1 Inhaltsverzeichnis

2 Abstract

This Document describes the Simplecrypt Encryption Method and Implementation.
Simplecrypt implements a Rotor Based Encryption Method. This Method is primarily famous through the Enigma Machine.
At its time it was a modern Method that got possible by the use of Machine based Encryption instead of the manual Methods before.
Due to this Fact the Germans thought that it was un-crack able, what would probably been right for manual decryption. But while the Allies also used Machines for decryption, they were finally able to decrypt Enigma Messages.
With the invention of the Microcomputer the rotor based encryption got out of the focus, while the rotor based principle was invented to be implemented by mechanical machines, not digital ones.
Also after it is famous, that Enigma as the most famous rotor based encryption machine, it does not seem to make sense to use this principle. But if we take a look at the lessions learned from the Mistakes that had been made with the Enigma, it gets obvious, that it was only possible to Hack the Enigma, because a couple of major mistakes had been made. Any of these mistakes could easily be avoided when this mechanical principle is moved to the digital computer.
In this case a very high level of encryption security can easily be achieved.
All the public known Methods of Cryptanalysis do fail on this Method. It is also presumable, that nonpublic methods of cryptanalysis, that might have been developed by government organizations do not focus on the rotor based method, because this was seen as old fashioned for the last 30 Years.
From this point of view a rotor based encryption could deliver a very good level of privacy, if the mistakes of the past are avoided.
One of the main weak points from the point of view of modern cryptanalysis is, that the Enigma did not align with “Kerkhoffs Prinzip” after which the secret of the key must guarantee the systems’ securety not the algorhthems’ secret. Furthermore the common

believe is that the system is even stronger when the algorhythem is published, because only in this way many experts can challenge the system and find possible weak points. For this reason simplecrypt shall be published here.

3 Method

3.1 General

3.1.1 Rotor Based Encryption

Rotor Based Encryption works on the principle of rotating alphabets. That means that the first character of a Message is encrypted with different alphabet than the second one and so on. To achieve this Rotor is shifting its alphabet for any character.
For Encryption one or multiple Rotors can be used. If more than one Rotor is used the encryption process is done by reading from “left to right”. For decryption it must be read from right to left.

Refer Chapter Algorithm for further Details

3.1.2 History

In the middle age and before a crypto system had to be easy enough, that a man is able to run the encryption and decryption algorithm.
The most famous method was the Cesar Code. In this Method the natural alphabet was shifted like A to B, B to C, C to D etc.
This is pretty simple and so it is pretty easy to decrypt a text. The number of possible Keys in a 26 Character Alphabet was 25. So it was relatively easy to tray any 25 possibilities.
In the middle age there were some enhancements of this like an additional transposition.
In the beginning of the 20th Century machine based encryption got possible. This allowed more complex algorithms. At this time the rotor based encryption was invented. It was based on simple typewriter similar machines that used rotors, to shift electrical contacts. The German Enigma is possibly the most famous machine out of this family.

3.1.3 Lessons learned from Enigma

The Germans thought, that Enigma is un-crack-able. This error was mainly caused by the fact that they did not took in mind, that the code breaking work could also be done machine based.
The encryption strength of the enigma was also based strongly on the secret of the algorithm. The Key strength was comparably low.
Additionally some construction Failures had been done, what was probably a result of the prior described wrong thoughts.
This was especially the Reflector. The reflector mapped the electrical power on the right end of the rotors and mapped it back on another character to the left. While the Germans thought this would increase the security, but while this leads to the facts, that one character could not be encrypted to itself, this decreased the number of possible alphabets significantly. Also this enabled the code breakers to do Position analysis of where a certaim crib could not be placed in the cipher text.

The “Reflektor” was later replaced by a “Umkehrwalze” (reverse Rotor), but that did not fix the weak point.
Another weak point was the fact that the Enigma rotors had only one Notch. The result of this is, that Character sequences of 26 Characters have effectively been encrypted by only one Rotor, because the other Rotors stayed in the same Position.

3.2 Enhancements

3.2.1 Moving the Alphabet from the Algorithm to the Key

At the Enigma only 5 (later 8) Alphabets have effectively been used. For this reason the Alphabets that had been implemented in a physical Rotor were part of the Algorithm not the Key.
In simplecryt, the Alphabet Mapping can be defined freely as a part of the Key. This generates a Key room of ACn! (Factor of the number of Characters in an Alphabet)

3.2.2 Skipping the “Reflektor/Umkehrwalze”

Another important improvement in simplecryt is not to use “Reflektor/Umkehrwalze”. This fixes one mayor weak points in the Enigma Algorithm. Due to this change the full amount of possible Alphabets can be used. There is no reduction of the Key room due to the Reflector. Also fix point exclusion attacks are not possible anymore.
In this way the full Key room can be used. For a Binary Alphabet this would be 256! What gives a number with 508 digits.

3.2.3 Increasing the Alphabet Length

In simplecryt the number of characters per Alphabet is not limited to 26. Actually it is limited to a Maximum of 2^{16} Characters per Alphabet. So the Maximum number of possible Alphabets is $(2^{16})!$ This is a gigantic number. However for Text encryption, this might not be totally useful, because the number of significantly smaller.
For Binary Encryption an Alphabet of 2^8 characters makes the most sense as it is currently implemented. However it would still be possible to add a double-byte-binary encryption to make use of the full set of possible Alphabets.
It does also make sense to increase the number of character above the really used, especially, when the encryption layer above of simplecrypt is making use of additional concepts to increase the security (refer the Chapter 3.4.4)

3.2.4 Unlimited number of Rotors

This increases the possible key room to theoretically unlimited

3.2.5 Free definition of Rotors and Rotor Positions

For Enigma, there was a choice of 5 Rotors in 3 Positions. While any Rotor could be placed only once, this summed up to 5*4*3 = 60 different Positions. In simplecryt it is possible to place one Rotor multiple times. For this reason the key Room is increased to $(\text{Number of possible Alphabets})^{\text{Number of Rotors}}$

3.2.6 Moving the Notch Position and Count to the Key

Different from the Enigma, the Notch Positions can freely be defined per Rotor. This gives 2 advantages against the Enigma:

a. A Code Breaker cannot assume, that a bigger sequence is effectively encrypted with only one Rotor. For this reason this attack is not possible or at least much harder.
b. The Key Room is increased by $2^{\text{number of Characters per Alphabet}}$ per Rotor
However, when defining the Notch Positions, this should be done balanced. If there are too few notches, the sequences with a single Alphabet might be too long. If the notches are too regular and /divide able by themselves, the period for rotor turns might be too short.
The extreme opposite example would be that any Position is a notch Position. In this case all Rotors would turn at any character. So effectively the encryption would run only on a single Rotor, no matter how many rotors are really used.

3.3 Algorithm

For Encryption the basic algorithm works like this:

1. Take the very left rotor
2. Read which char is mapped to the clear text char. (EncChar)
3. Check if right Rotor exists.
   If yes: Start with step one Start at step 1 and EncChar as Input
   If No Result = EncChar
4. Check if current Position has a notch and a Right Rotor Exists
   If yes: Trigger the Right Rotor Position
5. Trigger thisRotors Position

Sample1 Using one Rotor:
In our Example we use a short and simple Alphabet for one Rotor. In the 0 Position the Alphabet is defined as:

Clear Char	Enc Char
A	A
B	B
C	C
D	D

In this case the Text ABBA will be encoded to AADB

Char	RPos	RcCh	ReCh	Result
A	0	A	A	→ A
		B	B	
		C	C	
		D	D	
B	1	A	D	
		B	A	→ A
		C	B	
		D	C	
B	2	A	C	
		B	D	→ D
		C	A	
		D	B	

A	3	A	B	→ B
		B	C	
		C	D	
		D	A	

Sample2 Using two Rotors:
In our Example we use a short and simple Alphabet for two Rotors. In the 0 Position the Alphabets are defined as:

Rotor 1		Rotor 2	
Clear Char	Enc Char	Clear Char	Enc Char
A	A	A	D
B	B	B	C
C	C	C	B
D	D	D	A

In our Simple Sample only one Notch Position is defined. A Notch Position is a character that is turning the next Rotor for one Position.

In this case the Text ABBAABBA will be encoded to DDACAABD

	Rotor1			Rotor2			
Char	RPos	RcCh	ReCh	RPos	RcCh	ReCh	Result
A	0	**A**	**A**	0	**A**	**D**	→ D
		B	B		B	C	
		C	C		C	B	
		D	D		D	A	
B	1	A	D	0	**A**	**D**	→ D
		B	**A**		B	C	
		C	B		C	B	
		D	C		D	A	
B	2	A	C	0	A	D	
		B	**D**		B	C	
		C	A		C	B	
		D	B		**D**	**A**	→ A
A	3	**A**	**B**	0	A	D	
		B	C		**B**	**C**	→ C
		C	D		C	B	
		D	A		D	A	
A	0	**A**	**A**	1	**A**	**A**	→ A
		B	B		B	D	
		C	C		C	C	
		D	D		D	B	
B	1	A	D	1	**A**	**A**	→ A
		B	**A**		B	D	

		C	B		C	C	
		D	C		D	B	
B	2	A	C	1	A	A	
		B	**D**		B	D	
		C	A		C	C	
		D	B		**D**	**B**	→ B
A	3	**A**	**B**	1	A	A	
		B	C		**B**	**D**	→ D
		C	D		C	C	
		D	A		D	B	

So encrypting is reading from left to right and decrypting is reading from right to left.
However decryption will only work properly when starting at the same Rotor position.

Sample3 Using multiple Notches:
Here we use the same setup as in sample 2 but add an addition Notch at Position 0 (marked with a *.

In this case the Text ABBAABBA will be encoded to DABDAADB

	Rotor1			Rotor2			
Char	RPos	RcCh	ReCh	RPos	RcCh	ReCh	Result
A	0*	**A**	**A**	0	**A**	**D**	→ D
		B	B		B	C	
		C	C		C	B	
		D	D		D	A	
B	1	A	D	1	**A**	**A**	→ A
		B	**A**		B	D	
		C	B		C	C	
		D	C		D	B	
B	2	A	C	1	A	A	
		B	**D**		B	D	
		C	A		C	C	
		D	B		**D**	**B**	→ B
A	3*	**A**	**B**	1	A	A	
		B	C		**B**	**D**	→ D
		C	D		C	C	
		D	A		D	B	
A	0*	**A**	**A**	2	**A**	**B**	→ A
		B	B		B	A	
		C	C		C	D	
		D	D		D	C	

B	1	A	D	3	**A**	**C**	→ A
		B	**A**		B	B	
		C	B		C	A	
		D	C		D	D	
B	2	A	C	3	A	C	
		B	**D**		B	B	
		C	A		C	A	
		D	B		**D**	**D**	→ D
A	3	**A**	**B**	3	A	C	
		B	C		**B**	**B**	→ B
		C	D		C	A	
		D	A		D	D	

This shows how the Notch Positions change the chiffre text and shortens the sequences that use the same permutation.

3.4 Key

3.4.1 Elements

The key that is required for decryption consists of different Elements:

3.4.1.1 Alphabet Definition

The Alphabet Definition per Rotor defines the mapping of one Character to another.
Of course there are no duplicate characters allowed on both sides.
The simplecrypt implementation allows the use of Unicode characters so a maximum of about 65.000 Characters per Alphabet
The binary encryption uses 256 Byte values (Characters)
An Alphabet must at least cover all Characters used in the clear text.
The Alphabet mapping should not be ordered (like in the Samples above) to decrease patterns in the cipher text.

The definition is made with two Blocks of Characters, one for the clear text character (Rotors left side) and one for the Encryption text character (rotors right side.
The number of Characters in both blocks must be identical. The mapping is done by the characters Position within its block.

The characters used in the CC Block must all be defined in the EE Block as well.
No Character should be defined twice.

3.4.1.2 Notch Definition

The Notch Positions are the Rotor Positions at which the next right Rotor is triggered to move its Position.
The Notch Positions of the very right Rotor have no influence of the encryption.

The Notch Positions can be defined freely and should be non-regular and frequent enough (measured at the alphabet length) to leave less recurring patterns in the cipher text
Number of Rotors and Rotor Position

3.4.1.3 Start Positions

The start Position defines at what Rotor Position the encryption/decryption has to start. The number of possible Start Position is equal to the number of characters used in the Alphabet.

The start Position does not increase the effective key room, because another start Position of one Alphabet is identical to exactly one other Alphabet permutation that is already part of the key room.
Example:
A Rotor using the Alphabet (A=A, B=B, C=C, D=D) starting at Position 1 is identical to the use of the Alphabet (A=B, B=C, C=D, D=A) starting at Position 0

3.4.1.4 Rotor Definition

All Key Parts mentioned before can be defined per Rotor. A theoretical unlimited number of Rotors can be defined.

3.4.2 String Representation

The string representation of a key is an easy way to exchange readable keys. However in this representation it is difficult to encode Alphabets, containing non printable characters. For this reason it cannot be used for binary encryption

3.4.2.1 Alphabet

The String representation would be:
CC[ABCD]CCEE[DCBA]EE

The [] brackets open and terminate the actual Data.
The magic Number CC marks the start – CC[-- and the end --]CC of the block defining the clear character or the left Rotor side
The magic Number EE marks the start – EE[-- and the end --]EE of the block defining the encrypted character or the right Rotor side

3.4.2.2 Notch Positions

The String representation would be:
KK[0,3]KK

The magic number KK marks the start and the end of the notch position block. The actual position is a list of numbers separated by comma

3.4.2.3 Start Positions

The String representation would be:
SS[0]SS

The magic number SS marks the start and the end of the start position block. The actual position numbers representing the start position of this rotor

3.4.2.4 Rotor

The String representation would be:
WW{CC[ABCD]CCEE[DCBA]EE KK[0,3]KK SS[0]SS}WW

The {} brackets open and terminate the actual Data. A Rotor Definition must cover a Clear Character, a Encrypted Character, a notch positions and a start position Block.
The magic Number WW marks the start – WW{ -- and the end --}ww of the block defining the rotor
A Key can contain multiple Rotors. The order of rotors in the key defines the rotor positions from left to right

3.4.3 Binary Representation

3.4.3.1 Current Implementation

The binary Key format is exactly the same as the string format, but byte wise casted from character to byte.

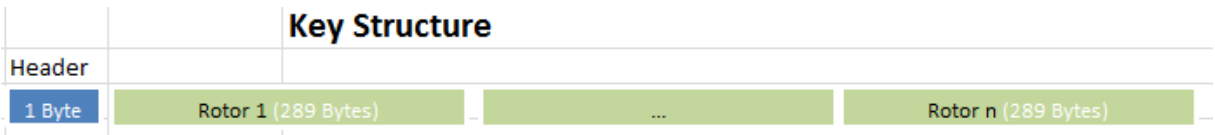
3.4.3.2 Problems with the current format

- The current binary key format is not able to define Unicode characters
- It is longer as necessary, because the numbers are encoded digitwise. The comma separator is also unnecessary
- It is highly recommended, that the keys themselves should not be exchanged uncrypted, because in this case the synchronous clear text key would weaken the cryptosystem significantly, no matter how good the encryption itself might be. Due to this fact, it is a weak point, that the key contains magic numbers and fixed tokens. This will make it easier to crack the keys' key.
- For a binary Key, the clear characters are not necessary, because the order will be the byte values -127 to 128 anyway.

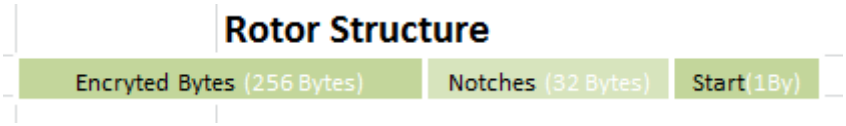
3.4.3.3 Plans for future format

The magic Numbers and Tokens shall be removed. Unnecessary characters shall be removed. Numbers shall be encoded as byte values not at digits in a string.

The Key would start with one Byte defining the number of rotors, followed by 289 Byte Chunks per rotor.



A Rotor chunk would be 256 Bytes for the encoded bytes (characters) followed by 32 Bytes for the notch positions and one byte for the start position



The 256 bits of the 32 bytes for the notch positions define if a position has a notch (1) or not (o). The bist from left to right represent the position 0 to 255

Header			
Pos	SubPos	Name	Content
1	1	KEY_LEN	Number of Rotors in this key
Rotor			
Pos	SubPos	Name	Content
2-257	1-256	ENC_CHAR	Encryted Byte corresponding to Alpabets Byte (1-256)
258-289	1-32	NOTCH_BYTE	The Bits of this Bytes represent wheather the Position has a notch or not from the left Bit to the Right Bit for Positions 1-256
290	289	START	Value indicating the start Position for this Rotor

This format would still not cover Unicode characters. It could be enhanced by using 2Bytes per character 8192 bytes for the notch positions.
However this would produce very long keys.

3.4.4 Advanced Handling

The secret of the key is a weak point for any cryptosystem. While simplecrypt in general uses a synchronous key (both communication partners require the same key for encryption and decryption, this is a weak point for communication with simplecrypt.
For this reason simplecrypt implements some functions that shall enable applications to implement secure key exchange/handling and keeping it easy to use for the user.

3.4.4.1 Hiding Keys in Images

One way to prevent a key from being stolen is to hide the key. SimpleCrypt implements functions to hide a binary key into a bitmap (Key Factory and CryptImage Classes). Depending on the covering Image, it would be hard to recognize the actual key. For a machine it could be made impossible.

3.4.4.2 Encrypting Keys by Passphrase

Even, if it is hard to locate the key, it is probably not impossible. For this reason, the key itself should be encrypted. The Key crypt Class offers a simple but effective way to encrypt a key stream by a passphrase.
The security of this encryption relies on the Key length. A Key that is as long as the stream to be encrypted would guarantee a perfect security.
This could be a passage of a book or textile. For an easier usage this could be any other passphrase, but the security might be reduced with short or easy to guess passphrases.

3.4.4.3 Asynchronous Key usage

For a message based communication, one key should not be used too often. But when a synchronous key is used, the exchange of the keys is the weak point in the system. Even with the enhancements above, a frequent key exchange could weaken the whole system.
A compromise could be to split the key. For example the main Key would contain only the rotors and the alphabet definitions, but not the Notch and Start Positions. The would be defined in a secondary Key that might change, based on an agreement, like the Date, the time a passphrase or anything else.
Currently simplecryt contains no implementations to support this.

3.4.4.4 Ideas for secure Applications

Next to the encryption and key strength itself, communication and encryption is always more secure, if you try to make the hackers life harder. The basic way for this is to work against possible attacks.
The Ideas and methods can be combined. They do mainly focus on keeping full benefit out of the strong encryption by making the hackers life harder.
They do also focus on keeping the balance of security on the one hand and an easy usage on the other. For nonprofessionals the major issue with cryptographic applications is that the usage is often complicated. That leads to the fact, that many people pass encryption at all. Even a perfect encryption would be useless, if it is not used.

3.4.4.4.1 Avoid cribs (binary or message)

Cribs are words or phrases that a hacker could know or easily guess to be part of the cipher text.
For Text messages this is a crucial Problem, because it does mainly remain to the submitter and his knowledge to avoid things like starting each Message with “Hello” or using names the hacker might guess etc. However some of the following ideas could help the user to avoid this.
For the encryption of binary files cribs would come with the file Format. For Example a Bitmap file foes always start with the two letters (magic number) “BM”.
If the complete Bitmap will now be encrypted, this would have two disadvantages.

1. The File will be corrupted, because the magic numbers will now be encrypted and the file will not be recognized as a Bitmap anymore.
2. A hacker would know that the clear text will start with “BM”

The first point is not important for the security but not that nice. But the second point could help the hacker a lot.
For this reason it is recommended to encrypt only a documents payload, instead of the complete data stream.
Simplecrypt implements some basic abstract classes (CryptDoc, CrytImage, CryptTextDoc) to help with the implementation of this Idea. It contains also some sample Classes with practical Implementations for some File Formats (CryptBMP, CryptPNG, CryptTXT)

3.4.4.4.2 Common Text shortening (message)

One possible way to improve the message communication would be the use of shorter language. Like SMS or Chat Language. An Application could do this automatically prior to submission with a simple replacement for example you → u, right → r8, at → @ and so on.

The advantage would be, that it will be harder to find cribs and it would also make it harder to perform a letter analysis (a cryptoanalysis technique to count the number of the used letters to identify weather a decryted text is of a natural language or not)
But his would require that the clear text language would be known and implemented, because only a very few of these rules are independent from the used language.
Also this will probably make it harder for the receiver to understand the message correctly.

Currently simplecrypt does not contain any help for this. It could be realized with a pre-processing of the message.

3.4.4.4.3 Custom Dictionaries (message)

A further advance of the text shortening could be to use a custom alphabet to encode a text. In this way often used names could be replaced as well. A dictionary could contain words, keywords, names or parts of words like stenography.
The advantage here would be, that this could be in both directions and the message integrity is not at risk.
At the same time this would bother a hacker a lot, because many of his methods are not able to work.

Currently simplecrypt does not contain any help for this. It could be realized with a pre- and post-processing of the message.

3.4.4.4.4 Hide crypted message in other data (message)

As well as hiding a key for example in a Bitmap it will improve security, if the hacker does not recognize the message at all.
It depends on the image content if or how easy a message could be identified, For example using a screenshot of this document would contain a lot of white pixels. At these areas pixels of multiple colors would be easy to discover. Using a photo of a multi-color theme will hide the “foreign pixels” much better. A computer based program might not be able to detect this, but it will still be detectable for the human eye.
Using a encrypted or random generated Bitmap will hide the “foreign pixels” completely. But this type of senseless Image might make an eavesdropper suspicious.
In simplecrypt the writeBinaryKeyToBuitmap methods could also be used to hide messages

3.4.4.4.5 One to multiple Character Encryption (message)

One method of cryptanalysis is to find or count the number of the different characters. Another way to make this harder is to encrypt certain or all characters to more than one character. For example a “U” will be encrypted not only to “e” but also to “&”
Simplecrypt doe currently not support this, but it could be implemented with pre- and post-processing of the message, or by adding special Alphabet and Rotor Classes to simplecrypt.

3.4.4.4.6 Hide the key (key handling)

The key should never be exchanged or stored in clear text, to avoid side attacks. Images or other big binary blobs are a good Idea to hide the key.
One Idea to combine an easy usage and a good privacy for a mail applications like an outlook plug in, could be to store the key for the communication with one person in the contact

photo of this person in the contacts or the local path to the image containing the key in a custom field.

3.4.4.4.7 Encrypt Keys when exchange (key handling)

The Key Encryption by Passphrase is pretty good, if the Passphrase is long enough. To keep this usable, the Passphrase could be a passage of a book or a block of bytes in an image.
In this way, the "Key to the Key" would be master file (like a book let's say the bible) and the start position of the passphrase. This key can be memorized well or exchanged separately by phone or face to face.

3.5 Strengths

- Simplecrypt can produce theoretically unlimited large key rooms
  A Binary Encryption with 4 Rotors would produce an effective key room of $(256!*2^{256})^3*256! = 1{,}33*10^{2845}$ this comes up to `9448 Bit`.
- At an average system we were able to check 25.000 keys/s. If we assume, that a powerful computer could be 1.000 times faster, and if we assume, that the implementation for the brute force could be 10 times more efficient than ours the average time (n/2) required to find the key in a brute force attack would be 8,47*102828 years.
  If we do assume, that the calculation power of computers continues to double each year (Moorsches Gesetz), it would still take 9399 Years until the key could be found in less than one Day.
- The strength of the key is very high.
- The classic methods of cryptanalysis do not work, to crack the code
- The methods, that had been used to crack enigma do not work
- The modern methods of cryptanalysis, should not work, because they focus on completely different algorithms
- Can be implemented be be very easy to use

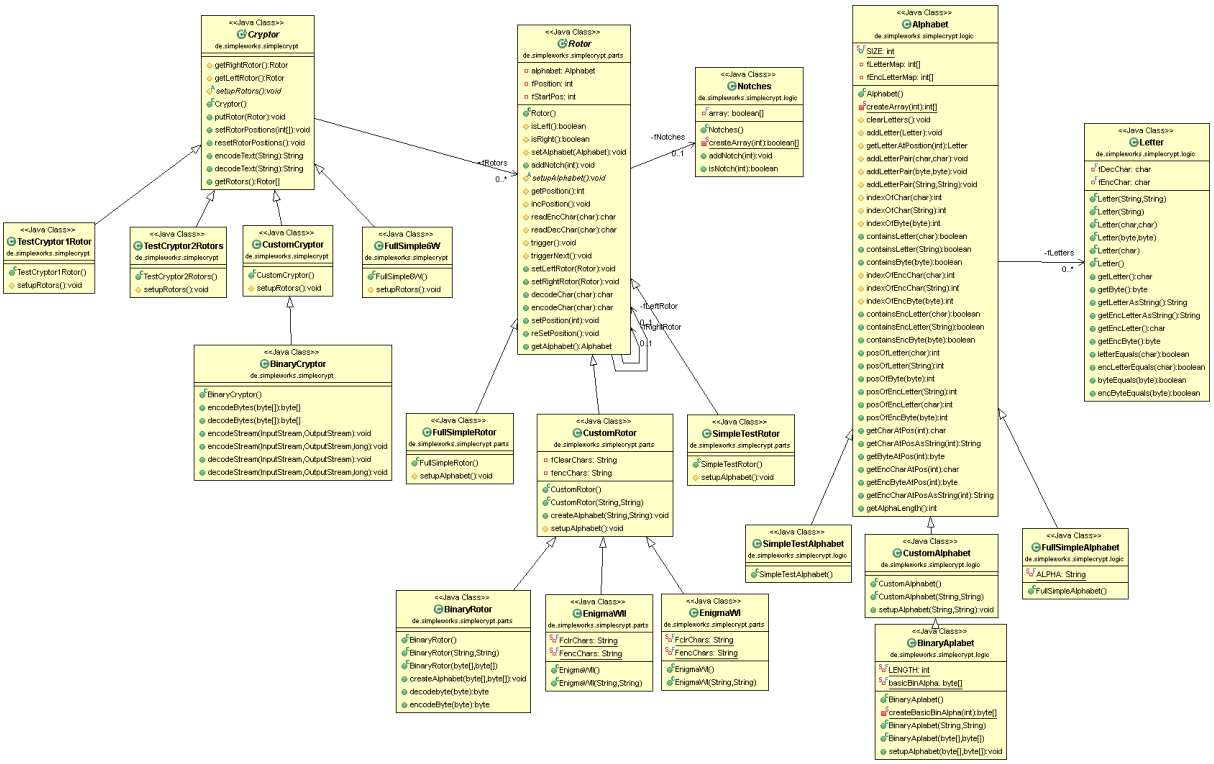
3.6 Weak Points

- The system has not been published for a long time, so the community has not had a lot of time to find more weak points.
- It uses synchronous keys, even if this can be mitigated (s. Ideas for key handling)

4 Implementation

4.1 General Architecture

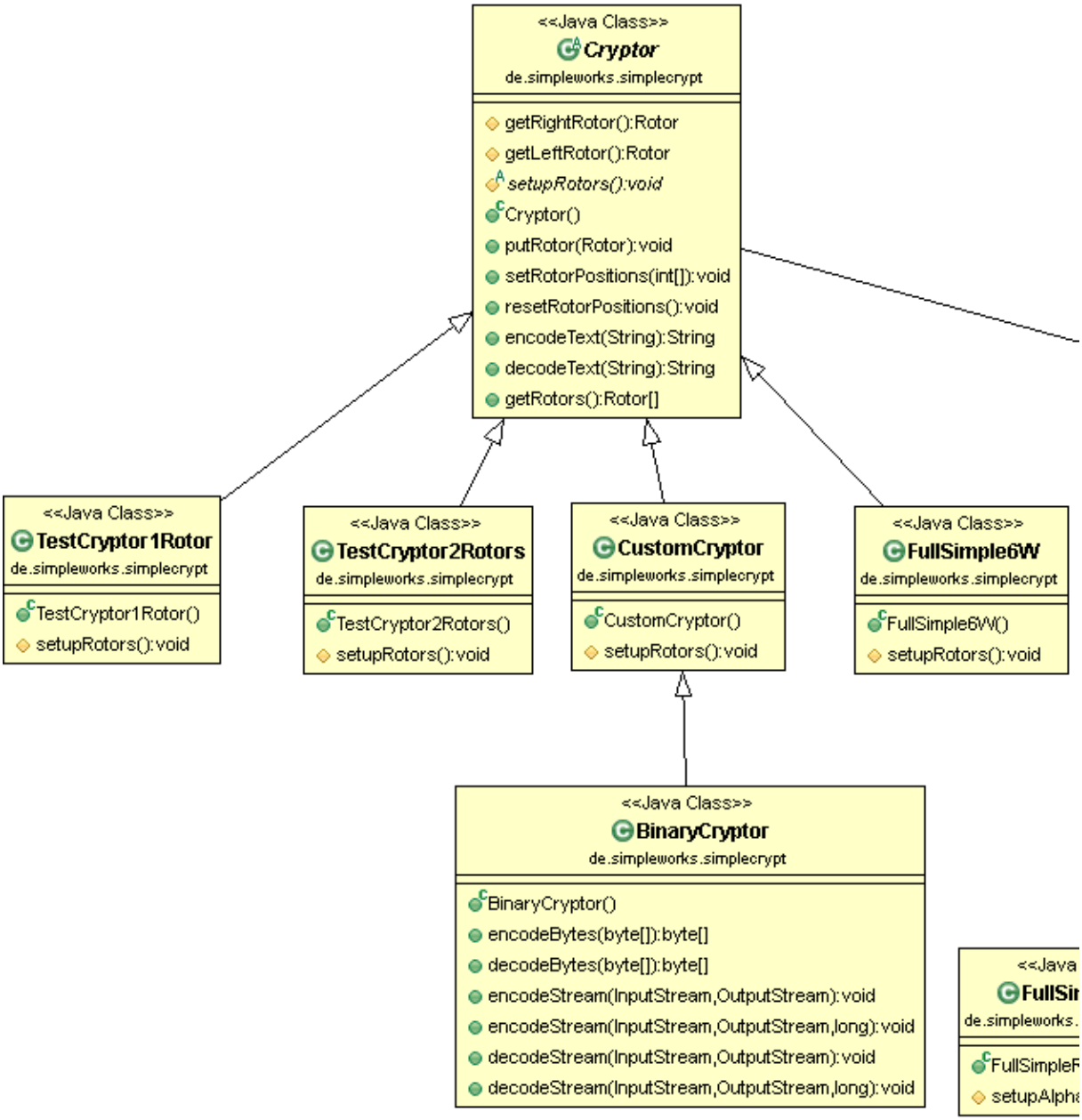
4.2 Core Simplecrypt Classes



The Core Encryption Engine consist of the Classes LetterPair (represents one pair of mapped characters), Alphabet (A Collection of LetterPairs), Notches (representing NotchPositions), Rotor (Implementing one Alphabet and Notches) and Cryptor (representing the Chiffre mashine).

4.2.1

4.2.2 Cryptor



The Cryptor Class represents the chiffre mashine.
One Crytor can have one or more Rotors.

Class Cryptor

Is the abstract base class for all Crytors
A fixed machine could derive from this class and implement the rotor setup. This way is not preferably, because only the start Positions could be used as a key.

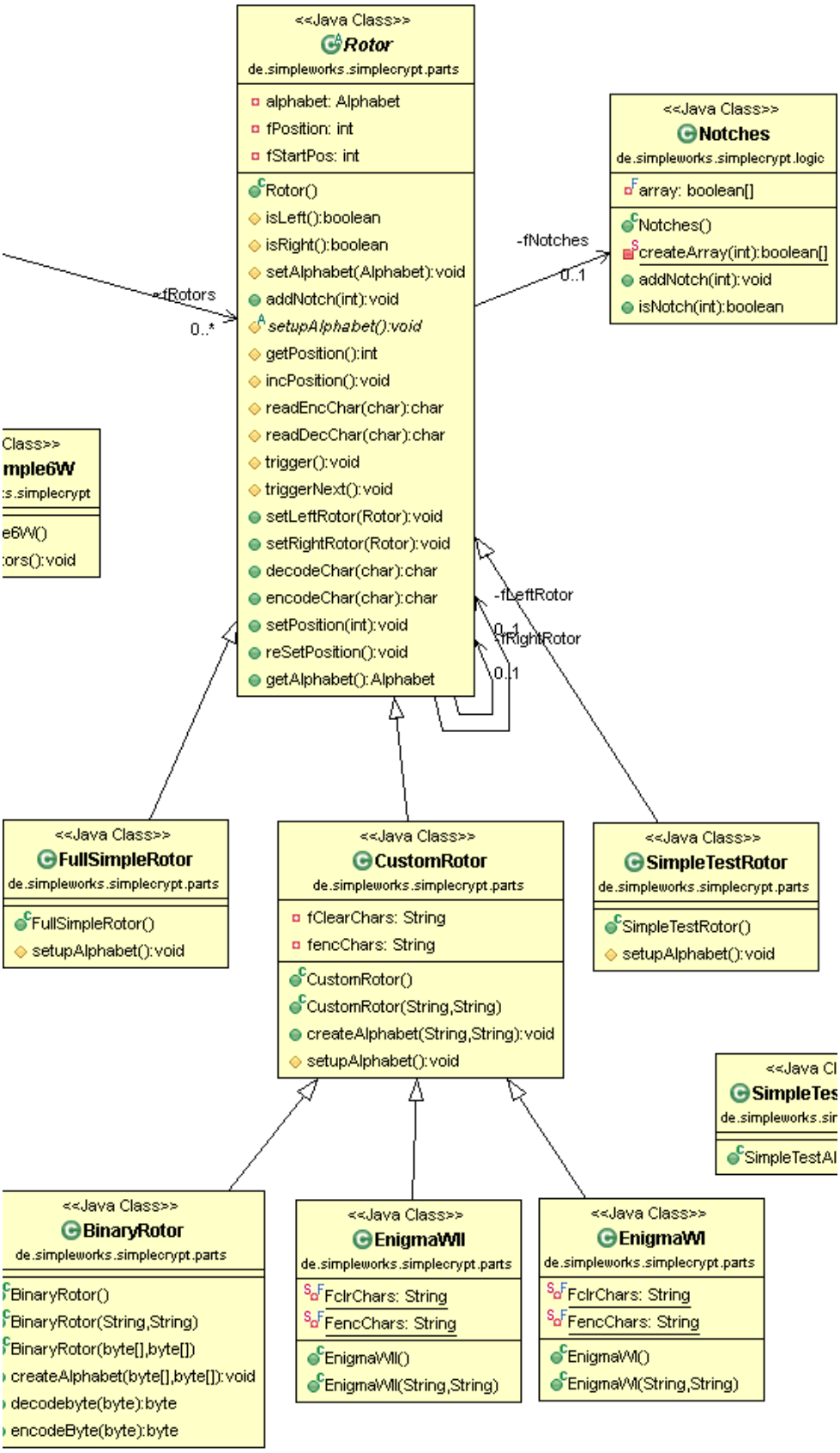
Class CustomCryptor

Derives from Cryptor and is a base class for all Crytors without a fixed rotor setup. It enables the user to use custom keys for the setup

Class BinaryCryptor

Derives from CustomCryptor. It implements functions to do byte based en-/decryption.

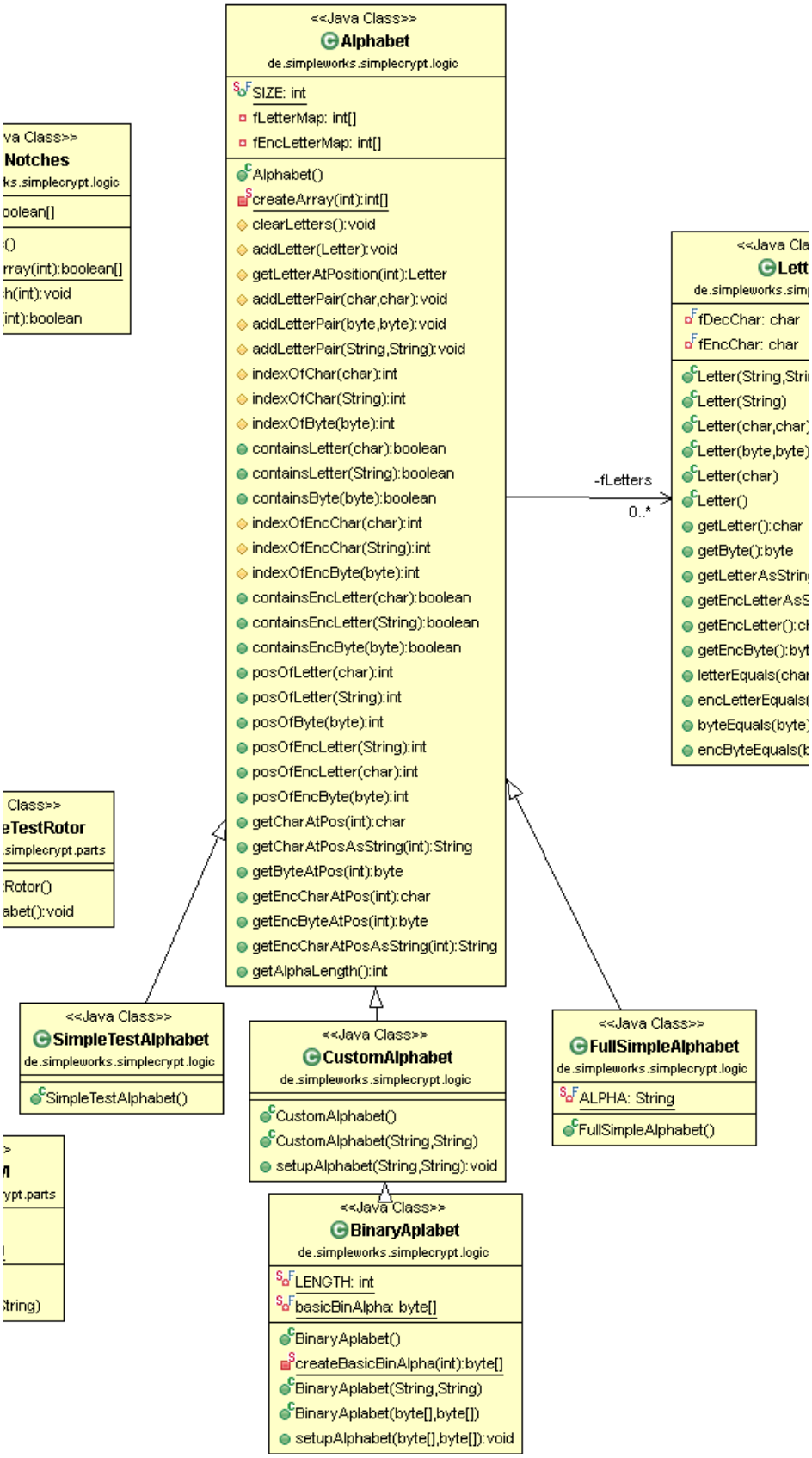
4.2.3 Rotor



The Rotor holds one Alphabet and one set of Notches.

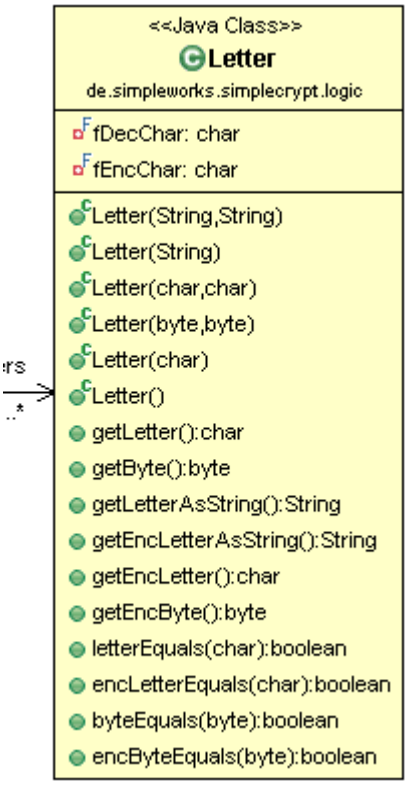
The Rotor Class is doing the main encryption work. It does keep its own position and using the rotors left or right from itself to fulfill this

4.2.4 Alphabet



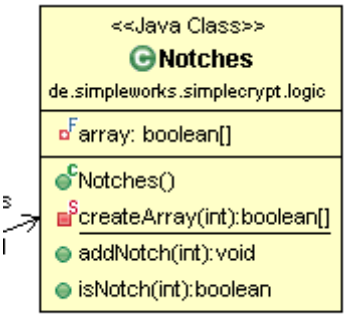
The Alphabet Class is a helper that holds the characters and finds the matching Letters

4.2.5 LetterPair



The letter pair holds the information of the mapping of one character

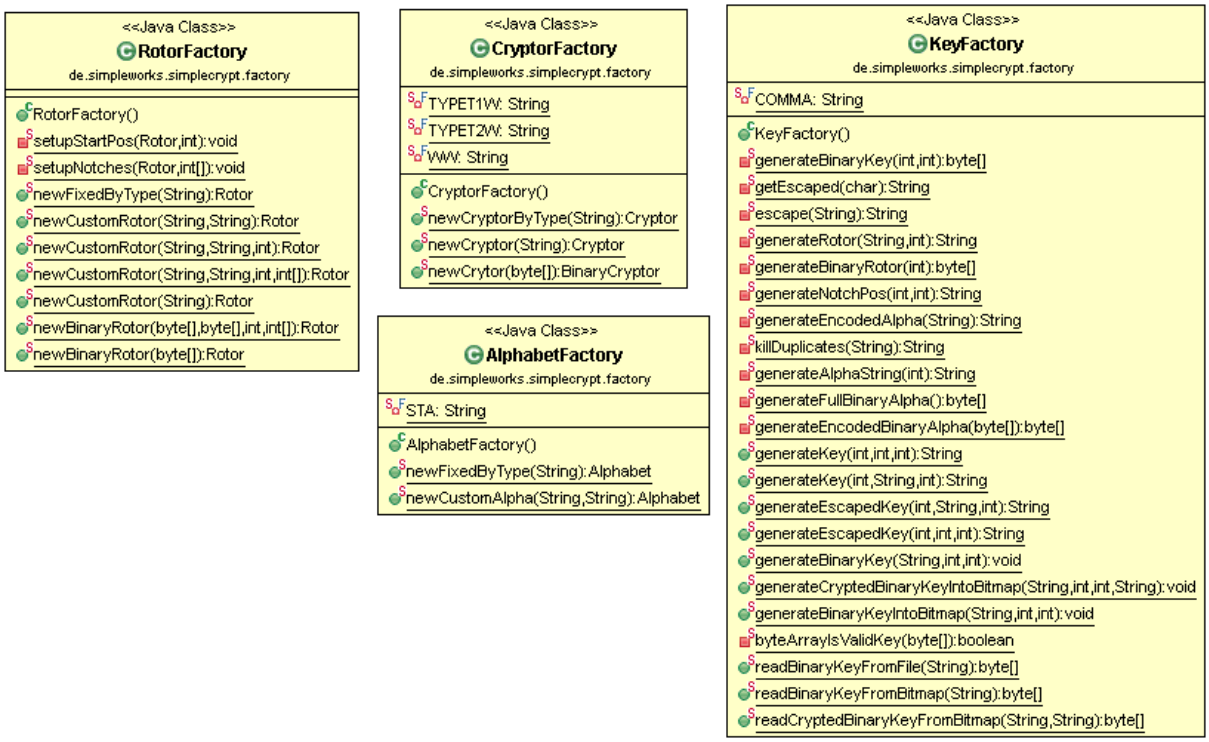
4.2.6 Notches



Holds the information of the potiotion with a notch

4.3 Factories

The Factory Classes help setting up a Custom Crytor.



4.3.1 AlphabetFactory

Creates Objects of the Alphabet Class

4.3.2 RotorFactory

Creates Objects of the Rotor Class

4.3.3 CrytorFactory

Creates Objects of the Cryptor Class Class

4.3.4 KeyFactory

Different from the other Factories, the KeyFactory produces Keys and helps to handle them.

public static String generateKey(int _anzWalzen, int _alphaLen, int _kerbenJeWalze)

Generates a string key representation for anzWalzen rotors, with an alphabet of a given number of characters (alphaLen) with a given number of notches (kerbenJeWalze)

public static String generateKey(int _anzWalzen, String _charSet, int _kerbenJeWalze)

Generates a string key representation for _anzWalzen rotors, with an alphabet of a given set of characters (_charSet) with a given number of notches (_kerbenJeWalze).

Any character in _charSet will be used as clear Character and mixed as a EncEhar

public static String generateEscapedKey(int _anzWalzen, String _charSet, int _kerbenJeWalze)

public static String generateEscapedKey(int _anzWalzen, int _alphaLen, int _kerbenJeWalze)

The same as generate Key, but the string Representation will contain the TAB \ " characters java escaped

public static void generateBinaryKey(String _fileName, int _anzWalzen, int _kerbenJeWalze) throws IOException

Generates a binary key representation for _anzWalzen rotors, with an alphabet of all bytes with a given number of notches (_kerbenJeWalze) and saves it into a file (_fileName).

public static void generateBinaryKeyIntoBitmap(String _fileName, int _anzWalzen, int _kerbenJeWalze) throws IOException

Generates a binary key representation for _anzWalzen rotors, with an alphabet of all bytes with a given number of notches (_kerbenJeWalze) and saves it into a bitmapfile (_fileName).

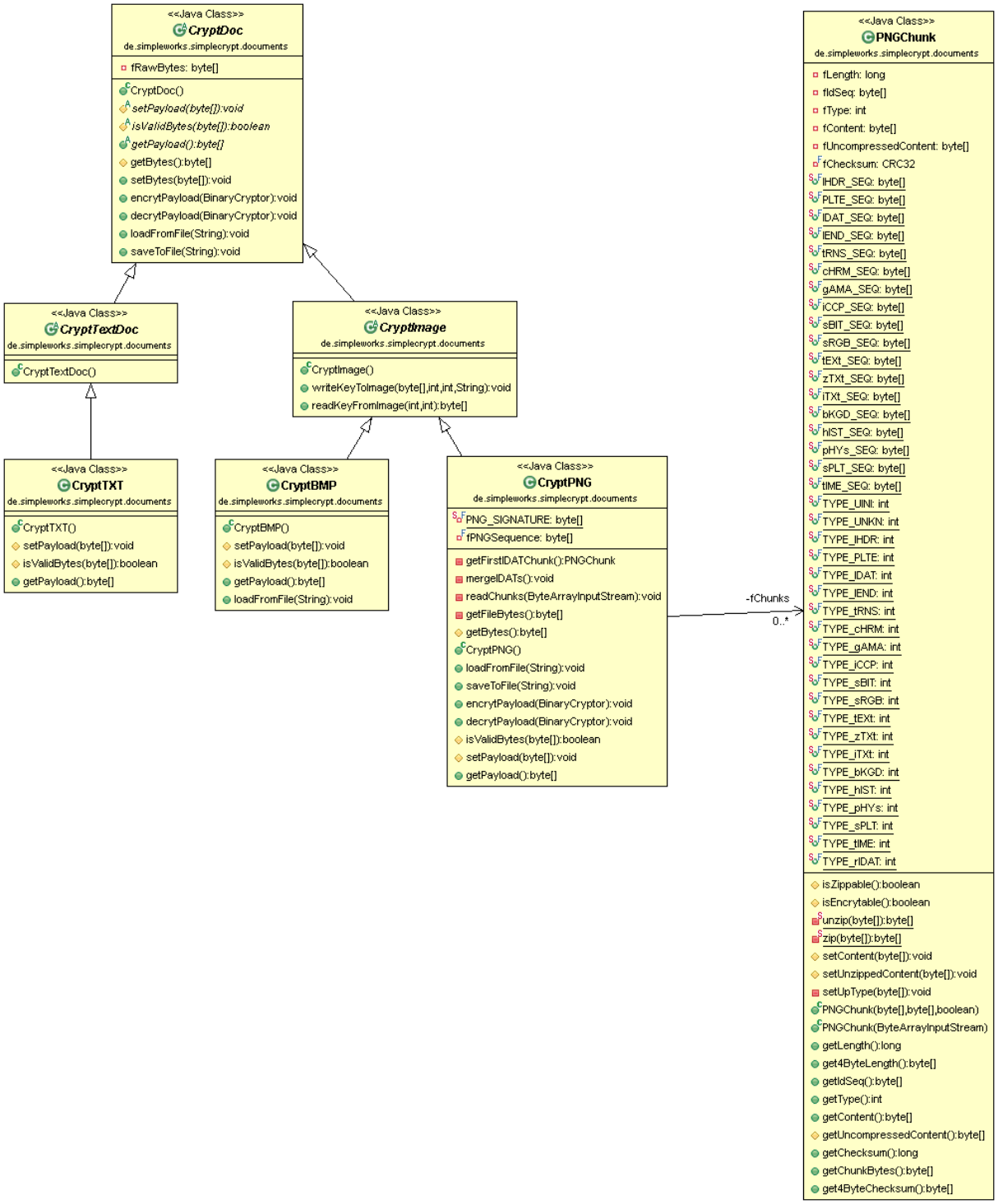
public static byte[] readBinaryKeyFromFile(String _FileName) throws IOException

Reads the binary key representation from a file (_FileName);

public static byte[] readBinaryKeyFromBitmap(String _FileName) throws IOException

Reads the binary key representation from a bitmap file (_FileName);

4.4 Documents

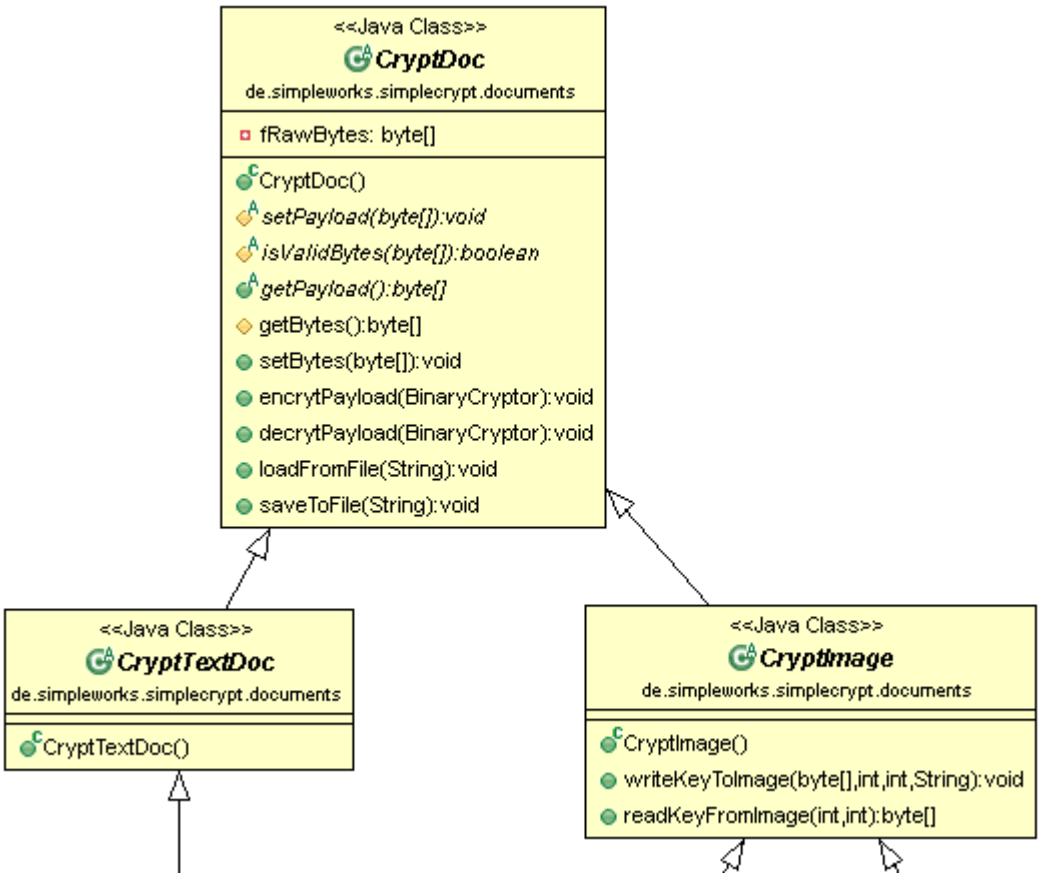


4.4.1 Purpose

The Core Classes can encrypt and decrypt any type of message or stream, but for an advanced encryption of generic binary data it makes sense not to encrypt a complete byte stream, but only the payload. However to do this knowlege about the file/data format is required.

This Class hierarchy shows how this knowlege could be implemented for different types of documents.

4.4.2 Abstract Classes / Structure CryptDoc, CryptImage



The CryptDoc is the abstract basic Class for all Document classes.
Deriving Classes must implement what part(s) of the documents' bytes are payload, that must be encrypted and which not.

The CrpytImage is also abstract, but Implemets the ability to save Keys into them or read keys out of them.

4.4.3 Final Types (BMP, PNG, TXT)

CrpytBMP and CryptPNG derive from CryptImage, they show as an example how to implement a specific Document Type.

CryptTXT derives from the abstract CryptTextDoc and represents any type of simple Text file.

5 Performance

To evaluate the Simplecrypt performance we executed a basic Benshmark, comparing different Simplecrypt configurations to other common Encryption Methods.

5.1 Setup

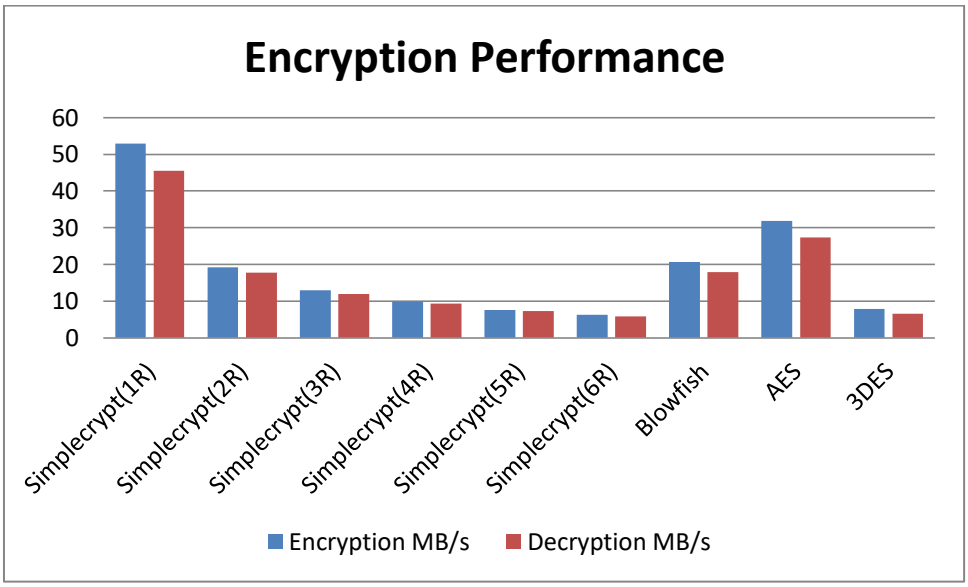
The Java JCA implementations for Blowfish, 3DES (DESede) and AES had been used to compare the encryption speed.
For this test any of the implementations had to encrypt a zip file with a size of about 25 MB.
All Tests had been performed on the same Laptop.

The Simplecrypt performance was measured for Cryptors containing one to 6 Rotors.

5.2 Results

5.2.1 Raw Performance

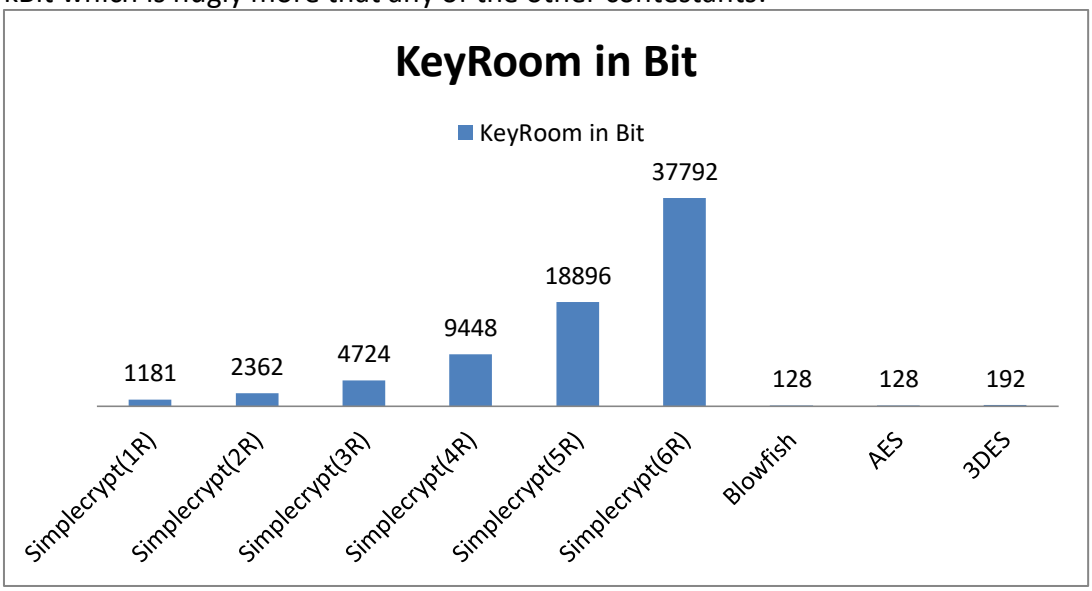
The results were as follows:



The Simplecrypt performance was competitive to the other algorhytms.
The fastest configuration was Simplecrypt with one Rotor.
AES is the 2nd fastest in the row and the 3rd (Blowfish) is slightly faster than Simplecrypt with 2 Rotors.
3DES is in Average about as fast as Simplecrypt with 5 Rotors.

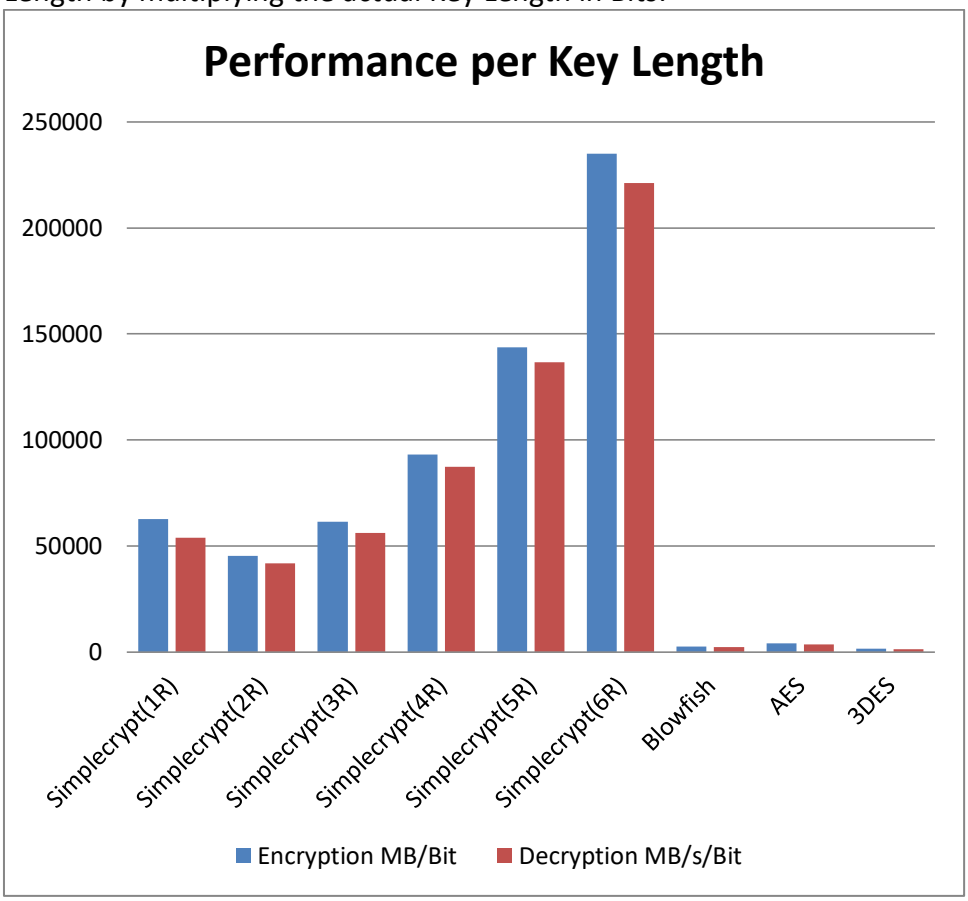
5.2.2 Encryption Stengh (Key Room)

But these are just the raw results. It must be taken into account, that the simplecrypt algorhytm used 1 to 6 Rotors what generated a key rooms of about 1 kBits up to about 36 kBit which is hugly more that any of the other contestants.



5.2.3 Relative Performance to Security

To get this into a picture we calculated the Encryption/decryption performance per Key-Length by multiplying the actual Key Length in Bits:



This comparison shows, that simplecrypt is superior in performance for very high security encryption.

A Configuration of 3 or 4 Rotors is recommended for good performance and enourmous high security.