView` which describes all the graphical interface and the user interaction of the plug-in. There are two drop-down menus where you can choose the number of key elements and a start value as you can see in figure 2.1. When the plug-in is loaded by default a private key and all needed components to perform the algorithm are generated. For that the method `private void generatePrivateKey(int numberOfElements, int startValue)` is used that is shown in 2.8. As start value there will be a random number generated between [$\frac{start}{2}$ and start] value which can be chosen in the drop-down menu.

```
private void generatePrivateKey(int numberOfElements, int startValue) {
    BigInteger startVal = BigInteger.valueOf(startValue);
    privKey = new MerkleHellman(numberOfElements);
    privKey.createPrivateKeys(MerkleHellman.randomNumber(startVal.shiftRight(1),
        startVal));

    for (int i = 0; i < privateKeyFields.size(); i++) {
        privateKeyFields.get(i).setText(String.valueOf(privKey.getPrivateKeyElement(i)));
    }

    textM.setText(String.valueOf(privKey.getM()));
    textSumA.setText(String.valueOf(privKey.getSum()));
    textU.setText(String.valueOf(privKey.getU()));
    textW.setText(String.valueOf(privKey.getW()));

}
```

Listing 2.8: Generation of the sample private key

When you change the values of the **Number of private key elements** or the **start** drop-down menu new private keys are dynamically generated. You can also click on the button **Generate private key** to generate new private keys. If you use the automatically generated way all conditions are fulfilled and you can click the button **Create public key**. When you change some values - all changeable fields are enabled and editable - then all conditions for the algorithm have to be checked while you are clicking the button **Create public key**. The calculation is only executed when not all fields are empty. There are four conditions that have to be checked:

1. Private key values have been changed and **W** or **M** values have not been changed.
2. Private key values have not been changed and **W** or **M** values have been changed.
3. Private key values have been changed and **W** or **M** values have been changed.
4. No values have been changed.

In the first case only the private key values have to be updated. For that we use the method `public void updatePrivateKey(BigInteger[] keys)` of the class `MerkleHellman` which assigns the new entered key to the algorithm and calculates the attribute `M` and `W`. The implementation is shown in listing 2.9. If the new entered private key values are not a superincreasing vector then a warning dialog appears. The user can accept the values by clicking the **Ok** button. In this case the algorithm might not work correctly because this condition is necessary for the uniqueness of the decryption

```java
public void updatePrivateKey(BigInteger[] keys) {
   this.privateKey = keys;
   this.setM();
   this.setW();
}
```

Listing 2.9: Update private key

In the second case only the values for `M` or `W` have been changed. In this case three things have to be checked. First, `M` has to be bigger as the sum of the private keys. If not an error dialog is shown for the user. Secondly, `W` has to be smaller than `M`. If not again an error dialog is shown. And at last the greatest common divisor of `M` and `W` has to be one. If none of these three cases are violated the new values for `M` or `W` are set for the algorithm.

In the third case both conditions private key and `M` or `W` are violated. Now we check if the private key is a super increasing vector, if `M` is bigger as the sum of the private key, `W` is smaller then `M` and if the the greatest common divisor is equal to one. If none of these conditions are violated then we update the private key values and assign the new values for `M` and `W` for the algorithm.

In the fourth case no values are changed so no extra handling is needed. We create the public key by executing the method `public void createPublicKeys()` in class `MerkleHellman`. The implementation is shown in listing 2.6. The public key elements are created by multiplying the corresponding private key element with the weight `W` modulus `M`.

Now that the private and public key are created, the user can enter a message into the corresponding field. While the user enters a value for the message the binary representation of the value is dynamically calculated and displayed below. For the input message there are only values between 0 to $2^{number\ of\ key\ elements}$ allowed. To guarantee this we use a `ModifyListener` which is shown in listing 2.10. First we instanciate a `BigInteger` with `BigInteger a = new BigInteger(message.getText())` and store the binary representation in a variable `bitRepresentation` with `String bitRepresentation = a.toString(2)`. Now we check if the length of `bitRepresentation` is smaller than `numberOfElements`. If so we have to add leading zeros to the bit representation. We determine the number of leading zeros with `int counter = numberOfElements - bitRepresentation.length()` and iterate over the variable `counter` to append the leading zeros with `sb.append("0")` to the resulting `StringBuilder sb`. Otherwise when the `bitRepresentation` is not smaller than `numberOfElements` we don't need leading zeros and can append the `bitRepresentation` directly to the `StringBuilder sb`. When a valid message is entered the button **Encrypt** will be activated.

```java
textM_encryption.addModifyListener(new ModifyListener() {
   @Override
   public void modifyText(ModifyEvent e) {
      if (e.getSource() instanceof Text) {
         Text message = (Text) e.getSource();
         if (message.getText().compareTo("") != 0) {
            int numberOfElements = Integer.parseInt(comboKeyElements.getText());
            BigInteger a = new BigInteger(message.getText());
            String bitRepresentation = a.toString(2);
            StringBuilder sb = new StringBuilder();

            if (bitRepresentation.length() <= numberOfElements) {
               int counter = numberOfElements — bitRepresentation.length();
               for (int i = 0; i < counter; i++) {
                  sb.append("0");
               }
```

```
                sb.append(bitRepresentation);
            }
            MerkleHellmanView.this.textBinaryM.setText(sb.toString());
            btnEncrypt.setEnabled(true);
        } else {
            MerkleHellmanView.this.textBinaryM.setText("");
            btnEncrypt.setEnabled(false);
        }
    }
  }
});
```

Listing 2.10: `ModifyListener` for the message

The user can start to encrypt by clicking the button `Encryption`. The implementation of the method is shown in listing 2.11. We iterate over the number of the key elements and add the calculation details into a table. The table consists of three columns. The first column represents the number of the iteration, the second column represents whether the bit of the message was used, and the third column represents the calculation details of the current iteration. The sum of all bits where the bit value is one is stored in the variable `String result` with the statement `String result = String.valueOf(privKey.encrypt(new BigInteger(textM_encryption.getText())))`. The encrypted message is also entered as input for the decryption. The private key elements, the value `M` and `W` are also editable for the decryption to have the possibility to perform the decryption with different keys.

```
@Override
public void widgetSelected(SelectionEvent e) {
   int numberOfElement = Integer.parseInt(comboKeyElements.getText());
   String bitRepresentation = textBinaryM.getText();

   for (int i = 0; i < numberOfElement; i++) {
      StringBuilder sb = new StringBuilder();
      sb.append(String.valueOf(bitRepresentation.charAt(i)) + " * " + privKey.
          getPublicKeyElement(i)
            + " mod " + privKey.getM() + " = ");
      sb.append(new BigInteger(String.valueOf(bitRepresentation.charAt(i))).multiply(
          privKey
            .getPublicKeyElement(i).mod(privKey.getM())));

      TableItem tmp = new TableItem(tableEncrypt, SWT.BORDER);
      tmp.setText(0, String.valueOf(i));
      tmp.setText(1, String.valueOf(bitRepresentation.charAt(i)));
      tmp.setText(2, sb.toString());
   }

   String result = String.valueOf(privKey.encrypt(new BigInteger(textM_encryption.
       getText())));

   tableEncrypt.setSelection(numberOfElement - 1);
   tableEncrypt.showSelection();

   textC_encryption.setText(result);
   textC_encryption.setEnabled(true);
   textC_decryption.setText(result);
   textC_decryption.setEnabled(true);
   textM_encryption.setEditable(false);
   btnEncrypt.setEnabled(false);
   btnDecrypt.setEnabled(true);

   styledTextDescription.setText(Messages.MerkleHellmanView_0000 + Messages.
       MerkleHellmanView_000);
```

```
    styledTextDescription.setStyleRange(header);
    textC_decryption.setFocus();
    textC_decryption.setSelection(textC_decryption.getText().length());
}
```

Listing 2.11: Encryption of a message

The implementation of the decryption is shown in listing 2.12. Because the private key input fields are editable the same conditions as for the encryption must to be checked. For the decryption we need to calculate $c' = c * U \ mod \ M$ and save the value in the variable `BigInteger cc`. Now we iterate over the number of the key elements starting from the last key element to the first. In the loop we compare the variable `cc` with the private key element stored in variable `tmpPrivateKeyElement`. If `cc` is equal or bigger than `tmpPrivateKeyElement` then the private key value `tmpPrivateKeyElement` is eligible for the calculation and we mark it in the table with a binary one. We subtract this value from `cc` for the next iteration. Otherwise we mark the entry with a binary zero. No subtraction is needed When we exit the loop we check if the binary string for encryption and decryption are equal. If so we color the field with the binary string green otherwise in red. After finishing the decryption it is possible for the user to enter another ciphertext and run the decryption again.

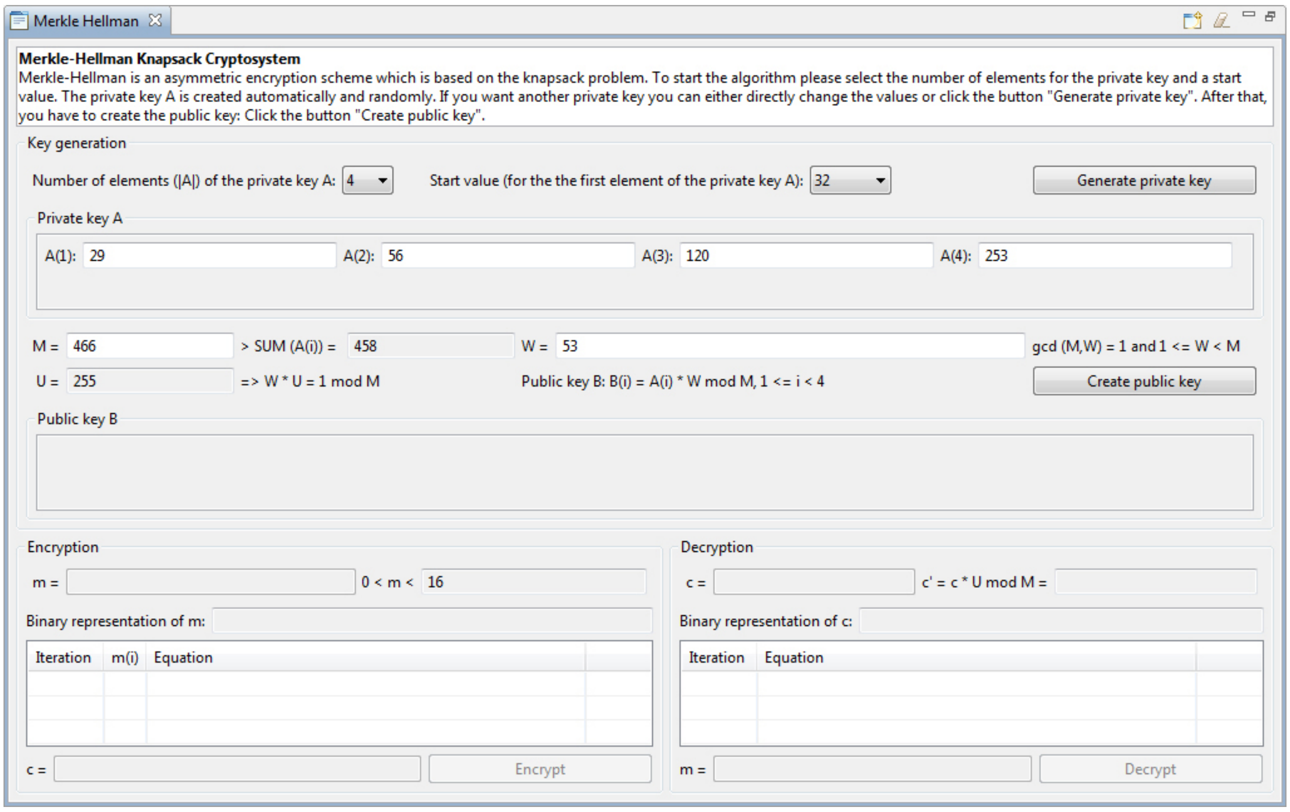
```
@Override
public void widgetSelected(SelectionEvent e) {
    ...
    BigInteger c = new BigInteger(textC_decryption.getText());
    BigInteger U = privKey.getU();
    BigInteger M = privKey.getM();
    BigInteger cc = c.multiply(U).mod(M);
    textCC.setText(String.valueOf(cc));
    int numberOfElement = Integer.parseInt(comboKeyElements.getText());
    BigInteger tmpCC = cc;
    StringBuilder binResult = new StringBuilder();
    for (int i = numberOfElement - 1; i >= 0; i--) {
        StringBuilder sb = new StringBuilder();
        BigInteger tmpPrivateKeyElement = privKey.getPrivateKeyElement(i);
        if (tmpCC.compareTo(tmpPrivateKeyElement) >= 0) {
            sb.append("c' = " + tmpCC + " >= " + tmpPrivateKeyElement + " = A(" + (i + 1)
                + ")");
            sb.append(" ==> p(" + i + ") = 1, ");
            sb.append("c' = " + tmpCC + " - " + tmpPrivateKeyElement + " = " + (tmpCC =
                tmpCC.subtract(tmpPrivateKeyElement)));
            binResult.insert(0, "1");
        } else {
            sb.append("c' = " + tmpCC + "  < " + tmpPrivateKeyElement + " = A(" + (i + 1)
                + ")");
            sb.append(" ==> p(" + i + ") = 0, ");
            sb.append("c' = " + tmpCC + " - 0" + " = " + (tmpCC = tmpCC.subtract(
                BigInteger.ZERO)));
            binResult.insert(0, "0");
        }
        TableItem tmp = new TableItem(tableDecrypt, SWT.BORDER);
        tmp.setText(0, String.valueOf(i));
        tmp.setText(1, sb.toString());
    }
    tableDecrypt.setSelection(numberOfElement - 1);
    textBinary_decrypted.setText(binResult.toString());
    textBinary_decrypted.setEnabled(true);
    if (textBinaryM.getText().compareTo(textBinary_decrypted.getText()) == 0) {
        textBinaryM.setBackground(new Color(null, new RGB(0, 255, 0)));
        textBinary_decrypted.setBackground(new Color(null, new RGB(0, 255, 0)));
        MessageDialog.openInformation(null, Messages.MerkleHellmanView_18, Messages.
            MerkleHellmanView_19
            + textBinaryM.getText() + " = " + binResult);
    } else {
        textBinaryM.setBackground(new Color(null, new RGB(255, 0, 0)));
        textBinary_decrypted.setBackground(new Color(null, new RGB(255, 0, 0)));
        MessageDialog.openError(null, Messages.MerkleHellmanView_18, Messages.
            MerkleHellmanView_30
            + textBinaryM.getText() + " = " + binResult);
    }
    textC_decryption.setFocus();
    btnDecrypt.setFocus();
}
```

Listing 2.12: Decryption of a ciphertext

The user can click on the button **Restart** or **Reset** in the menu bar at any time. The **Restart** button resets the view of the plug-in to the initial state so that the user can start from scratch. New private key values will be created. The **Reset** button resets the tables and the public keys, enables the input fields for the private key to give the user the possibility to change the values without generating new values.

2.5 The functionality of the plug-in

In this section we want to describe the functionality of the plug-in **Merkle-Hellman**. The plug-in can be started via the menu **Visuals** or via the crypto explorer tab **Visuals**.



Merkle Hellman

Merkle-Hellman Knapsack Cryptosystem
Merkle-Hellman is an asymmetric encryption scheme which is based on the knapsack problem. To start the algorithm please select the number of elements for the private key and a start value. The private key A is created automatically and randomly. If you want another private key you can either directly change the values or click the button "Generate private key". After that, you have to create the public key: Click the button "Create public key".

Key generation

Number of elements (|A|) of the private key A: 4 Start value (for the the first element of the private key A): 32

Private key A

A(1): 29 A(2): 56 A(3): 120 A(4): 253

M = 466 > SUM (A(i)) = 458 W = 53 gcd (M,W) = 1 and 1 <= W < M

U = 255 => W * U = 1 mod M Public key B: B(i) = A(i) * W mod M, 1 <= i < 4

Public key B

Encryption

m = 0 < m < 16

Binary representation of m:

Iteration	m(i)	Equation

c =

Decryption

c = c' = c * U mod M =

Binary representation of c:

Iteration	Equation

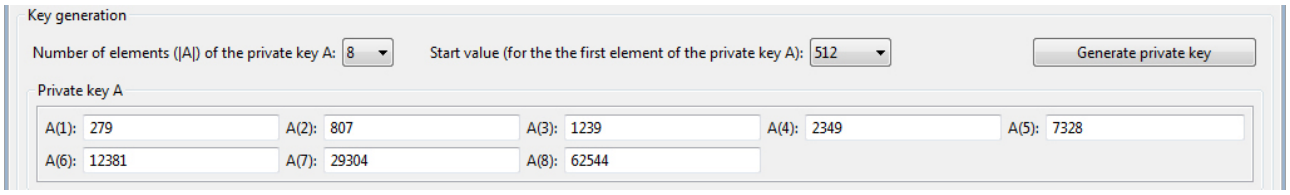
m =

Figure 2.2: Merkle-Hellman plug-in

The plug-in consists of a description field and three basic areas: **Key generation**, **Encryption** and **Decryption**. In the description field the corresponding statements about the algorithm are displayed dynamically, depending on which step of the algorithm the user is currently performing.

2.5.1 Key generation

This section summarizes the actions that are required to generate a private key and a public key.
First of all the number of the elements for the private key and the initial start value must be selected from the drop-down boxes. By default, the number of elements for the private key is 4 and the start value is 32. The start value is needed for the generation of the first element of the private key $A(1)$. It will be used a random number in the range of $[startvalue/2 - startvalue]$. The number of the private key elements will be dynamically created depending on user selection.



Key generation

Number of elements (|A|) of the private key A: 8 Start value (for the the first element of the private key A): 512

Private key A

A(1): 279 A(2): 807 A(3): 1239 A(4): 2349 A(5): 7328
A(6): 12381 A(7): 29304 A(8): 62544

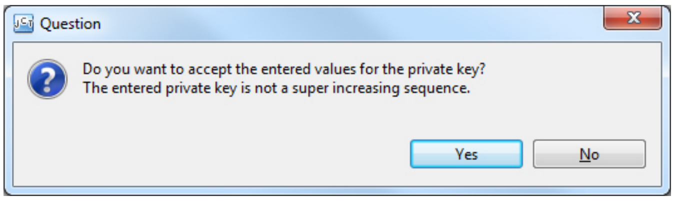
Figure 2.3: Key generation

On this selection, the user can either use the automatically and randomly generated sample values for $A(i)$, M and W, or he can manually enter new values, or he can generate new values for the private key by clicking the button **Generate private key**.

Figure 2.4: Button **Generate private key**

If the entered private key elements $A(i)$s are not a superincreasing vector and the user is clicking on the button **Create public key**, the user gets displayed the following note message:

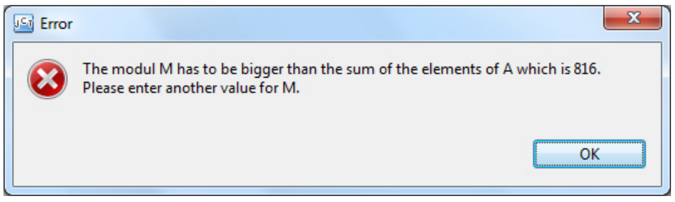
Figure 2.5: Button **Create public key**



Do you want to accept the entered values for the private key?
The entered private key is not a super increasing sequence.

Figure 2.6: Button **Super increasing vector**

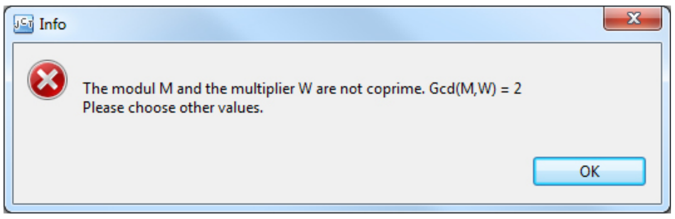
There is also a check for the parameters M and W when the user is clicking the button **Create public key**. Depending on which condition is violated different information dialogs pop up.
For $M \leq \sum A(i)$ the following information is shown:



The modul M has to be bigger than the sum of the elements of A which is 816.
Please enter another value for M.

Figure 2.7: Button **Warning message for** M

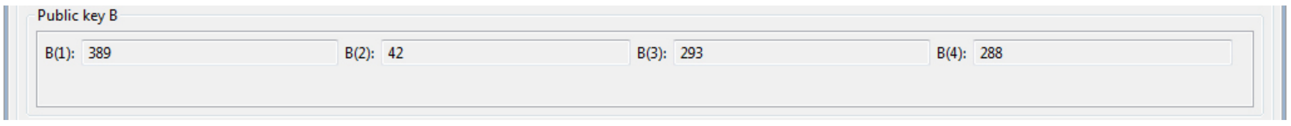
For $ggT(M,W) \neq 1$ the following information is shown:



The modul M and the multiplier W are not coprime. Gcd(M,W) = 2
Please choose other values.

Figure 2.8: Button **Warning message for** W

Now all the conditions are fulfilled, the public key will be calculated and the elements $B(i)$s are shown in the **Public key B** area.


Public key B

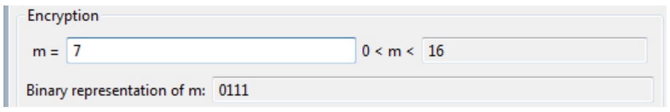
B(1): 389 B(2): 42 B(3): 293 B(4): 288

Figure 2.9: Button **Public keys**

2.5.2 Encryption

After the keys have been generated, the user can enter a message m in the corresponding field. As you type the message, the binary representation is dynamically calculated in addition and shown below in the field **binary representation of m**. The button **Encrypt** will be activated once a valid value m is entered.

Figure 2.10: Button Encrypt

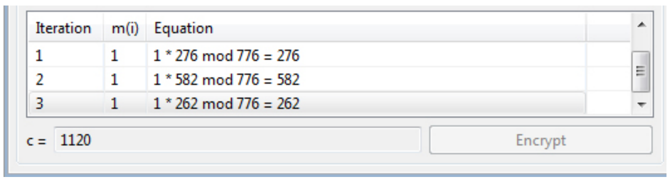

Encryption

m = 7 0 < m < 16

Binary representation of m: 0111

Figure 2.11: Button **Binary representation of m**

When you click on the button **Encryption** the message parts are calculated and the iteration of the calculations will be shown in the table below.
The ciphertext is also calculated, displayed and entered for the decryption.



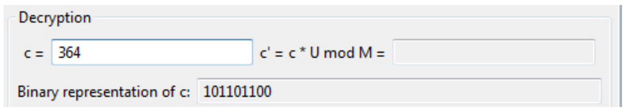
Iteration	m(i)	Equation
1	1	1 * 276 mod 776 = 276
2	1	1 * 582 mod 776 = 582
3	1	1 * 262 mod 776 = 262

c = 1120 Encrypt

Figure 2.12: Iteration of the encryption

2.5.3 Decryption

For the decryption you can enter any values for c. The binary representation of the entered value is dynamically calculated and shown again.


Decryption

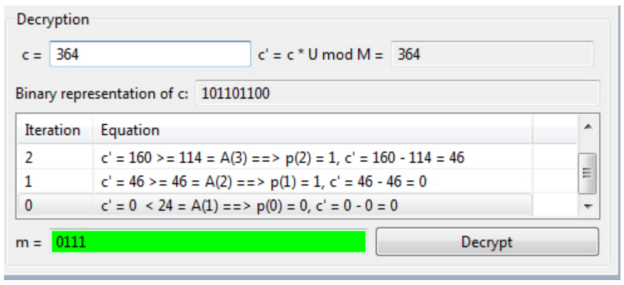
c = 364 c' = c * U mod M =

Binary representation of c: 101101100

Figure 2.13: Decryption

By clicking the button **Decryption** the ciphertext will be decrypted and its iterations of the calculation will be shown in the table. After the decryption the user has the possibility to decrypt other messages with the same key.

Figure 2.14: Button **Decrypt**


Decryption

c = 364   c' = c * U mod M = 364

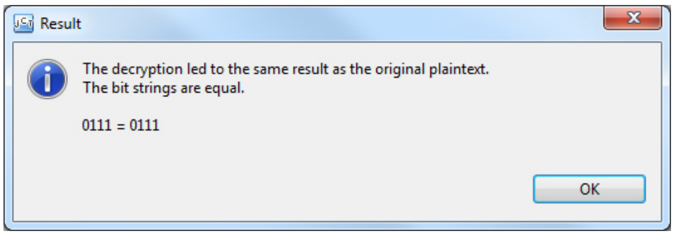
Binary representation of c: 101101100

Iteration	Equation
2	c' = 160 >= 114 = A(3) ==> p(2) = 1, c' = 160 - 114 = 46
1	c' = 46 >= 46 = A(2) ==> p(1) = 1, c' = 46 - 46 = 0
0	c' = 0 < 24 = A(1) ==> p(0) = 0, c' = 0 - 0 = 0

m = 0111

Figure 2.15: Iteration of the decryption

In case that the decryption was successful the following message will be displayed and both the plaintext and the ciphertext will be colored green:

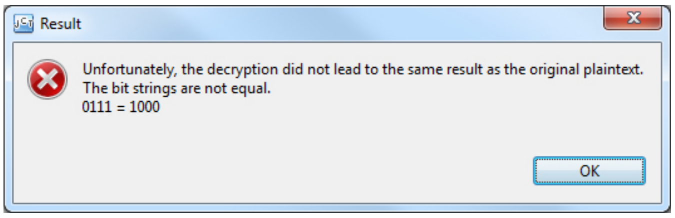

Result

The decryption led to the same result as the original plaintext.
The bit strings are equal.

0111 = 0111

Figure 2.16: Decryption successful

Otherwise, the plaintext and the ciphertext are highlighted in red color and the following message appears:


Result

Unfortunately, the decryption did not lead to the same result as the original plaintext.
The bit strings are not equal.
0111 = 1000

Figure 2.17: Decryption failed

2.5.4 The menu bar

- A click on the icon **Restart** in the menu bar brings the plug-in back to its initial state, and you can start the algorithm from scratch.
- You can change the values of the private key by clicking on the icon **Reset** in the menu bar. Clicking this icon deletes all entries in the plug-in besides the private key.

Figure 2.18: Button **Restart**

Figure 2.19: Button **Reset**

3 Hash Sensitivity plug-in

Cryptographic hash functions play an important role in modern cryptography. They are used in data processing, digital signatures, message authentication codes (MACs), building of block ciphers etc.

In this chapter we go into the **Hash Sensitivity**. At first we describe what a cryptographic hash function is, as next step we illustrate and realize three hash function families which are Message Digest Algorithm (MD in the variant MD2, MD4 and MD5), Secure Hash Algorithm (SHA, in the variants SHA-1 and SHA-2), and RACE Integrity Primitives Evaluation Message Digest (RIPEMD, in the variant RIPEMD-160).

Under a cryptographic hash function we mean a mapping

$$h\colon \Sigma^* \to \Sigma^n,\ n \in \mathbb{N} \tag{3.1}$$

That means that hash functions map strings of any size to strings of fixed size. They are never injective [Buc08].

Hash functions receive a message as input and give back an output as hash value called as message digest.
In the following sections we want to deal with the topic hash Sensitivity. We present some popular hash functions only briefly because our goal is to realize them as plug-ins. For more details about these we recommend the literature to which we refer.

3.1 Message Digest (MD)

The family of the Message Digest (MD) algorithms contains MD2, MD4, and MD5. Ronald L. Rivest was the developer of these cryptographic hash functions. All these algorithms get a message of any length as input and return a 128-bit hash value as output called also message digest. These algorithms are structurally similar but they differ in construction. While MD2 is developed for 8-bit computers, MD4 and MD5 are designed for 32-bit computers. MD6 is the current version, but not offered in the Bouncy Castle library yet. Now we take a look at these three hash functions.

3.1.1 Message Digest2

The algorithm Message Digest2 (MD2) is the first cryptographic hash function of the series of the message digest group which was designed by Ronald L. Rivest in 1989 [RSA15]. MD2 was broken in 1995 [NR95]. The digest length is 128 bit (16 byte). MD2 was designed for 8-bit computers. Its specification is defined in RFC 1319. As a first step the message has to be extended that its block length is dividable by 16 byte (128 bit), then a checksum with 16 byte gets added to the message, and finally the hash value is calculated on the complete message.

3.1.2 Message Digest4

The algorithm Message Digest4 (MD4) is the successor version of MD2 and the predecessor of MD5. As in MD2 Ronald L. Rivest is also the developer of this hash function which is published in 1990 [Riv90]. MD4 is broken in 1995 [RSA15]. The hash value (output) is 128 bit long (16 byte). MD4 is designed for 32-bit computers. Its specification is described in RFC 1320. Hash functions such as MD5, SHA1 and SHA2 use the design principles of MD4. The message is extended with a leading 1, followed by 0s so that the length of the message is congruent to 448 modulo 512. Now, bits between 1 and 512 are appended to the message and finally the size of the original message is concatenated to the last 64 bit.

3.1.3 Message Digest5

The algorithm Message Digest5 (MD5) was designed by Ronald L. Rivest in 1991 [Riv92]. It is the successor of MD4 and MD2. MD6 is its successor. MD5 was broken in 1996 [Dob96]. This hash function returns also a 128-bit hash value as output as MD2 and MD4. MD5 is also designed for 32-bit computers. The specification of this algorithm is shown in RFC 1321. The following description of the MD5 algorithm is from [MD515].

Description of the MD5 algorithm

MD5 processes a variable-length message into a fixed-length output of 128 bit. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64}.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted A, B, C, and D. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed rounds; each round is composed of 16 similar operations based on a non-linear function F, modular addition, and left rotation. There are four possible functions F; a different one is used in each round:

- $F(B,C,D) = (B \wedge C) \vee (\neg B \wedge D)$ with $(0 \leq i \leq 15)$
- $G(B,C,D) = (B \wedge D) \vee (C \wedge \neg D)$ with $(16 \leq i \leq 31)$
- $H(B,C,D) = B \oplus C \oplus D$ with $(32 \leq i \leq 47)$
- $I(B,C,D) = C \oplus (B \vee \neg D)$ with $(48 \leq i \leq 63)$

$\oplus, \wedge, \vee, \neg$ denote the XOR, AND, OR and NOT operations respectively.

3.2 Secure Hash Algorithm (SHA)

The series of the Secure Hash Algorithm (SHA) are cryptographic hash functions including SHA-0, SHA-1, SHA-2, and lastly SHA-3. Beside SHA-3, all of these have been developed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology NIST. The history of SHAs began in 1993. The SHA algorithms have different structures. Especially SHA-3 (Keccak) has a completely different design as SHA-0, SHA-1 and SHA-2. Keccak was nominated as SHA-3, but NIST didn't publish SHA-3 as a final standard yet. The SHAs get a message as input and give back a hash value as output called message digest. The size of the message digests varies between 160 bit and 512 bit, based on the algorithm. They are used in digital signatures, in message authentification codes etc. Now we take a look at the SHA versions.

3.2.1 Secure Hash Algorithm SHA/SHA-0

The Secure Hash Algorithm SHA/SHA-0 is the original version of the SHA series which was developed by the NSA and published by NIST in 1993 [NI12] under the name SHA. It generates a 160-bit (20-byte) hash value as output. This algorithm was broken in 1998 [FC93]. A construction fault that was found after the publishing led to a correction of this algorithm, and 1995 this correction was renamed as SHA-1.

3.2.2 Secure Hash Algorithm SHA-1

SHA-1 is the improved version of SHA/SHA-0. SHA-1 was published by NIST in 1995 [NI12]. The SHA-1 algorithm was broken in 2005 [XW05]. It also generates a 160-bit (20-byte) hash value as its result. The hash value is a hexadecimal number with a length of 40 hex digits. SHA-1's design is similar to MD5. SHA-1 is still the most popular of the SHA hash functions. The specification of this cryptographic hash function is described in RFC 3174. The difference between SHA-0 and SHA-1 lies in the left rotation of the compression function.

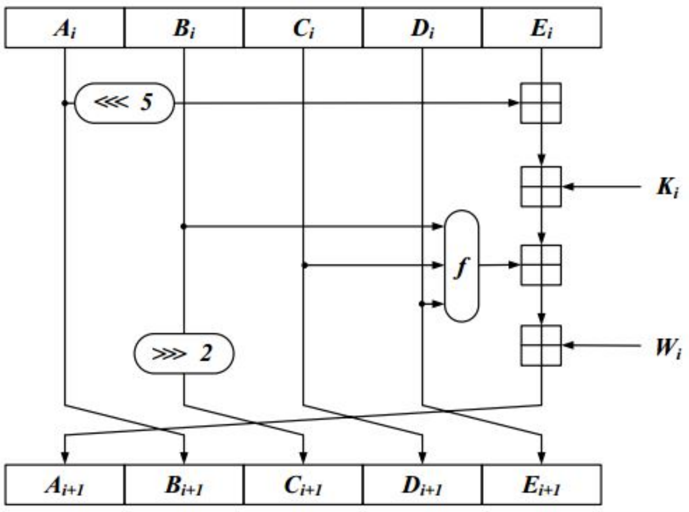


Figure 3.1: Schematic overview of a SHA-0/SHA-1 round from [WP08]

In the figure 3.1 above we see a typical iteration within SHA-1. In the following the notation of SHA-0/SHA-1 is shown.

- A, B, C, D and E are 32-bit words of the state;
- F is a nonlinear function that varies;
- $\lll_n$ denotes a left bit rotation by n places;
- $\ggg_n$ denotes a right bit rotation by n places;
- n varies for each operation;
- W_t is the expanded message word of round t;
- K_t is the round constant of round t;
- $\boxplus$ Addition denotes addition modulo 2^{32}.

The following functions and constants are used in the SHA-1 [SHA01]:

A sequence of logical functions f(0), f(1),..., f(79) is used in SHA-1. Each f(t), $0 \leq t \leq 79$, operates on three 32-bit words B, C, D and produces a 32-bit word as output. f(t;B,C,D) is defined as follows: for words B, C, D,

- $f(t;B,C,D) = (B \wedge C) \vee ((\neg B) \wedge D) \ \ (0 \leq t \leq 19)$
- $f(t;B,C,D) = B \oplus C \oplus D \ \ (20 \leq t \leq 39)$
- $f(t;B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) \ \ (40 \leq t \leq 59)$
- $f(t;B,C,D) = B \oplus C \oplus D \ \ (60 \leq t \leq 79)$

A sequence of constant words K(0), K(1), ... , K(79) is used in the SHA-1. In hex these are given by

- $K(t) = 5A827999 \ \ (0 \leq t \leq 19)$
- $K(t) = 6ED9EBA1 \ \ (20 \leq t \leq 39)$
- $K(t) = 8F1BBCDC \ \ (40 \leq t \leq 59)$
- $K(t) = CA62C1D6 \ \ (60 \leq t \leq 79)$

3.2.3 Secure Hash Algorithm SHA-2

The Secure Hash Algorithm SHA-2 is the successor of SHA-1 and was published by NIST in 2001 [WP08]. SHA-2 contains four cryptographic hash functions named as 224, 256, 384 and 512. That means that each variant has different block sizes and messages digest lengths in bits. There is also a difference in word size: SHA-256 uses for example 32-bit words and SHA-512 uses 64-bit words. The structure of these hash functions is indeed equally but different numbers of shift, rounds and extra constants are used by them for their computation. The family of SHA-2 is still an interesting object of cryptanalysis research because until now this hash function is still unbroken. (NIST) nominated SHA-3 as an alternative cryptographic hash function as successor of SHA-2 if it eventually should be broken but it doesn't mean that SHA-3 replaces SHA-2 today.

Now we take a brief look at the versions of SHA-2/256 and SHA-2/512, also called SHA-256 and SHA-512.

The Secure Hash Algorithm SHA-2/256 produces a 256-bit (32-byte) hash value as result. The block size is 512-bits long and this algorithm works with 32-bit words. This hash function uses six logical functions and 64 constant words each of 32-bit. The function definitions for SHA-2/256 are in [WP08]:

$$W_i - \begin{cases} M_i, & \text{if } 0 \le i \le 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & \text{if } 16 \le i \le 63 \end{cases}$$

including

- $\Sigma_0(x) - \ggg 2 \oplus x \ggg 13 \oplus x \ggg 22$
- $\Sigma_1(x) - \ggg 6 \oplus x \ggg 11 \oplus x \ggg 25$
- $\sigma_0(x) - \ggg 7 \oplus x \ggg 18 \oplus x \gg 3$
- $\sigma_1(x) - \ggg 17 \oplus x \ggg 19 \oplus x \gg 20$
- $f_{if}(b,c,d) - b \wedge c \oplus \neg b \wedge d$
- $f_{maj}(b,c,d) - b \wedge c \oplus b \wedge d \oplus c \wedge d$

The Secure Hash Algorithm SHA-2/512 generates a 512-bit (64-byte) hash value as output. The block size is 1024-bits long and this hash function works with 64-bit words. This algorithm also uses six logical functions and 80 constant words each of 64-bit. The function definitions for SHA-2/512 are in [SHA12]:

$$W_i - \begin{cases} M_i, & \text{if } 0 \le i \le 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & \text{if } 16 \le i \le 79 \end{cases}$$

including

- $\Sigma_0(x) - \ggg 28 \oplus x \ggg 34 \oplus x \ggg 39$
- $\Sigma_1(x) - \ggg 14 \oplus x \ggg 18 \oplus x \ggg 41$
- $\sigma_0(x) - \ggg 1 \oplus x \ggg 8 \oplus x \gg 7$
- $\sigma_1(x) - \ggg 19 \oplus x \ggg 61 \oplus x \gg 6$
- $f_{if}(b,c,d) - b \wedge c \oplus \neg b \wedge d$
- $f_{maj}(b,c,d) - b \wedge c \oplus b \wedge d \oplus c \wedge d$

In figure 3.2 below a classical iteration within SHA-2 is depicted.

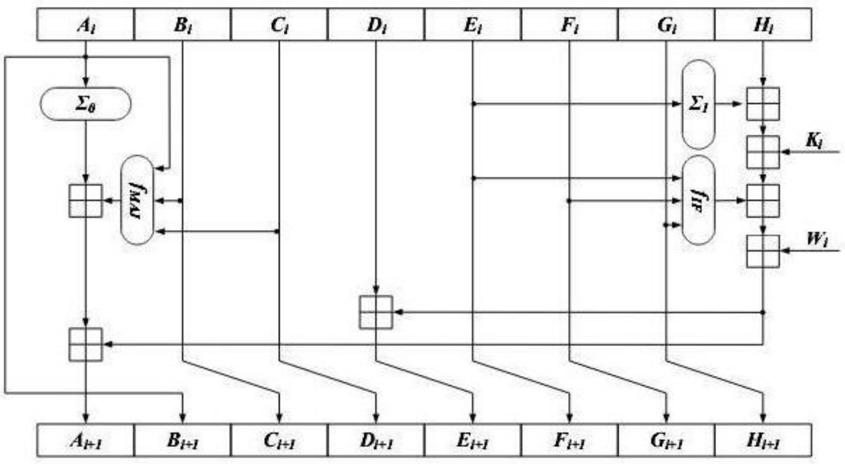


Figure 3.2: Schematic overview of a SHA-2 round from [WP08]

Here is the notation used in the SHA-2 descriptions.

- A, B, C, D, E, F, G and H are 32-bit/64-bit[1] words of the state;
- F is a nonlinear function that varies;
- n varies for each operation;
- W_t is the expanded message word of round t;
- K_t is the round constant of round t;
- $\boxplus$ Addition denotes addition modulo $2^{32}/2^{64}$ [1]
- $\oplus$ denotes bitwise XOR
- $\wedge$ denotes bitwise AND
- $\vee$ denotes bitwise OR
- $\neg$ denotes bitwise complement
- $\ggg_n$ denotes right shift by n bits
- $\gg_n$ denotes right rotation by n bits

3.3 RACE Integrity Primitives Evaluation Message Digest (RIPEMD-160)

The RIPEMD-160 cryptographic hash function was developed by Hans Dobbertin, Antoon Bosselaers and Bart Preneel [HD96] und published in 1996. It is the successor of the original version of RIPEMD. The family of the RIPEMD algorithms contains all in all four series with different bit sizes, namely RIPEMD-128, RIPEMD-160, RIPEMD-256, and finally RIPEMD-320. RIPEMD-160 is the most popular version of this series. RIPEMD-160 is optimized for 32-bit processors. This algorithm works 512-bit input message blocks and generates a 160-bit hash value as outcome. The design principles of MD4 are used in RIPEMD-160 and the performance of SHA-1 is similar to RIPEMD-160. That's why RIPEMD-160 is seen as a subfamily of the MD-SHA family. RIPEMD-160 is also still unbroken like SHA-2.

A schematic overview of a RIPEMD-160 round is depicted in the figure 3.3.

[1] It depends on the version, if SHA-256 then 32 bit, and if SHA-512 then 64 bit.

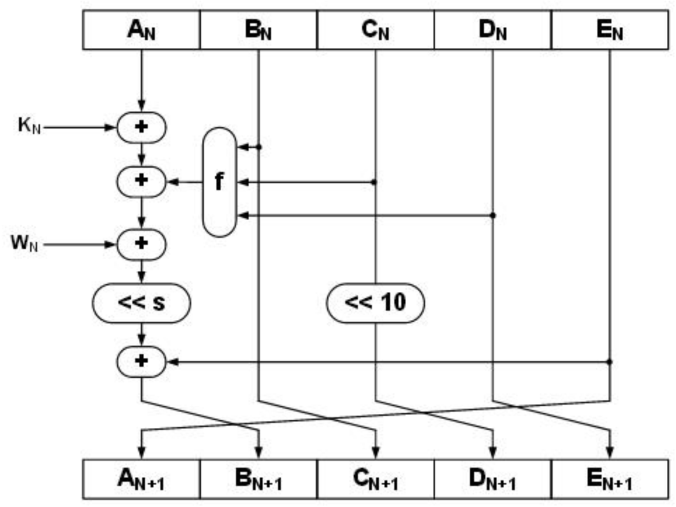


Figure 3.3: Schematic overview of a RIPEMD-160 round from [FM06]

In the following the notation of RIPEMD-160 is given.

- A, B, C, D and E are 32-bit words of the state;
- F is a nonlinear function that varies;
- $\lll_n$ denotes a left bit rotation by n places;
- $\ggg_n$ denotes a right bit rotation by n places;
- n varies for each operation;
- W_n is the expanded message word of round t;
- K_n is the round constant of round t;
- $\boxplus$ Addition denotes addition modulo 2^{32}.
- $\oplus$ denotes bitwise XOR
- $\wedge$ denotes bitwise AND
- $\vee$ denotes bitwise OR
- $\neg$ denotes bitwise complement

The used boolean functions are:

$$\begin{array}{ll} f(j,x,y,z) = x\ XOR\ y\ XOR\ z & (0 <= j <= 15) \\ f(j,x,y,z) = (x\ AND\ y)\ OR\ (NOT(x)\ AND\ z) & (16 <= j <= 31) \\ f(j,x,y,z) = (x\ OR\ NOT(y))\ XOR\ z & (32 <= j <= 47) \\ f(j,x,y,z) = (x\ AND\ z)\ OR\ (y\ AND\ NOT(z)) & (48 <= j <= 63) \\ f(j,x,y,z) = x\ XOR\ (y\ OR\ NOT(z)) & (64 <= j <= 79) \end{array}$$

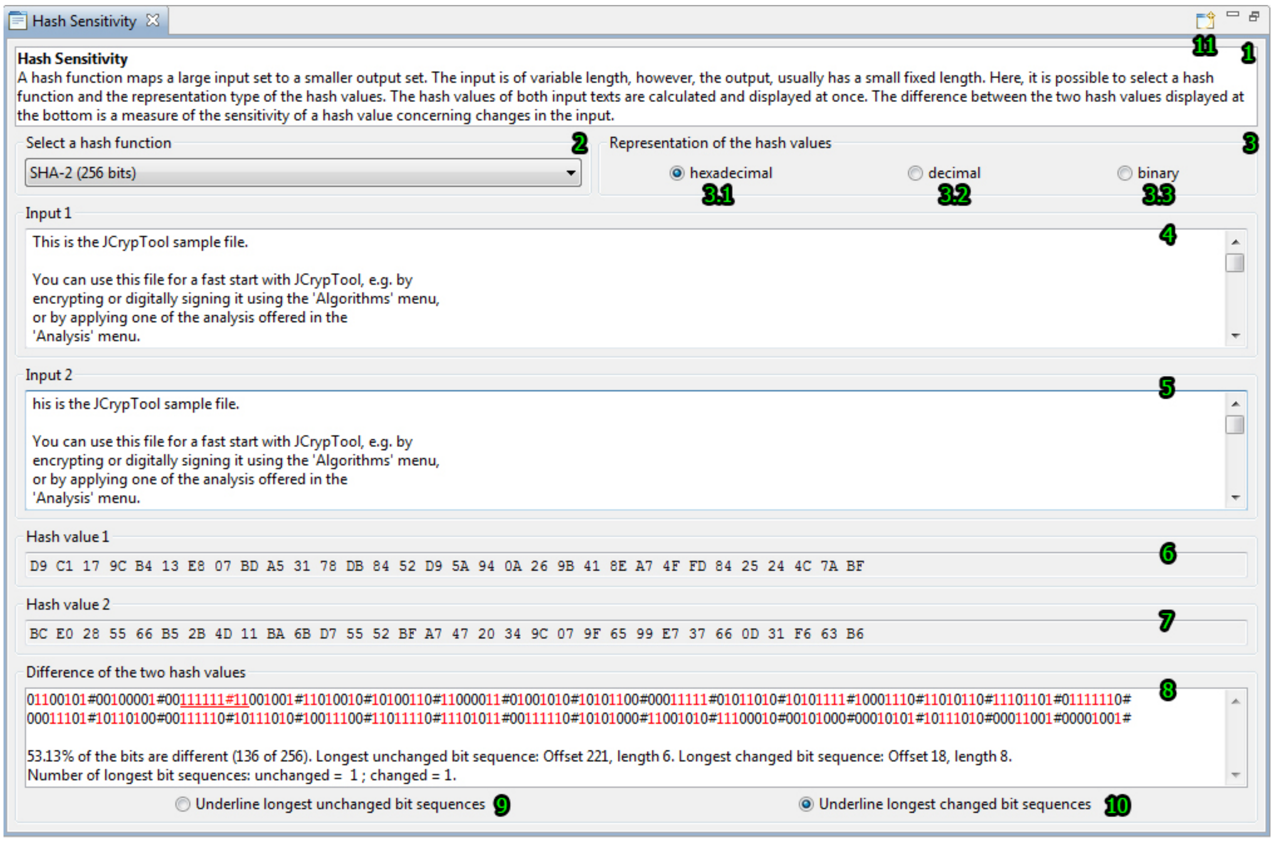
In this chapter we intentionally explained only three hash methods as examples. For clarity reasons listing 3.3 also only contains these three hash methods. In the plug-in all hash methods listed in table 3.1 under #2.1 are implemented.

3.4 The design of the plug-in

In this section we want to go into the design of our plug-in. Again, we use the GUI Guideline [JCT12] of JCrypTool to create the user interface.

The plug-in shows how sensitive the hash value is, when doing even minor changes to one of the inputs. For the visualization the hash values are observed as bit sequence (binary representation) and the comparison of two hash values results in a bit sequence as well. Within this sequence two different types of sub sequences occur: The so-called zero-runs only consist of zeros, whereas the so-called one-runs only consist of ones. A zero-run indicates a bit sequence being identical in both the first input and the second input. The longest zero-run together with its length and its zero-based offset is stated in the bottom line. The statistical analysis of the longest zero-run was the basic approach to the successful analysis of the MD-4 hash function.

The **Hash Sensitivity** plug-in is schematically shown in the figure 3.4. The user interface and its attributes are described in table 3.1 and table 3.2:



Hash Sensitivity

Hash Sensitivity
A hash function maps a large input set to a smaller output set. The input is of variable length, however, the output, usually has a small fixed length. Here, it is possible to select a hash function and the representation type of the hash values. The hash values of both input texts are calculated and displayed at once. The difference between the two hash values displayed at the bottom is a measure of the sensitivity of a hash value concerning changes in the input.

Select a hash function: SHA-2 (256 bits)

Representation of the hash values: [✓ hexadecimal] decimal binary

Input 1

```
This is the JCrypTool sample file.

You can use this file for a fast start with JCrypTool, e.g. by
encrypting or digitally signing it using the 'Algorithms' menu,
or by applying one of the analysis offered in the
'Analysis' menu.
```

Input 2

```
his is the JCrypTool sample file.

You can use this file for a fast start with JCrypTool, e.g. by
encrypting or digitally signing it using the 'Algorithms' menu,
or by applying one of the analysis offered in the
'Analysis' menu.
```

Hash value 1

```
D9 C1 17 9C B4 13 E8 07 BD A5 31 78 DB 84 52 D9 5A 94 0A 26 9B 41 8E A7 4F FD 84 25 24 4C 7A BF
```

Hash value 2

```
BC E0 28 55 66 B5 2B 4D 11 BA 6B D7 55 52 BF A7 47 20 34 9C 07 9F 65 99 E7 37 66 0D 31 F6 63 B6
```

Difference of the two hash values

```
01100101#00100001#00111111#11001001#11010010#10100110#11000011#01001010#10101100#00011111#01011010#10101111#10001110#11010110#11101101#01111110#
00011101#10110100#00111110#10111010#10011100#11011110#11101011#00111110#10101000#11001010#11100010#00101000#00010101#10111010#00011001#00001001#

53.13% of the bits are different (136 of 256). Longest unchanged bit sequence: Offset 221, length 6. Longest changed bit sequence: Offset 18, length 8.
Number of longest bit sequences: unchanged =  1 ; changed = 1.
```

Underline longest unchanged bit sequences [✓ Underline longest changed bit sequences]

Figure 3.4: Hash Sensitivity plug-in

- **Type**: The `SWT-Widget` element.
- **Initial state**: The Initial state of the `Widget` (enabled or disabled).
- **Text**: The name of the used `Widget`.
- **Description**: The description of the `Widget`.

#	Type	Initial state	Text	Description
1	StyledText	disabled	see figure 3.4	Description of the plug-in.
2	Group	enabled	Select a hash function	Includes drop down menu for hash function selection.
2.1	Combo	enabled	see figure 3.4	In this place you can select the hash function. Currently available hash functions are: • MD2 (128 bit) • MD4 (128 bit) • MD5 (128 bit) • SHA-1 (160 bit) • SHA-2 (256 bit) • SHA-2 (512 bit) • SHA-3 (224 bit) • SHA-3 (256 bit) • SHA-3 (384 bit) • SHA-3 (512 bit) • SKEIN-256 (256 bit) • SKEIN-512 (512 bit) • SKEIN-1024 (1024 bit) • SM3 (256 bit) • RIPEMD-160 (160 bit) • TIGER (192 bit) • GOST3411 (256 bit) • WHIRLPOOL (512 bit)
3	Group	enabled	see figure 3.4	Shows the representation of the hash values.
3.1	Button	enabled	hexadecimal	Switch to hexadecimal view of the hash values.
3.2	Button	enabled	decimal	Switch to decimal view of the hash values.
3.3	Button	enabled	binary	Switch to binary view of the hash values.
4	Textfield	enabled	see figure 3.4	In this field the user can enter the input 1 text for the hash value calculation. The JCrypTool example text is loaded by default when the plug-in is initially shown.
5	Textfield	enabled	see figure 3.4	In this field the user can enter the input 2 text for the hash value calculation. The JCrypTool example text is loaded by default when the plug-in is initially shown.
6	Textfield	enabled	see figure 3.4	Generates automatically the hash value for the input 1 text which was entered in the input 1 text field.

Table 3.1: The behavioral description of the **Hash Sensitivity** plug-in – part 1

#	Type	Initial state	Text	Description
7	Textfield	enabled	see figure 3.4	Generates automatically the hash value for the input 2 text which was entered in the input 2 text field.
8	Textfield	enabled	differences of the hash values	In this field the differences of the hash values of input 1 and input 2 are shown. The bits which are different are highlighted in red.
9	Button	active	Underline longest unchanged bit sequences	By clicking on the button the longest unchanged bit sequence will be underlined.
10	Button	deactivated	Underline longest changed bit sequences	By clicking on the button the longest changed bit sequence will be underlined.
11	Icon	enabled	see figure 3.4	**Restart** icon: By clicking on the button the initial state of the plug-in will be opened. If you move the mouse over the icon, the tooltip **Restart** will be shown.

Table 3.2: The behavioral description of the **Hash Sensitivity** plug-in – part 2

3.5 The implementation of the plug-in

The plug-in is grouped in four packages:

- `org.jcryptool.visual.hashing`
- `org.jcryptool.visual.hashing.algorithm`
- `org.jcryptool.visual.hashing.handlers`
- `org.jcryptool.visual.hashing.views`

The activator class `HashingPlugin` of the plug-in is located in the package `org.jcryptool.visual.hashing`. The package `org.jcryptool.visual.hashing.handlers` contains the `RestartHandler` which implements the restart functionality of the plug-in. In the package `org.jcryptool.visual.hashing.algorithm` we implement an enumeration `HashFunction` which is shown in listing 3.1. The method `getName(String name)` returns the corresponding hash function for the given name and otherwise it returns `null`.

```java
public enum HashFunction {
   MD2("MD2 (128 bits)"), MD4("MD4 (128 bits)"), MD5("MD5 (128 bits)"),
   SHA1("SHA-1 (160 bits)"), SHA256("SHA-2 (256 bits)"), SHA512("SHA-2 (512
   bits)"), RIPEMD160("RIPEMD-160 (160 bits)");
   private final String hashFunctionName;
   private HashFunction(String name) {
      hashFunctionName = name;
   }
   public HashFunction getName(String name) {
      for (HashFunction h : values()) {
         if (h.hashFunctionName.compareToIgnoreCase(name) == 0) {
            HashFunction value = valueOf(h.name());
            return value;
         }
      }
      return null;
   }
}
```

Listing 3.1: Enumeration class `HashFunction`

The user interface is implemented in the class `HashingView` of the package `org.jcryptool.visual.hashing.views`. First we have the description field where the functionality is explained. For the selection of a hash function we use a `Combo` and add all the supported hash functions. It implements a `SelectionListener` which is shown in listing 3.2.

```java
public void widgetSelected(SelectionEvent e) {
   if (!textInput.getText().isEmpty()) {
      hashInputValueHex = computeHash(comboHash.getText(), textInput.getText(),
          textHashInput);
   }
   if (!textOutput.getText().isEmpty()) {
      hashOutputValueHex = computeHash(comboHash.getText(), textOutput.getText(),
          textHashOutput);
   }
   if (!textInput.getText().isEmpty() && !textOutput.getText().isEmpty()) {
      computeDifference();
   } else {
      textDifference.setText("");
   }
}
```

Listing 3.2: Selection of a hash function

The method checks if the texts from input 1 and input 2 are not empty. If so, the hash values of the two input fields will be calculated and are displayed in the field below. To calculate the hash value the method `private String computeHash(String hashName, String inputText, Text hashText, String hashHexValue)` will be used. Only if both text fields are not empty the differences of both hash values will be calculated with the method `computeDifference()`. The implementation of this method is shown in listing 3.3.

```java
private String computeHash(String hashName, String inputText, Text hashText) {
   hash = hash.getName(hashName);
   byte[] digest = null;
   switch (hash) {
   case MD2:
      MD2Digest md2 = new MD2Digest();
      md2.update(inputText.getBytes(), 0, inputText.getBytes().length);
      digest = new byte[md2.getDigestSize()];
      md2.doFinal(digest, 0);
      break;
   case MD4:
      MD4Digest md4 = new MD4Digest();
      md4.update(inputText.getBytes(), 0, inputText.getBytes().length);
      digest = new byte[md4.getDigestSize()];
      md4.doFinal(digest, 0);
      break;
   case MD5:
      MD5Digest md5 = new MD5Digest();
      md5.update(inputText.getBytes(), 0, inputText.getBytes().length);
      digest = new byte[md5.getDigestSize()];
      md5.doFinal(digest, 0);
      break;
   case SHA1:
      SHA1Digest sha1 = new SHA1Digest();
      sha1.update(inputText.getBytes(), 0, inputText.getBytes().length);
      digest = new byte[sha1.getDigestSize()];
      sha1.doFinal(digest, 0);
      break;
   .
   .
   .
   default:
      break;
   }
   String hashHexValue = new String(Hex.encode(digest));
   if (btnHexadezimal.getSelection()) {
      String hashValueOutput = hashHexValue.toUpperCase().replaceAll(".{2}", "$0 ");
      hashText.setText(hashValueOutput);
   } else if (btnDezimal.getSelection()) {
      String hashValue = hexToDecimal(hashHexValue);
      hashValue = hashValue.replaceAll(".{3}", "$0 ");
      hashText.setText(hashValue);
   } else if (btnBinary.getSelection()) {
      String hashValue = hexToBinary(hashHexValue);
      hashValue = hashValue.replaceAll(".{8}", "$0#");
      hashText.setText(hashValue);
   }
   return hashHexValue;
}
```

Listing 3.3: Implementation of the hash calculation

The method has three parameters, the name of the hash function, the text which has to be used for the calculation and the text field where we write the calculated hash value back and returns the calculated hash value. First we deliver by name the hash function and store it into our enumeration variable with `hash = hash.getName(hashName)`. Then we use a `switch` statement for the hash function. To calculate the hash function we use the Bouncy Castle[2] crypto library. We will demonstrate this for SHA256 here, the procedure differs only in the creation of the hash digest. We create a `SHA256Digest` with `SHA256Digest sha256 = new SHA256Digest()`. Then we use the `update` method to update it with the input text. For the result we create a `byte` array of the size of the `SHA256Digest` with `digest = new byte[sha256.getDigestSize()]` and finally we write the calculated hash value into the array with `sha256.doFinal(digest, 0)`. We use a `Hex` encoder to encode the `byte` array to a `String` with `String hashHexValue = new String(Hex.encode(digest))`. Before we return the hash value we first check which representation of the hash value is selected to transform the hash value to the correct representation.

To compute the difference in the bit sequences we use the method `computeDifference()` which is shown in listing 3.4. We store the input 1 and input 2 hex values into two variables `input` and `output` of type `BigInteger` as hexadecimal. Then we use the `xor` method to build the `xor` of them and save it into the variable `result` with `String result = input.xor(output).toString(16)`. We convert the hexadecimal representation into binary to show the difference of the bits.

```java
private void computeDifference() {
   BigInteger input = new BigInteger(hashInputValueHex, 16);
   BigInteger output = new BigInteger(hashOutputValueHex, 16);
   String result = input.xor(output).toString(16);
   result = hexToBinary(result);

   if (result.toString().equalsIgnoreCase("0")) {
      textDifference.setText((hexToBinary("0").replaceAll(".{8}", "$0#")));
   } else {
      int count = result.length();
      int zeroBits = result.length() - result.replace("0", "").length();
      int oneBits = result.length() - result.replace("1", "").length();
      double percent = ((double) oneBits / (double) count) * 100;
      int[] sequence = find(result);

      result = result.replaceAll(".{8}", "$0#");
      int lengthPrettyPrint = result.length();
      count = lengthPrettyPrint / OUTPUT_SEPERATOR;

      StringBuilder sb = new StringBuilder(result);
      for (int i = 0; i < count; i++) {
         sb.insert(((OUTPUT_SEPERATOR) * (i + 1) + i), "\n");
      }
      if (hash == HashFunction.RIPEMD160 || hash == HashFunction.SHA1) {
         sb.insert(sb.length(), "\n");
      }
      result = sb.toString();
      char[] bitArray = result.toCharArray();

      textDifference.setText(result + "\n" + String.format("%1$,.2f", percent)
            + Messages.HashingView_12 + oneBits + Messages.HashingView_13 + (zeroBits +
                oneBits)
            + Messages.HashingView_14 + sequence[1] + Messages.HashingView_15 +
               sequence[0] + ".");
      for (int i = 0; i < bitArray.length; i++) {
         if (bitArray[i] == '1') {
            StyleRange bits = new StyleRange();
            bits.start = i;
```

2 https://www.bouncycastle.org/

```
            bits.length = 1;
            bits.foreground = this.getSite().getShell().getDisplay().getSystemColor(SWT
                .COLOR_RED);
            textDifference.setStyleRange(bits);
        }
    }
}
```

Listing 3.4: Calculate the bit differences

If the texts of input 1 and input 2 are equal then there are no differences in the bits or their hash values, and we fill up the output with zeros. Otherwise, we count the bits which represent binary ones and binary zeros and the percentage of the different bits for the statistics. We also want to find the longest unchanged bit sequence and its offset. For that we use the method `private int[] findLongestSequence(String s)` shown in listing 3.5 which returns an `int` array with the longest sequence and its offset.

```
private int[] findLongestSequence(String s) {
    int[] result = new int[2];
    String currentSequence = null;
    String prevSequence = null;

    Matcher m = Pattern.compile("(0+)").matcher(s);
    m.find();
    prevSequence = m.group();
    currentSequence = m.group();

    result[0] = prevSequence.length();
    result[1] = m.start();

    while (m.find()) {
        currentSequence = m.group();
        if (prevSequence.length() < currentSequence.length()) {
            prevSequence = m.group();
            int pos = m.start();

            result[0] = prevSequence.length();
            result[1] = pos;
        }
    }
    return result;
}
```

Listing 3.5: Finding longest sequence

We use a regular expression to find all sequences of zeros. Then we iterate all matching elements and search the longest element and its offset. The length of the sequence we store at index 0 and the offset at index 1 and return the `result` array. Finally we highlight the binary ones in red with a `StyleRange`.

The user can switch to any time the representation of the calculated hash values from hexadecimal to decimal to binary. The `SelectionListener` of the decimal button is shown in listing 3.6. The `SelectionListener` for the hexadecimal and binary conversion are equal. The conversion of the values will only be performed if the input 1 or the input 2 text are not empty.

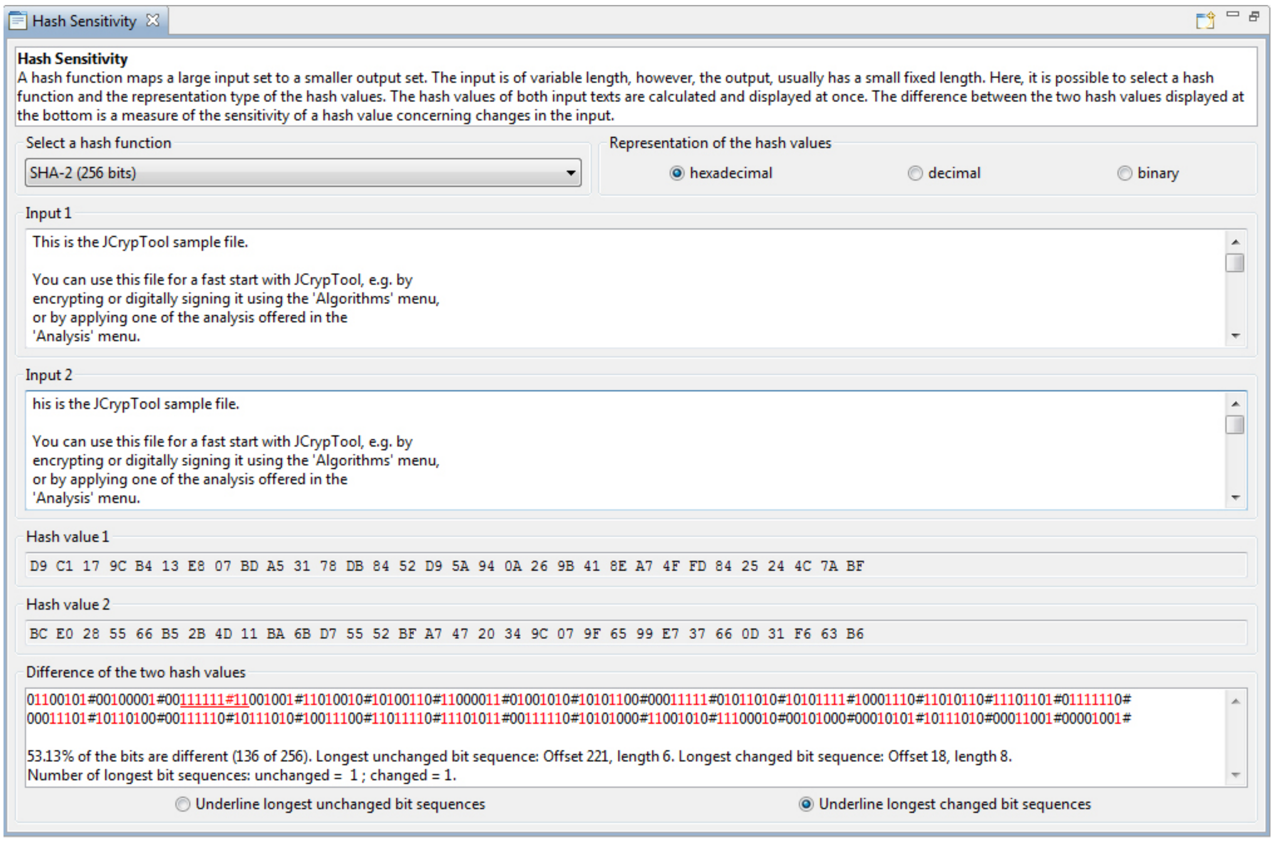
```
public void widgetSelected(SelectionEvent e) {
    String hash = null;
    if (!textInput.getText().isEmpty()) {
        hash = hexToDecimal(hashInputValueHex);
        hash = hash.replaceAll(".{3}", "$0 ");
        textHashInput.setText(hash);
    }
    if (!textOutput.getText().isEmpty()) {
       hash = hexToDecimal(hashOutputValueHex);
       hash = hash.replaceAll(".{3}", "$0 ");
       textHashOutput.setText(hash);
    }
}
```

Listing 3.6: Decimal representation button

It is easy to add new hash functions to the plug-in if they are supported by the Bouncy Castle crypto library. To add a new hash function you only have to add in the method `computeHash` shown in listing 3.3 an instance of the new hash function. The calculation of the hash function itself is very performant even if the hash values are longer than 512 bit. What the performance of the plug-in slows down a bit is the calculation of the statistics of the hash sensitivity when the hash function is bigger than 512 bits.

3.6 The functionality of the plug-in

In this section we describe the functionality of the plug-in **Hash Sensitivity**. The plug-in can be started via the menu **Visuals** or via the crypto explorer tab **Visuals**.



Hash Sensitivity
A hash function maps a large input set to a smaller output set. The input is of variable length, however, the output, usually has a small fixed length. Here, it is possible to select a hash function and the representation type of the hash values. The hash values of both input texts are calculated and displayed at once. The difference between the two hash values displayed at the bottom is a measure of the sensitivity of a hash value concerning changes in the input.

Select a hash function: SHA-2 (256 bits)

Representation of the hash values: [✓ hexadecimal] decimal binary

Input 1

```
This is the JCrypTool sample file.

You can use this file for a fast start with JCrypTool, e.g. by
encrypting or digitally signing it using the 'Algorithms' menu,
or by applying one of the analysis offered in the
'Analysis' menu.
```

Input 2

```
his is the JCrypTool sample file.

You can use this file for a fast start with JCrypTool, e.g. by
encrypting or digitally signing it using the 'Algorithms' menu,
or by applying one of the analysis offered in the
'Analysis' menu.
```

Hash value 1

```
D9 C1 17 9C B4 13 E8 07 BD A5 31 78 DB 84 52 D9 5A 94 0A 26 9B 41 8E A7 4F FD 84 25 24 4C 7A BF
```

Hash value 2

```
BC E0 28 55 66 B5 2B 4D 11 BA 6B D7 55 52 BF A7 47 20 34 9C 07 9F 65 99 E7 37 66 0D 31 F6 63 B6
```

Difference of the two hash values

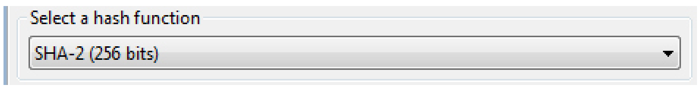
```
01100101#00100001#00111111#11001001#11010010#10100110#11000011#01001010#10101100#00011111#01011010#10101111#10001110#11010110#11101101#01111110#
00011101#10110100#00111110#10111010#10011100#11011110#11101011#00111110#10101000#11001010#11100010#00101000#00010101#10111010#00011001#00001001#

53.13% of the bits are different (136 of 256). Longest unchanged bit sequence: Offset 221, length 6. Longest changed bit sequence: Offset 18, length 8.
Number of longest bit sequences: unchanged =  1 ; changed = 1.
```

Underline longest unchanged bit sequences [✓ Underline longest changed bit sequences]

Figure 3.5: Hash Sensitivity plug-in

The plug-in consists of a description field and four basic areas: **Hash function**, **Input 1 text**, **Input 2 text** and **Difference**. In the description field some statements about the algorithm are displayed.
In the hash function area you can select a hash function and the representation you want.



Select a hash function: SHA-2 (256 bits)

Figure 3.6: Hash function selection

There are the following hash functions available:

- MD2 (128 bit)
- MD4 (128 bit)
- MD5 (128 bit)
- SHA-1 (160 bit)
- SHA-2 (256 bit)
- SHA-2 (512 bit)
- SHA-3 (224 bit)
- SHA-3 (256 bit)

- SHA-3 (384 bit)
- SHA-3 (512 bit)
- SKEIN-256 (256 bit)
- SKEIN-512 (512 bit)
- SKEIN-1024 (1024 bit)
- SM3 (256 bit)
- RIPEMD-160 (160 bit)
- TIGER (192 bit)
- GOST3411 (256 bit)
- WHIRLPOOL (512 bit)

The hash function can be represented in **hexadecimal**, **decimal** or **binary**.

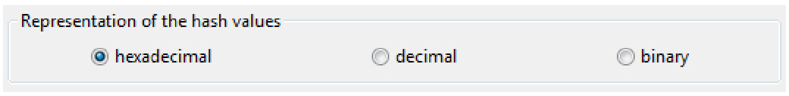
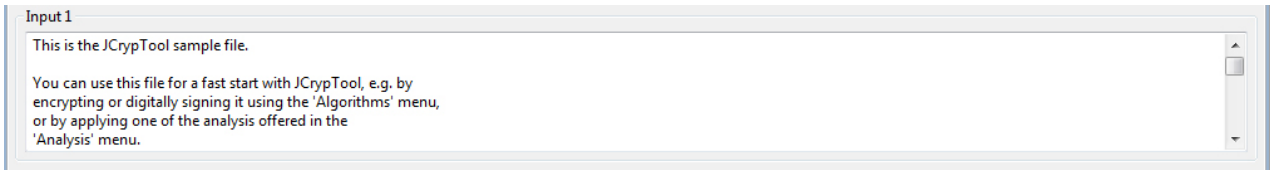


Figure 3.7: Hash value representation

In the input 1 text area you can enter any text.

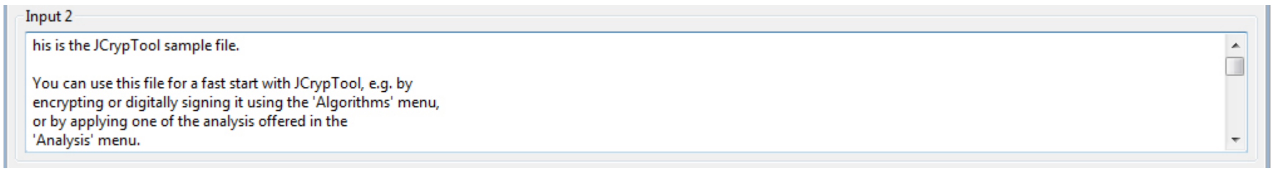

Input 1

This is the JCrypTool sample file.

You can use this file for a fast start with JCrypTool, e.g. by
encrypting or digitally signing it using the 'Algorithms' menu,
or by applying one of the analysis offered in the
'Analysis' menu.

Figure 3.8: Input 1 text

The same for the input 2 text area.


Input 2

his is the JCrypTool sample file.

You can use this file for a fast start with JCrypTool, e.g. by
encrypting or digitally signing it using the 'Algorithms' menu,
or by applying one of the analysis offered in the
'Analysis' menu.

Figure 3.9: Input 2 text

The hash value of input 1 and input 2 text will be calculated and displayed below in the corresponding representation.

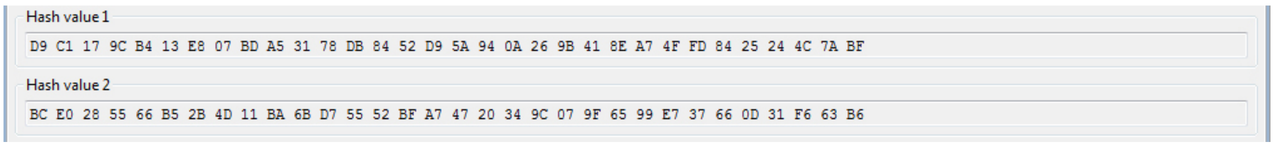
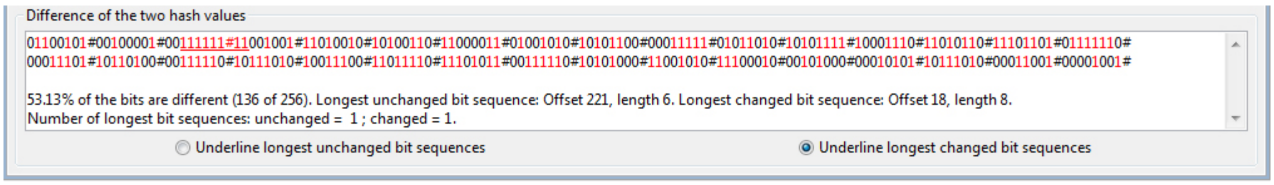

Hash value 1
```
D9 C1 17 9C B4 13 E8 07 BD A5 31 78 DB 84 52 D9 5A 94 0A 26 9B 41 8E A7 4F FD 84 25 24 4C 7A BF
```
Hash value 2
```
BC E0 28 55 66 B5 2B 4D 11 BA 6B D7 55 52 BF A7 47 20 34 9C 07 9F 65 99 E7 37 66 0D 31 F6 63 B6
```

Figure 3.10: Hash values

Based on the entry of the input 1 and input 2 text the differences of the bits in the hash value will be displayed in the difference area. The different bits are highlighted in color red.


Difference of the two hash values

```
01100101#00100001#00111111#11001001#11010010#10100110#11000011#01001010#10101100#00011111#01011010#10101111#10001110#11010110#11101101#01111110#
00011101#10110100#00111110#10111010#10011100#11011110#11101011#00111110#10101000#11001010#11100010#00101000#00010101#10111010#00011001#00001001#

53.13% of the bits are different (136 of 256). Longest unchanged bit sequence: Offset 221, length 6. Longest changed bit sequence: Offset 18, length 8.
Number of longest bit sequences: unchanged =  1 ; changed = 1.
```

[ ] Underline longest unchanged bit sequences [✓ Underline longest changed bit sequences]

Figure 3.11: Difference between the hash values of input 1 and input 2

There are also some statistics displayed. You can change at any time the hash function and the representation of the hash value.

When you click the radio button **Underline longest unchanged bit sequences** the unchanged bit sequence will be underlined. The same is for the button **Underline longest changed bit sequences**.

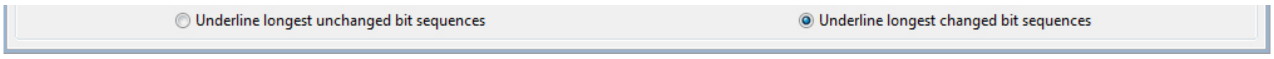

Underline longest unchanged bit sequences
Underline longest changed bit sequences

Figure 3.12: Underline longest unchanged and changed bit sequence

When you click the **Restart** icon in the menu bar the default text will be loaded into the text areas of input 1 and input 2.

Figure 3.13: Restart functionality

4 Outlook

In this diploma thesis we designed, implemented and presented the two plug-ins **Merkle-Hellman** and **Hash Sensitivity** for the visualization category of the e-learning platform JCrypTool. A seamless integration in the existing open-source project and an interactive and easy usability of the plug-ins were the main purpose. The user group which includes students, teachers and any cryptography-interested person should get a clear overview and easy access to these cryptographic algorithms. In the bilingual online help (German and English), the user can read the description of the algorithms and the functionality of the plug-ins directly.

During the preparation of our thesis we learned a lot by participating in this open-source project: On the one hand we could use the content of our classes at the university by enhancing our own understanding of the theory and bring it into practice. On the other hand we learned how to participate and contribute to an active and distributed open-source project, to develop plug-ins with the Eclipse Framework, and especially how to design user-friendly GUIs.

We also took advantage by some students who tested the plug-ins and who gave feedback which led to improvements of the plugins. Overall, we received a very positive feedback for both plug-is and their online help confirming they helped to better understand both the idea of the asymmetric cipher from Merkle-Hellman and the high sensitivity of cryptographic hash methods to any kind of change in the text inputs.

For the future we recommend interested students to visit the website of the JCrypTool wiki[1] which proposes other projects which could be implemented to enhance the knowledge about cryptography and the awareness for IT security. Another resource of interesting algorithms for other projects is the book by Prof. Dr. Johannes Buchmann [Buc08].

[1] https://github.com/jcryptool/core/wiki/project-Ideas

Bibliography

[Buc08] BUCHMANN, Johannes: *Einführung in die Kryptographie*. 4., erweiterte Auflage. Springer Verlag, 2008. – ISBN 978–3–540–74451–1

[DG03] DAVID GALLARDO, Robert M. Ed Burnette B. Ed Burnette: *Eclipse in Action: A Guide for Java Developers*. 1. Auflage. Manning Publications, 2003. – ISBN 978–1–930110–96–0

[Dob96] DOBBERTIN, Hans: *The Status of MD5 after a Recent Attack*. 1996

[EC09] ERIC CLAYBERG, Dan R.: *Eclipse Plug-ins*. 3. Auflage. Addison-Wesley Verlag, 2009. – ISBN 978–0–321–55346–1

[FC93] FLORENT CHABAUD, Antoine J.: *Differential Collisions in SHA-0*. 1993

[FM06] FLORIAN MENDEL, Christian Rechberger Vincent R. Norbert Pramstaller P. Norbert Pramstaller: *On the Collision Resistance of RIPEMD-160*. https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=17675. Version: 2006

[HD96] HANS DOBBERTIN, Bart P. Antoon Bosselaers B. Antoon Bosselaers: *RIPEMD-160: A strengthened Version of RIPEMD*. http://homes.esat.kuleuven.be/~bosselae/ripemd160/pdf/AB-9601/AB-9601.pdf. Version: 1996

[JCT12] *Guideline for Designing the GUI in JCrypTool Plug-ins*. https://github.com/jcryptool/doc/blob/master/Guidelines/JCrypTool-GUI-Guidelines.pdf. Version: Juli 2012

[Kna04] KNAPPEN, Jörg: *Schnell ans Ziel mit LATEX 2e*. 2. Auflage. Oldenbourg Wissenschaftsverlag, 2004. – ISBN 978–3–486–27447–3

[Kop00] KOPKA, Helmut: *LaTeX Band 1: Einführung*. 3., überarbeitete Auflage. Addison-Wesley, 2000. – ISBN 978–3–8273–7038–8

[MD515] *MD5*. http://en.wikipedia.org/wiki/MD5. Version: 2015

[MG00] MICHEL GOOSSENS, Alexander S. Frank Mittelbach M. Frank Mittelbach: *Der LaTeX-Begleiter*. 1. Auflage. Addison-Wesley Verlag, 2000. – ISBN 978–3–8273–7044–0

[NI12] *Secure Hash Standard*. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf. Version: 2012

[NR95] N. ROGIER, Pascal C.: *The compression function of MD2 is not collision free*. 1995

[Riv90] RIVEST, Ronald L.: *The MD4 message-digest algorithm, Request for Comments (RFC 1186)*. http://tools.ietf.org/html/rfc1186. Version: 1990

[Riv92] RIVEST, Ronald L.: *The MD5 message-digest algorithm, Request for Comments (RFC 1321)*. http://tools.ietf.org/html/rfc1321. Version: 1992

[RM78] RALPH MERKLE, Martin H.: Hiding information and signatures in trapdoor knapsacks. In: *Information Theory, IEEE Transactions* (Sep 1978)

[RSA15] *3.6.6 What are MD2, MD4 and MD5?* http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/md2-md4-and-md5.htm. Version: 2015

[Sha84] SHAMIR, Adi: A polynomial-time algorithm for breaking the basic Merkle - Hellman cryptosystem. In: *Information Theory, IEEE Transactions Ausgabe* (1984)

[SHA01] *US Secure Hash Algorithm 1 (SHA1)*. http://www.faqs.org/rfcs/rfc3174.html. Version: 2001

[SHA12] *Descriptions of SHA-256, SHA-384 and SHA-512*. http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf. Version: 2012

[WP08] Wouter Penard, Tim van W.: *On the Secure Hash Algorithm family*. `http://www.staff.science.uu.nl/~werkh108/docs/study/Y5_07_08/infocry/project/Cryp08.pdf`. Version: 2008

[XW05] Xiaoyun Wang, Hongbo Y. Yiqun Lisa Yin Y. Yiqun Lisa Yin: Finding Collisions in the Full SHA-1. In: *In Proceedings of Crypto*, Springer, 2005, S. 17–36

List of Figures

List of Tables

Source code directory