School of EEE & Computer Science

CSC3048 : Information Systems Security

Report Submission

Student Names:	Student Numbers:
Andrew Fletcher	40086915
Chris Cooper	40067386
David Fee	40042567
Rory Powell	40081529

Description:
CSC3048 Project Submission

Declaration of Academic Integrity

Before signing the declaration below, please check that the submission:

1. has a full bibliography attached, laid out according to the guidelines specified in the Assignment Guidelines
2. contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. does not exceed the specified page limit
4. is clearly presented, word-processed and proof-read
5. is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

I declare that I have read the School of Computer Science student handbook guidelines (http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism/) and the Queen's University regulations on plagiarism, and that the attached submission is my own original work. No part of it has been submitted for any other assignment, and I have acknowledged in my notes and bibliography all written and electronic sources used.

Student's signature **Date of Submission 18/04/2016**

Index

1 Report

1.1 Overall Code Structure

1.1.1 Project build

The project uses maven as a build and dependency management system. The main interaction point with maven is the parent **pom.xml** file in the project root, this defines the project structure via sub maven modules and dependencies of the project.

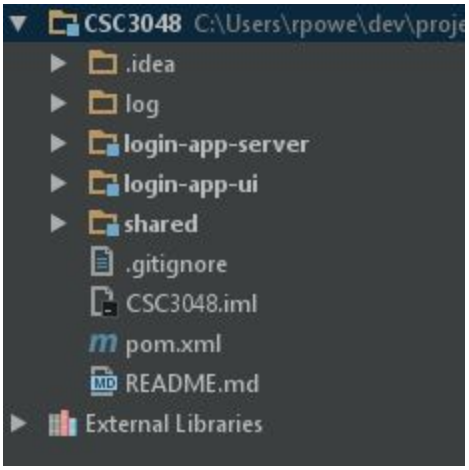
The dependencies of the parent module are:

- junit 4.11 (Unit testing)
- spring-data-jpa (Object relational mapping)
- spring-security (Authentication)
- spring-web (Rest services)
- h2 (Database)
- commons-io 2.4 (Utils)
- gson 2.6.2 (JSON serialization)

1.1.2 Database

An embedded database is used for the login application, this is for both speed and usability purposes. The inclusion of the h2 dependency in the parent pom file includes the database in the project, it is configured using java based config to provide the datasource. The default configuration stores a flat file in the home directory of the user under ~/h2/.

1.1.3 Module structure

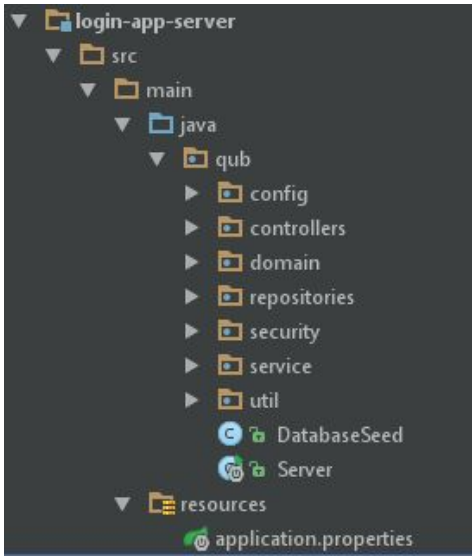



```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.3.RELEASE</version>
</parent>

<modules>
    <module>shared</module>
    <module>login-app-ui</module>
    <module>login-app-server</module>
</modules>
```

The pom file in the root of the project determines the submodules programmatically. The parent module itself extends from the spring-boot-starter-parent artifact, this allows for automatic versioning of the spring and h2 dependencies. It also provides the spring boot plugin that is used on the server, an entry point to the spring ecosystem.

1.1.4 Server module



The server module of the project consists of a spring-boot application. Spring is a java framework for many common utilities and design patterns, with tight integration with maven and it is highly configurable. The server contains all authentication logic as well as any interactions with the database.

1.1.4.1 Package Description

config
The configuration package of the server. This contains java based configuration files.

- Database connectivity
    - Supplies the connection parameters for creating a datasource bean
    - Console support - configures the url endpoint to use for viewing the embedded database.
- Authentication
    - Configures the default users of the authentication framework via the datasource
    - Configures allowed endpoints that can be accessed without being previously authenticated (such as /login and /register), all other endpoints are secured by default.

controllers
Contains the REST controllers for the application. These include:

- *AuthController*: This controller provides Login / logout endpoints. The Login endpoint is an anonymously accessible endpoint that allows the user to exchange a username and password for a signed, authentication token. The logout endpoint allows the user to formally logout, therefore invalidating the token.
- *RegisterController* : This controller provides an anonymously accessible endpoint for creating new user accounts.

- *UserController*: This controller provides an endpoint that returns details about the logged in user.

domain

The data transfer package. Contains entities for object relational mapping to the database via spring data, which is a facilitator for hibernate. This was useful for the project as no manual SQL statements had to be written to create, read, update or delete information in the database. Instead the objects for db storage are annotated with *@Entity* and used alongside a custom repository for that class. The entities in the package include a base entity common to all domain objects, a *User* entity that is inherited by *StandardUser* and *AdminUser,* as well as an issued token that is assigned to a user upon successful login.

repositories

The data access package. Contains repository interfaces that extends the CrudRepository interface provided by spring. This allows for common operations like save, find, exists, delete on the object assigned to that repository. The interface does not need to be implemented as spring will implement it at run time and the common operations can be extended to use the custom fields of the object. As such there exists a *UserRepository* and an *IssuedTokenRepository.* Repository classes are annotated with *@Transactional* so that if an error occurs on the database write the state will be rolled back to before the method call.

security

This package contains classes that contribute to the Application's authentication scheme. The two main classes here are:

- *AuthByHeaderToken:* This class provides a filter that can be hooked into the Spring request pipeline. This filter looks for a client provided authentication token that identifies the user to the Server. If a token is found, it is passed over to the Authentication service for validation.
- *AuthToken:* This class represents the AuthToken that is given to a client as an identity object after a successful login.

service

The main server logic of the application. A service is a gateway between a controller and a repository, where any intermediate logic may take place. There are three services in use:

- *UserService*: Used to wrap the functionality of the *UserRepository* and the *IssuedTokenRepository* to provide the base user interactions such as creation.
- *AuthenticationService*: A service to manage the authentication process in the application. This service implements operations such as credential validation, lockout enforcement and the creation and verification of authentication tokens.
- *CryptoHashingService:* A service to wrap the Java cryptographic operations used in the application in a single location. This service provides access to the selected Java Hashing implementation and HMAC implementation and also provides a method of verifying a string against as hash.This service also encomposes the logic required to create a secure hash.

Each service is interfaced fully and annotated with *@Service* so that dependency injection may be leveraged via the *@Autowired* annotation.

util
Utilities used in the server. A *LoginIdGenerator* exists here to generate a random number between 10,000,000 and 99,999,999. This margin ensures that there is ample scope for expansion in the database and a login id large enough to not be easily guessed.

This package also includes a custom exclusion strategy for our JSON serializer. This exclusion logic allows sensitive fields that should not be exposed to be easily excluded from the final payload.

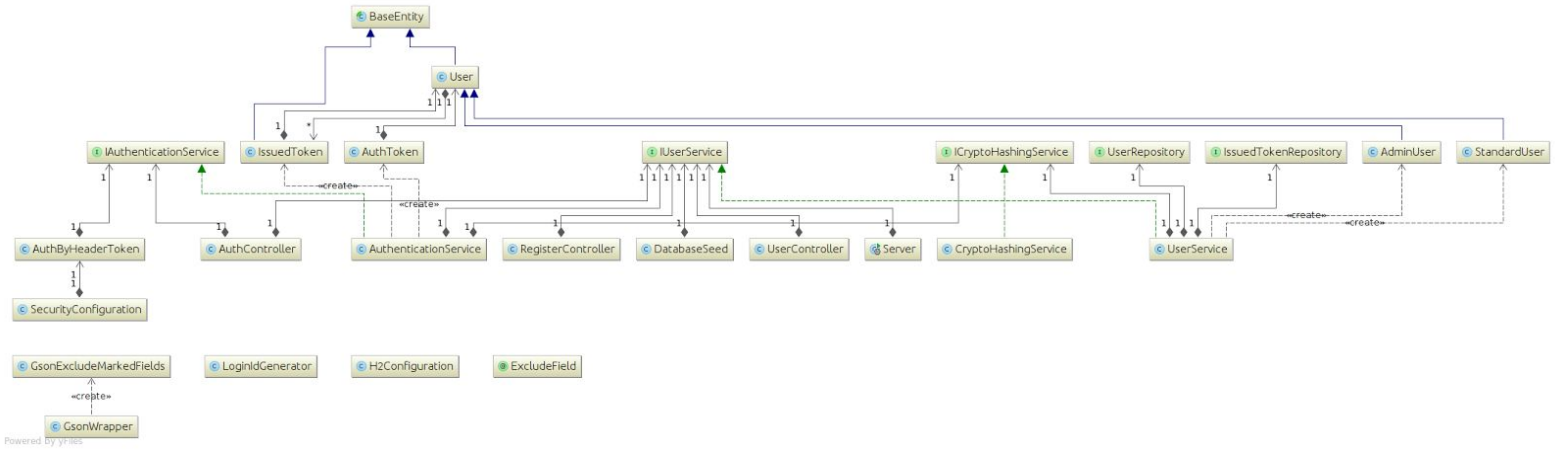
default
The default package. Contains the main class of the server - Server, as well as class to seed the default members into the database should they not already exist.

1.1.4.2 Resource Description

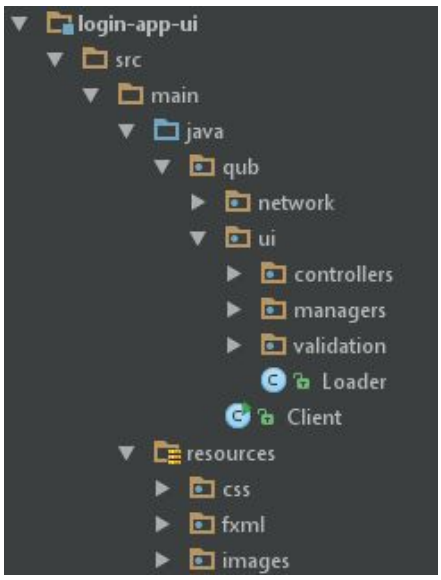
Configuration file
There is a configuration file under *resources/application.properties* in the server. This specifies hibernate dialect and update mode of the database, automatically picked up by spring via property name. SSL settings exist here also to automatically configure all endpoints to use HTTPS. The server port, certificate location and certificate password are specified here.

1.1.4.3 Server Module Diagram

Outlined below is the class diagram for the server module. A higher resolution of this as well as a more detailed version can be found on the git repo under *resources/uml.*



1.1.5 Client module



The client of module of the project consists of a JavaFX application. This is considered to be the new standard in native java ui library vs legacy code such as swing. The client contains logic to parse information from the ui and send it to the server. Validation is performed on front end fields to ensure the integrity of data.

1.1.5.1 Package Description

network
The network package is used to connect the server and the client. There is a class named *ServerConnector* that has calls to each endpoint in the various server controllers. A rest template is used to make the GET and POST requests to the defined endpoints.

ui.controllers
There exists a controller class linked to each screen of the UI, it's purpose is to control click events and handle data along with data validation. Validation is performed on all fields to ensure data integrity.

ui.managers
There is a single manager located in the managers package, *NavigationManager.* It controls the navigation flow of the ui and is passed to each ui controller.

ui.validation
A third party framework, FXValidation, was used to perform custom validation on the ui fields. As it is not available on maven it was forked from github and placed in this package. Original source at https://github.com/dukke/FXValidation

default
The main class of the client, *Client*, resides in the default package. It will start the user interface of the program.

1.1.5.2 Resource Description

The resources package of the client houses various elements linked to the user interface, these are:

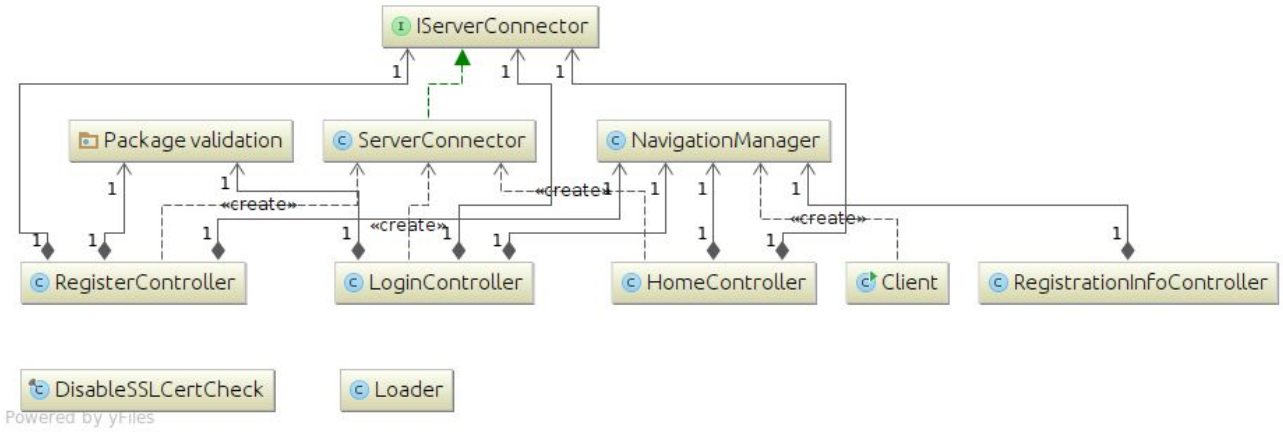
css
JavaFX is capable of integrating with css for styling. Here the background style is defined in *style.css,* the style is then applied to the parent pane of each ui screen.

fxml
JavaFX uses .fxml files to define user interface elements, similar to html. Each screen of the ui has its own fxml file and within it a reference to it's controller is stored.

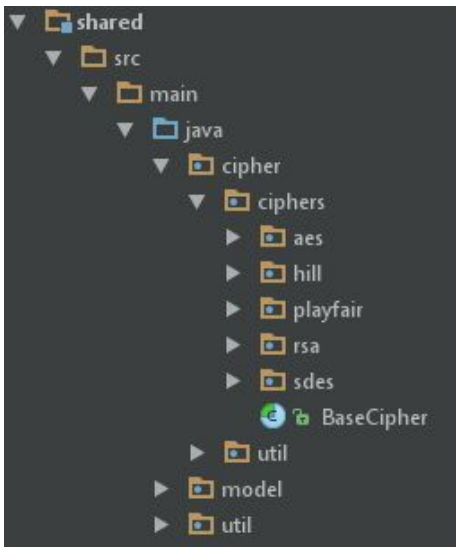
Images
The background image referenced is *style.css* is located here.

1.1.5.3 Client Module Diagram

Outlined below is the class diagram for the client module. A higher resolution of this as well as a more detailed version can be found on the git repo under *resources/uml.*



1.1.6 Shared module



The shared module is used as a bridge between the client and the server for common objects. These include the data models for communication, common utilities and the ciphers asked for.

1.1.6.1 Package Description

cipher.ciphers
The parent package of all assessment ciphers. There exists a base cipher from which all ciphers extend and individual implementations of AES, Hill, Playfair, RSA and SDES.

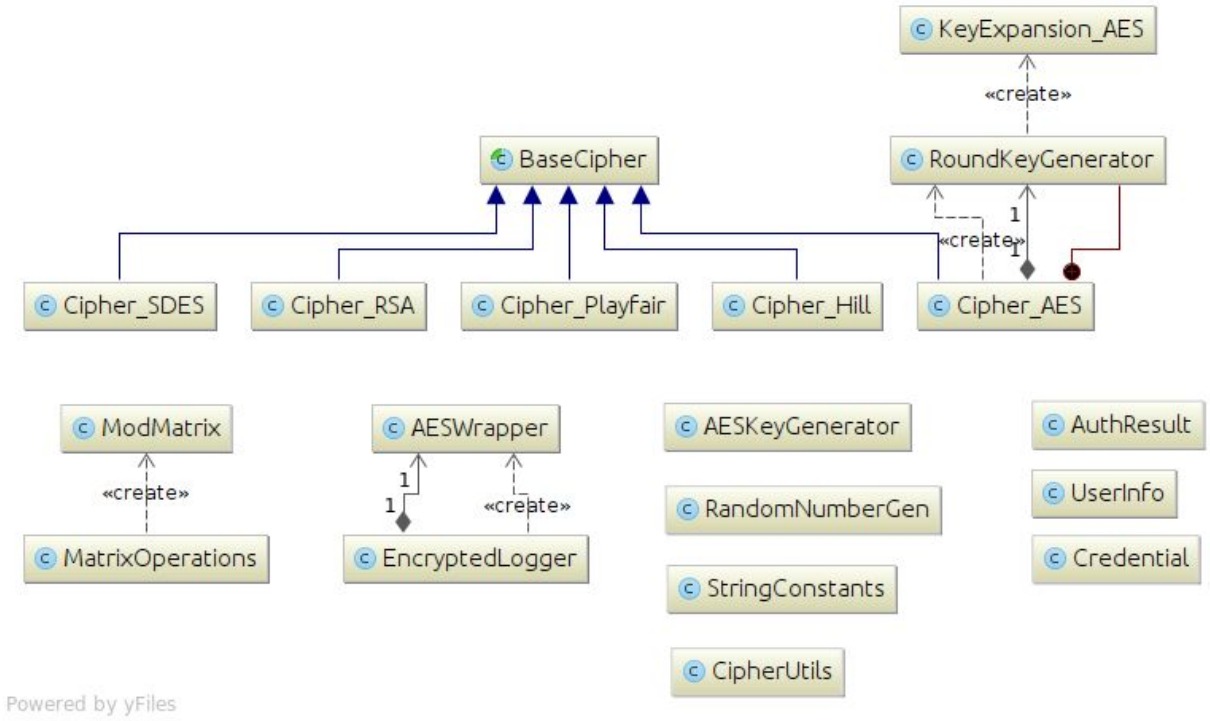
cipher.util
The utility package for ciphers. Common functionality resides here as well as a class to generate a random AES secret key.

model
The model package contains data transfer information between the client and server. These come in the form of *AuthResult* (result of authorisation request), *Credential* (login information passed to the auth controller) and *UserInfo* (a delegate user class to user when getting the current user info).

Util
The util package contains common utilities used throughout the application. This includes a *RandomNumberGenerator* (used for aes key generation and id generation), *StringConstants* and an *EncryptedLogger*. The encrypted logger wraps a standard java logger. It takes log requests, encrypts them using AES and writes them to disk. This is to monitor system activity while ensuring that no intruders could modify the log to make their activity go unnoticed.

1.1.6.2 Shared Module Diagram

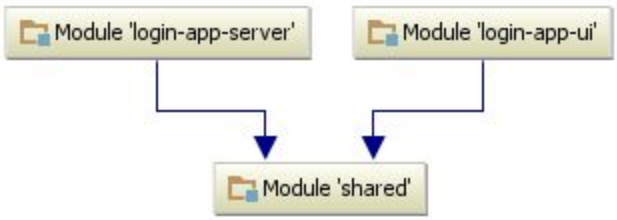
Outlined below is the class diagram for the shared module. A higher resolution of this as well as a more detailed version can be found on the git repo under *resources/uml.*



1.1.7 Project structure diagrams

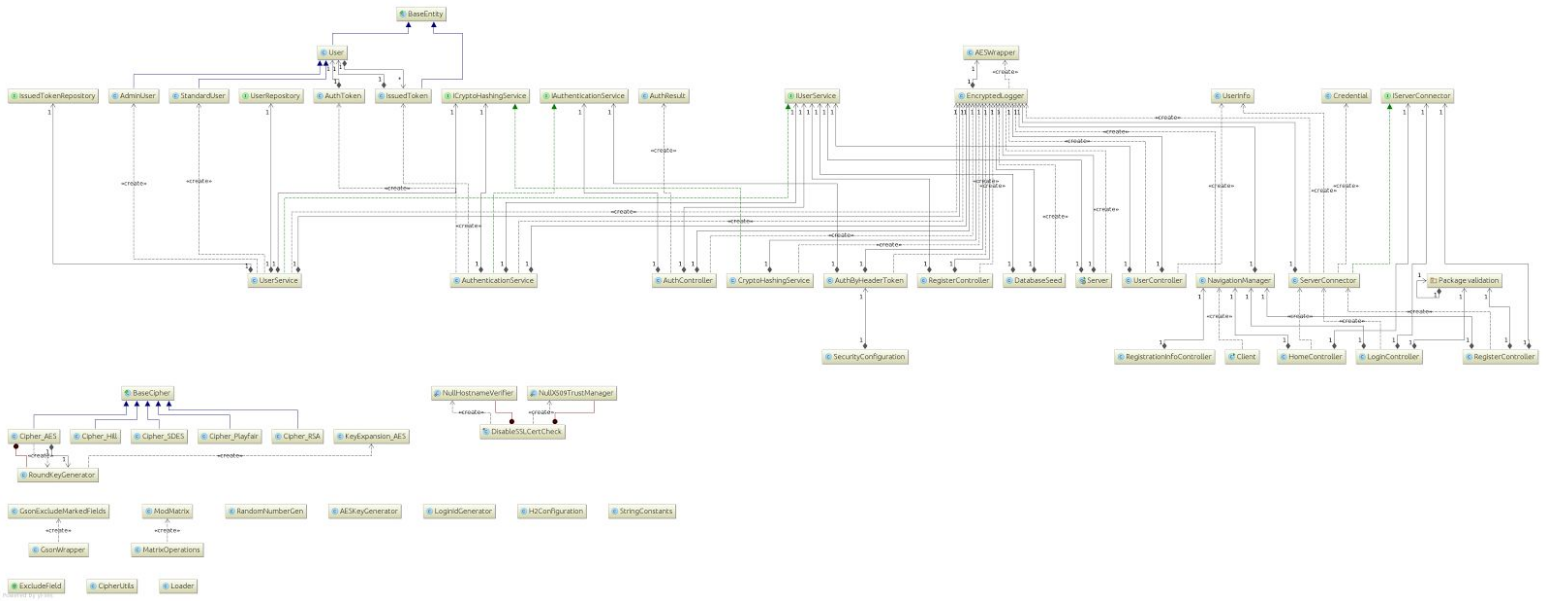
1.1.7.1 Modules

The diagram below clearly illustrates the dependency model in use between the server, client and shared module.



1.1.7.2 Classes

The diagram below shows the interaction between all components of the system. A higher resolution of this as well as a more detailed version can be found on the git repo under *resources/uml.*



1.2 Individual description of cipher algorithms

1.2.1 Hill

Input - Key : 3x3 matrix
plainText : text to encrypt (any length, adds padding)

Process- Split plainText into blocks of 3, convert chars to numbers (a = 0, b = 1... etc).
For each row in the key matrix, multiply this by the current matrix of the plainText.
, then mod 26. Convert this number back to char.

Repeat

Output - cipherText : encrypted message

The hill cipher applies a key transformation to a piece of text. The method for encrypting and decrypting text is the same, with the only difference being the key which is used (key, or InverseKey).

To encrypt we convert the string of characters into numbers, where A = 0, B = 1.

Text	: h	e	l	l	o
Conversion	: 7	4	11	11	14
Encrypted	: 8	8	23	9	15
Text	: i	i	x	j	p

Using an example key of {17, 17, 05}, {21, 18, 21}, {02, 02, 19}.

We take the first row {17, 17, 05} and the first 3 numbers of the converted text {7, 4, 11}. We then multiply the (17, 17, 05) by each element {7, 4, 11}, total the 3 results and then mod 26 this. The resulting number is then converted back to a character, this being the encrypted character

17 * 7 = 119
17 * 4 = 68 Total = 119 + 68 + 55= 374 mod 26 = 8 = i
05 * 11= 55

We then apply the same method to row 2 and 3

21* 7 = 147
18 * 4 = 72 Total = 147+ 72+ 231= 450 mod 26 = 8= i
21* 11= 231

2* 7 = 14
2* 4 = 8 Total = 14+ 8+ 209= 374 mod 26 = 23 = x

19* 11= 209

1.2.2 S-DES

Field	Purpose
int[][] sBox1	The first lookup S-Box
int[][] sBox2	The second lookup S-Box
int[] key	The 10 bit encryption key
int[] ip	The permutation to use for the initial permutation
Int[] inverse_ip	The permutation to use for inverse permutation.
Int[] p10, p8, p4	P10, p8 and p4 permutations used for keygen and function fK.
Int[] k1, k2	Used to store the k1 and k2 generated keys

K1 Key generation
First we take our predefined 10 bit key and apply our P10 permutation on this. We then split this result into 2 groups of 5 bits and shift these 1 bit left with wrap around. We finally perform P8 permutation on this to give us K1.

K2 Key generation
The generation of K2 is very similar to K1, we perform the P10 and P8 permutations in the same way as previously but the shift this time is done with 3 bits to the left instead of 1.

Initial Permutation
After we have generated K1 and K2 we move on to our initial permutation. We apply the permutation (2, 6, 3, 1, 4, 8, 5, 7) on our 8bits of plaintext and store the result.

Fk (K1)
First we take our rightmost 4 bits from our initial permutation result, we then apply the E/P permutation and store the result from this. We then XOR this result with our K1 and store this. We then move onto getting our SBox values, we input the first half of the XOR result into SBox 1 and the second half of the XOR result into SBox 2 - for getting the SBox values we use bits 1 and 4 for the row and bits 2 and 3 for the column. We then combine the result from both SBoxes, this gives us 4 bits when combined, we then perform P4 permutation on these 4 bits and store this result.

Function fK1
We take our result from Fk and perform an XOR operation with the left half of the initial permutation result that we stored earlier, we then combine this result with the right half of the

initial permutation to give us a result of 8 bits. We then take this 8 bit result and switch the left and right halves and store this result.

Fk (K2)
We take our result from function fK1 and apply the E/P permutation with the right half and store this result. We then XOR this result with K2 that we produced earlier giving us an 8 bit result. We take the left half of this result and get the SBox 1 value the same as we did in Fk(K1), and again we take the right half and get our SBox 2 value in the same way as we did in Fk(K1). We combine both SBox results to give us a 4 bit result, we then perform P4 permutation on this result.

Function fK2
We take our 4 bit result from Fk(K2) and perform an XOR operation with our left half result from Fk(K1) and store this result.

Inverse Permutation
We take our output from Function fK2 and our right half from function fK1, combine these to give us 8 bits and then perform an inverse permutation on this.

Once we have performed our inverse permutation as described we will have an output of 8 bits encrypted ciphertext.

1.2.3 AES

The RSA encryption algorithm is implemented in the Cipher_RSA class. This class exposes an Encrypt method to allow the algorithm to be ran. The input and outputs of both encryption and decryption are of type String, with another method exposed to provide the raw byte state of an encrypted String.

To support the AES algorithm, this class has a number of important fields, as shown on the table below:

Field	Purpose
String[][] DEFAULT_KEY	The default key to be used for encryption.
String[][] S_BOX	The lookup S-Box used for sub bytes.
String[][] MIX_COLUMN_MATRIX	The lookup matrix for which operation to perform in mic columns, 01, 02 or 03.
RoundKeyGenerator keyGen	Instance of an inner class. Wrapper around the key expansion class described below to extract the needed keys.
int blockSideLength	The size of the input block's rows. Used over several loops throughout the algorithm

The algorithm takes a string as input, which it then transforms into a 2D array of hexadecimal values. The input is expected to be 16 characters long to facilitate the full block size. Each inner array in the matrix represents a column of the state. The initial input state is then xor'd with the initial key used in the key expansion. This is achieved by performing a hexadecimal xor between each input in the state and each input in the key by their index in their 2D array.

The round count is extracted from the keyGen and rounds performed. Sub bytes is performed by extracting each value from the new state. For each value, the left and right hexadecimal values are extracted using substring, the values are then converted to integers to represent the row and column and a lookup done using the S-Box. The new found value then replaces the original value at the current position in the state.

Shift rows is then performed on the state. The state is inverted by swapping the row and column values of each entry to perform this operation changing it from column orientation to row orientation. Each inner array (row) is now circular shifted left by the corresponding amount for that row count. This is done using arraycopy and a shift index. At the end of the operation the state is reinverted to use column orientation again.

Mix columns is then performed on the state. For each column in the state it is converted to binary and passed to a secondary mix columns helper alongside the corresponding matrix row. For each entry in the matrix row the value is extracted, a comparison is then done using a switch statement to determine which mix column operation to perform on the corresponding column value.

For the 01 operation nothing is performed. For the 02 operation, a substring extracts the first binary bit and it's value is checked, if it is 0 the binary string is shifted left by one using a substring / concat operation. If the value is 1, the binary string is shifted left and additionally xor'd with the binary representation of the integer 27. For the 03 operation the result of the 02 operation is xor'd with the original value. After each entry in the column has been processed and stored in a temporary array, the first and second results are xor'd, that result is then xor'd with the second and that result xor'd with the final result and returned to be the new column value in the state.

After each column entry has been processed using the above the results are converted back to hex and returned as the new state.

The final stage in the round is to get the corresponding round key and xor it with the new state as was done in the first stage of the algorithm. After all rounds have been performed the resulting state is passed into the performFinalRound method. Finally this state is converted to its string representation and returned from the encrypt method.

1.2.3.1 Key Expansion

Key expansion initially takes in the key to expand it into 10 additional keys, each consisting of 32 characters (128 bits).

For each initial row in each block of 4, the row is rotated, S-Boxxed, XORd with Rcon. Rcon is generated on the fly, despite being constantly the same.

Then regardless of what row it is, it is XORd with the previous word (4 rows previous). And stored for use by the next row.

For 128 bit key the initial key is expanded to 10 additional keys which is then used by the AES algorithm

1.2.4 RSA

The RSA encryption algorithm is implemented in the Cipher_RSA class. This class exposes an Encrypt and Decrypt method to allow the algorithm to be ran.

To support the RSA algorithm, this class has a number of fields, as shown on the table below:

Field	Purpose
P	The large, prime number, P, to be used in the algorithm.
Q	The second large prime number to be used in the algorithm.
N	The value that is calculated from P*Q. N-1 equals the max value that can be encrypted.
W	The value that is calculated from (P-1)(Q-1).
D	A value that must be relatively prime to W. I.e., gcd(d,w) = 1. This value is used, with N, to form the private key.
E	A value that is calculated via the Extended Euclidian algorithm. This value forms the public key when combined with N.

To implement the algorithm, the Encryption method takes the plaintext string as an Input. The ciphertext is then returned as space separated blocks. Each step that is taken to produce this ciphertext, along with a description of how each is implemented in Java, is discussed below:

1.2.4.1 Calculation of E:

The class is provided with a value for P,Q and D. Given these values, E must be calculated before the message can be encrypted. As the value of D is selected to be relatively prime to W, it can be said that de+vw = gcd(d,w) = 1. Given this fact, E can be calculated via the Extended Euclidian Algorithm. This algorithm is implemented in the calculateEFromEuclidGcd method. In this method, A and B are divided and a remainder is worked out via R = A - Quotient / B. This method is then re-ran with A = B and B = R until R = 0. At the point where R = 0, the value of E is retrieved and returned for use in the encryption operation.

1.2.4.2 Division into numerical blocks:

RSA is a Block Cipher, therefore, the message is encrypted in individual segments of text. To support this, the message must be divided into into chunks. Alongside this, the RSA encryption method should be ran on numerical data. Therefore, an encoding scheme must be used to represent the message as a number.

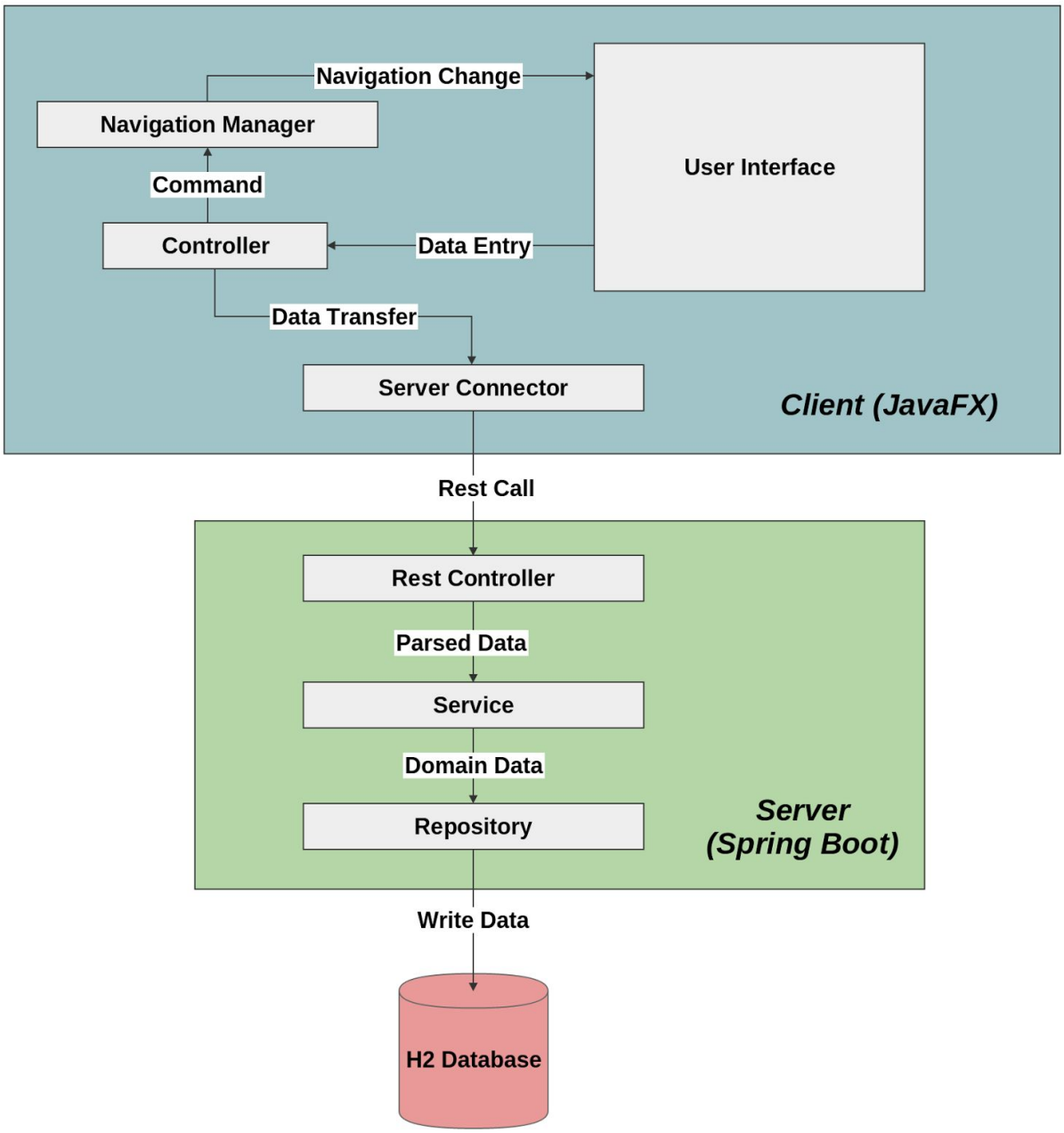
To support this, the implementation proceeds by looping over every pair of characters in the message. Within each pair, each character is converted into a number, with A = 1, B = 2 ... Z = 26. The two numbers are then combined to provide a single, numerical representation of each block. This process is completed for each pair of characters to give a set of blocks. As the max value that can be encrypted is N-1, values of P and Q have been selected to fully contain the maximum value of this encoding scheme.

1.2.4.3 Application of the Encryption Function:

Finally, to complete the encryption, each numerical block is passed through the RSA encryption function. Given the calculated N and E values, the ciphertext, C, for each block, can be calculated using $C = M^e \bmod N$. To calculate this value, Exponentiation by squaring and dividing has been implemented. This process is implemented in the applyExponentiationBySquaringDivision method. In this method, the exponent is represented as a binary string. Foreach character in the binary string, C is updated to equal $C^2 \bmod N$. If the binary character is a 1, C is then set to CM mod N.

Having completed these steps, the final ciphertext is constructed by combining the result of encrypting each block of the plaintext message.

1.3 Application Architecture Design



1.4 Security Design Justification

In designing the Client-Server login application, we have made every design decision with security in mind. Below, we have outlined a number of the decisions we have made, along with our justification for these decisions.

1.4.1 Scalability Considerations

In developing the client-server login system, we have affordable considerable capacity for the scalability of the system. Initially, the system is required to hold 4 users, based on the requirement that each member of the team must have an account. Alongside this, we must be able to register new users. Therefore, given an initial user capacity of 20, using the double comma rule, we reach a count of 4000. In our case, the only limitation to scalability is the number of User ID's that can be issued. Currently, ID's are issued from the range 10,000,000 and 99,999,999, therefore allowing considerable headroom for the scalability of the system, even beyond the initial target of 4000 users.

1.4.2 Authentication Data Security

In designing our login application, we must securely store the authentication data for users who are registered on the system. In our registration system, the main form of authentication data is the password. This is a persisted piece of authentication data. Each user account has a password that, when combined with the user ID, can be used to access the account. User passwords are stored in the database with each user's account information.

If these passwords were attacked and recovered, the accounts of our users would be exposed. Therefore, it is crucial that this data is secured. To achieve this, a password hashing strategy has been used. The strategy chosen must provide a strong level of protection against brute force attacks from ever faster machines and rainbow tables. Given this, it should also offer a strong level of complexity that is safe, yet doesn't impact the experience of the user.

Given these constraints, we have implemented password security by means of the Java provided implementation of PBKDF2. This algorithm provides a way of generating a one-way, secure password hash that cannot be feasibly reversed. This algorithm provides the following benefits over weaker algorithms such as SHA or MD5:

- This algorithm allows for an Iteration count. As computers become more powerful, the number of iterations can be continuously increased to increase the complexity of the algorithm, therefore strengthening the algorithm against brute force attacks. Currently, our application uses 10000 iterations.
- This algorithm uses a Salt that is mixed into the hashing process. In our case, a salt value is randomly generated for each password being stored. This salt value ensures that even matching passwords produce a completely different hash, therefore strengthening the approach against attacks such as rainbow tables where hashes are pre-computed.

1.4.3 Use of Java Framework Encryption implementations

At the core of our secure login application is the choice and strength of the encryption, hashing and HMAC algorithms employed. For our system to be secure, it is crucial that these algorithms and their implementations are proven to be secure in all respects. Alongside this, the algorithms employed must be reliable in terms of their ability to consistently deal with all types of input.

Although we have implemented our own encryption algorithms for Part 1 of this assignment, given the importance of this requirement, we chose to employ the algorithms provided within the Java framework. Given the wide range, industry accepted validation and verification of these implementations, we are confident that these implementations can better guarantee the confidentiality and integrity of our user data.

1.4.4 Authentication

The main issues of this system include the following:

- *Identity*: The process of unique tying a single marker to a single person.
- *Authentication:* The process of securely and reliably verifying that a person is who they say they are.
- *Authorization:* The process of determining what resources a given user can access.

In our system, these problems are addressed by our Authentication system. This key component of the application provides the means by which a user can validate their identity, this component provides a multi-factor solution by which users can securely login, be associated to a role and access resources with a single, secure login token.

For our system, we chose to implement a token based authentication approach. This authentication system follows the flow outlined below:

- The user provides their unique user ID and password, from a specific client on a given IP Address.
- The system verifies the credentials by checking the username against the user store and by verifying that the password provided hashes, with the original salt, to match the stored hash.
- Once the credentials have been verified, an AuthToken is generated for the user. This token contains the user's information, alongside an expiry date and the IP address from which the user logged in.
- Finally, a HMAC, or as hashed checksum, is generated for the token using a private key that is safely stored on the server. This token is then issued to the client as their identity object.
- Using this token, the client can access secured resources on the server. The token is re-sent to the server on each future request. The server then verifies the identity of the client by ensuring that the HMAC hash of the token matches the hash at the end of the token.

As a result of this, if the token changes, or the signature is changed, or the expiry date is reached or the IP Address of the request is changed or if the user logs out, therefore invalidating the token on the server side, the verification will fail and the user will be denied

access. Overall, this solution provides a method of ensuring the identity of the user is true, therefore ensuring the continued confidentiality of the private information.

1.4.5 ID Generation

When securing a system, it is crucial that the user can be identified. Identification is the process of assigning a unique stamp to each user of the system. This is a crucial first step in the implementation of access controls. When selecting an identifier, it is crucial that the identifier is unique, non-descriptive and issued securely. Given this, we have decided to use a randomly generated numeric ID as our user identifier. Our system randomly assigns an available ID in the range 10,000,000 and 99,999,999, therefore allowing for a substantial number of users. This ID is then securely issued over the uniquely encrypted connection between the client and server.

1.4.6 Field Validation

Each field on the application is validated to ensure that the information entered is correct and conforms to our standards. We have required fields on both the login and registration view to ensure no null data is transmitted. If any of the fields do not meet the requirements then a validation error will be displayed to the user. This preventative access control technique ensures that the data collected by the system is full, complete and valid, therefore ensuring that forged data that could cause the system to become unavailable or that could attempt to expose secured data is blocked at the point of entry.

1.4.7 Password Policy

Passwords must:

- Be at least 8 characters long
- Contain at least 1 lowercase character
- Contain at least 1 uppercase character
- Contain at least 1 number
- Contain at least 1 special character out of @#$%!£%&

All of the above ensure that each user's password is strong by enforcing this policy on account creation.

1.4.8 Account Lockout Policy

We have enforced an account lockout policy for this system. If someone tries to gain access to a certain account 3 times unsuccessfully, then the account will be locked for 5 minutes, in which time even if the correct details are input the account will be denied access. This will be logged.

1.4.9 Deterrent Message

We have an acceptable usage policy which is linked on both the login and registration screens. This usage policy outlines our policy for users of the application and by them using the application means that they consent to our policy. This access control, which falls into the Deterrent category, informs users that any unlawful activity will be reported and criminally prosecuted.

1.4.10 Encrypted Log Storage & Auditability

One of the key requirements of access control is that the actions of the system must be auditable - that is, every action carried out by the system should be traceable. This way, if there is a system breach, the data can be used to reconstruct the chain of events that led to the attack.

To this end, we have designed our system to audit every operation. Logging is used throughout the system at every main call so as to provide a clear audit trail for all operations. This include successful and unsuccessful authentication attempts and new account registration attempts.

Alongside this auditability requirement, as the logs store crucial information related to the security of the system, it is crucial that these are secured as-well. So as to meet this requirement, we have added an encryption layer to the log. Using this component, log messages are encrypted right at the point of being written to the log. This way, potential attackers cannot modify the log, therefore ensuring the safety of this data in case of an attack.

1.4.11 SSL

This application must ensure the confidentially and integrity of all communications between the client and the server. This means that private information must be kept secure in transit and that we must be able to prevent attacks such as “man-in-the-middle” attacks where user data could be modified in transit.

So as to ensure that these requirements are met, we have employed SSL in the transport layer of the client-server application. SSL, standing for Secure Socket Layers, provides a means of confidentiality and integrity at transport layer. Through the use of SSL, encryption keys are securely exchanged between the client and server on a per-request basis. This way, each client has a uniquely encrypted connection. Through the use of these key pairs, the server can also verify that the message has come from the client.

For these reasons, SSL has been employed in our design. SSL is used on all requests between the client and server to ensure communications are kept private and to ensure that the client is in fact the sender.

2 Appendix

2.1 Cipher algorithms

2.1.1 Hill

```
public class Cipher_Hill extends BaseCipher {

   private static int matrixDimension = 3;

   private static int[][] key = new int[matrixDimension][matrixDimension];
   private static int[][] keyInverse = new int[matrixDimension][matrixDimension];

   private static int blockPos = 0;
   private static int[] block = new int[matrixDimension];
   private static int[] blockSum = new int[matrixDimension];

   /**
    * Default constructor for Hill cipher using assignment provided key
    *
    */
   public Cipher_Hill() {
       int[][] key = {
           {15, 10, 29},
           {8, 17, 23},
           {38, 13, 5}
       };
       setKey(key);
   }

   /**
    * Custom constructor for Hill cipher using parameter provided key
    *
    * @param key is the key provided for the cipher
    */
   public Cipher_Hill(int[][] key) {
       setKey(key);
   }

   /**
    * Sets the key for the Hill cipher using the passed param key
    *
    * @param key is the key provided for the cipher
    */
   public void setKey(int[][] key) {
       Cipher_Hill.key = key;
       setKeyInverse(key);
   }

   /**
    * Sets the inverse key for the Hill cipher using the passed param key
```

```
     *
     * @param key is the key provided for the cipher
     */
    public void setKeyInverse(int[][] key) {
        try {
            Cipher_Hill.keyInverse = MatrixOperations.inverse(key);
        } catch (Exception e) {
            logMessage("Cannot invert key. " + Arrays.deepToString(key));
            Cipher_Hill.keyInverse = null;
        }
    }

    /**
     * gets the matrixDimension
     *
     * @return
     */
    public int getMatrixDimension() {
        return matrixDimension;
    }

    /**
     * Sets the matric dimension
     *
     * @param matrixDimension the square size of the matrix
     */
    public void setMatrixDimension(int matrixDimension) {
        Cipher_Hill.matrixDimension = matrixDimension;
    }

    /**
     * Method to decrypt encrypted text
     *
     * @param cipherText is the text to be decoded
     * @return returns the unencoded cipherText
     */
    @Override
    public String decrypt(String cipherText) {
        if (Cipher_Hill.keyInverse == null) {
            return "Cannot decrypt as the provided key is not invertible yet";
        } else {
            return process(cipherText, keyInverse, "Encrypted cipherText");
        }
    }

    /**
     * Method to encrypt plaint text
     *
     * @param plainText is the text to be encrypted
     * @return returns the encrypts cipherText
     */
    @Override
    public String encrypt(String plainText) {
        return process(plainText, key, "Encrypting plainText");
    }
```

```java
/**
 * Method is used to apply a key cipher to the text
 *
 * @param text is the String on which the key will be applied to
 * @param key is either the key or the inverse key,depends on encrypt/decrpt
 * @param conversion is for providing which way the conversion is being done
 * @return the encrypted/decrypted text
 */
public String process(String text, int[][] key, String conversion) {
    String result = "";
    ArrayList<Integer> spacePositions = new ArrayList<>();

    int initalTextLength = text.length();
    int additional = 0;

    char chr;

    //count spaces
    for (int pos1 = 0; pos1 < text.length(); pos1++) {
        chr = text.charAt(pos1);
        if (chr != ' ') {
            additional++;
        } else {
            //if it is a space then note the position for later
            spacePositions.add(pos1);
        }
    }
    //remove spaces
    text = text.replaceAll("\\s","");

    //padding
    while ((text.length() + additional) % matrixDimension != 0) {
        text = text + "a";
        additional++;
    }

    block = new int[matrixDimension];
    blockSum = new int[matrixDimension];

    for (int pos = 0; pos < text.length(); pos++) {

        //for each char in the plainText
        chr = text.charAt(pos);

        block[blockPos] = CipherUtils.charToDigit(chr);
        blockPos++;

        //if the block/matrix is full
        if (blockPos == matrixDimension) {
            blockPos = 0;

            //for each row in the key
            for (int vert = 0; vert < matrixDimension; vert++) {
                //multiply it by the block/matrix and sum the results
                for (int i = 0; i < matrixDimension; i++) {
                    blockSum[vert] += key[vert][i] * block[i];
```

```
                    }
                    //mod the total
                    blockSum[vert] = blockSum[vert] % 26;
                }

                //convert the modded digit to a char and add to cipherText
                for (int i = 0; i < matrixDimension; i++) {
                    result += CipherUtils.digitToChar(blockSum[i]);
                    blockSum[i] = 0;
                }
            }
        }

        //add spaces back into the cipher
        for (Integer spacePosition : spacePositions) {
            result = result.substring(0, spacePosition) + " " +
result.substring(spacePosition, result.length());
            text = text.substring(0, spacePosition) + " " +
text.substring(spacePosition, text.length());
        }

        System.out.println("result = " + result);

        result = result.substring(0, initalTextLength);
        text = text.substring(0, initalTextLength);

        log(text, result, conversion);

        return result;

    }

}

public class MatrixOperations {

    public static int[][] inverse(int[][] key) {

        BigInteger[][] keyInverse = new BigInteger[3][3];
        int[][] keyInverseInt = new int[3][3];

        for (int i = 0; i < key.length; i++) {
            for (int j = 0; j < key.length; j++) {
                keyInverse[i][j] = BigInteger.valueOf(key[i][j]);
            }
        }

        ModMatrix obj2 = new ModMatrix(keyInverse);
        ModMatrix inverse2 = obj2.inverse(obj2);

        keyInverse = inverse2.getData();

        for (int i = 0; i < key.length; i++) {
            for (int j = 0; j < key.length; j++) {
                keyInverseInt[i][j] = keyInverse[i][j].intValue();
            }
```

```
        }

        return keyInverseInt;

    }

}

//Source github.com/PraAnj/Modular-Matrix-Inverse-Java

public class ModMatrix {

    private int nrows;
    private int ncols;
    private BigInteger[][] data;
    private final BigInteger mod = new BigInteger("26");

    public ModMatrix(BigInteger[][] dat) {
        this.data = dat;
        this.nrows = dat.length;
        this.ncols = dat[0].length;
    }

    public ModMatrix(int nrow, int ncol) {
        this.nrows = nrow;
        this.ncols = ncol;
        data = new BigInteger[nrow][ncol];
    }

    public int getNrows() {
        return nrows;
    }

    public void setNrows(int nrows) {
        this.nrows = nrows;
    }

    public int getNcols() {
        return ncols;
    }

    public void setNcols(int ncols) {
        this.ncols = ncols;
    }

    public BigInteger[][] getData() {
        return data;
    }

    public void setData(BigInteger[][] data) {
        this.data = data;
    }

    public BigInteger getValueAt(int i, int j) {
        return data[i][j];
    }
```

```
    public void setValueAt(int i, int j, BigInteger value) {
        data[i][j] = value;
    }

    public int size() {
        return ncols;
    }

    // Take the transpose of the Matrix..
    public static ModMatrix transpose(ModMatrix matrix) {
        ModMatrix transposedMatrix = new ModMatrix(matrix.getNcols(),
matrix.getNrows());
        for (int i = 0; i < matrix.getNrows(); i++) {
            for (int j = 0; j < matrix.getNcols(); j++) {
                transposedMatrix.setValueAt(j, i, matrix.getValueAt(i, j));
            }
        }
        return transposedMatrix;
    }

    // All operations are using Big Integers
    public static BigInteger determinant(ModMatrix matrix) {

        if (matrix.size() == 1) {
            return matrix.getValueAt(0, 0);
        }
        if (matrix.size() == 2) {
            //return (matrix.getValueAt(0, 0) * matrix.getValueAt(1, 1)) -
(matrix.getValueAt(0, 1) * matrix.getValueAt(1, 0));
            return (matrix.getValueAt(0, 0).multiply(matrix.getValueAt(1,
1))).subtract((matrix.getValueAt(0, 1).multiply(matrix.getValueAt(1, 0))));
        }
        BigInteger sum = new BigInteger("0");
        for (int i = 0; i < matrix.getNcols(); i++) {
            sum = sum.add(changeSign(i).multiply(matrix.getValueAt(0,
i).multiply(determinant(createSubMatrix(matrix, 0, i)))));
        }
        return sum;
    }

    private static BigInteger changeSign(int i) {
        if (i % 2 == 0) {
            return new BigInteger("1");
        } else {
            return new BigInteger("-1");
        }
    }

    public static ModMatrix createSubMatrix(ModMatrix matrix, int excluding_row, int
excluding_col) {
        ModMatrix mat = new ModMatrix(matrix.getNrows() - 1, matrix.getNcols() - 1);
        int r = -1;
        for (int i = 0; i < matrix.getNrows(); i++) {
            if (i == excluding_row) {
                continue;
```

```
            }
            r++;
            int c = -1;
            for (int j = 0; j < matrix.getNcols(); j++) {
                if (j == excluding_col) {
                    continue;
                }
                mat.setValueAt(r, ++c, matrix.getValueAt(i, j));
            }
        }
        return mat;
    }

    public ModMatrix cofactor(ModMatrix matrix) {
        ModMatrix mat = new ModMatrix(matrix.getNrows(), matrix.getNcols());
        for (int i = 0; i < matrix.getNrows(); i++) {
            for (int j = 0; j < matrix.getNcols(); j++) {
                mat.setValueAt(i, j,
(changeSign(i).multiply(changeSign(j)).multiply(determinant(createSubMatrix(matrix, i,
j)))).mod(mod));
            }
        }

        return mat;
    }

    public ModMatrix inverse(ModMatrix matrix) {
        return (transpose(cofactor(matrix)).dc(determinant(matrix)));
    }

    private ModMatrix dc(BigInteger d) {
        BigInteger inv = d.modInverse(mod);
        for (int i = 0; i < nrows; i++) {
            for (int j = 0; j < ncols; j++) {
                data[i][j] = (data[i][j].multiply(inv)).mod(mod);
            }
        }
        return this;
    }
}

//Source github.com/PraAnj/Modular-Matrix-Inverse-Java
```

2.1.2 S-DES

```
public class Cipher_SDES extends BaseCipher {

  private int[] plainText = {1,1,0,0,1,0,0,1};
  private int[] cipherText = {1,0,0,0,1,1,1,1};
  private int[] key = {1,1,1,1,0,1,1,0,0,0};
  private int[] ip = {2, 6, 3, 1, 4, 8, 5, 7};
  private int[] ep = {4, 1, 2, 3, 2, 3, 4, 1};
  private int[] inverse_ip = {4, 1, 3, 5, 7, 2, 8, 6};
  private int[] p10 = {3, 5, 2, 7, 4, 10, 1, 9, 8, 6};
  private int[] p8 = {6, 3, 7, 4, 8, 5, 10, 9};
  private int[] p4 = {2, 4, 3, 1};

  private final int[][] sBox1 = {  {1, 0, 3, 2},
        {3, 2, 1, 0},
        {0, 2, 1, 3},
        {3, 1, 3, 2} };

  private final int[][] sBox2 = {  {0, 1, 2, 3},
        {2, 0, 1, 3},
        {3, 0, 1, 0},
        {2, 1, 0, 3} };

  private int[] temp10 = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 };
  private int[] p4Result = { -1, -1, -1, -1 };
  private int[] f1Result = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 };
  private int[] f2Result = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 };
  private int[] k1 = { -1, -1, -1, -1, -1, -1, -1, -1 };
  private int[] k2 = { -1, -1, -1, -1, -1, -1, -1, -1 };
  private int[] IPP = { -1, -1, -1, -1, -1, -1, -1, -1 };
  private int[] left_nibble = {-1, -1, -1, -1};
  private int[] right_nibble = {-1, -1, -1, -1};

  public ArrayList<int[]> encryptWord(String plainText)
  {
     ArrayList<int[]> encrypted = new ArrayList<>();
     String temp;
     int[] input = new int[8];

     byte[] b = plainText.getBytes(StandardCharsets.US_ASCII);
     for (byte currByte:b) {
        temp = String.format("%8s", Integer.toBinaryString(currByte)).replace(' ', '0');
        int count = 0;
        for (char ch: temp.toCharArray()) {
           input[count] = Integer.parseInt(Character.toString(ch));
           count++;
        }
        encrypted.add(encrypt(input));
     }
     return encrypted;
  }

  /**
   * Method to decrypt a binary ciper input
```

```
     * @param cipherText
     * @return
     */
    public String decryptWord(List<int[]> cipherText)
    {
        String word = "";
        for(int i = 0; i < cipherText.size(); i++){
            int [] temp = decrypt(cipherText.get(i));
            String temp2  = Arrays.toString(temp).replace(", ", "");
            temp2 = temp2.replaceAll("\\[", "").replaceAll("\\]","");

            int charCode = Integer.parseInt(temp2, 2);
            word += new Character((char)charCode).toString();
        }

        return word;
    }

    /**
     * Encryption method
     * @param input
     * @return
     */
    public int[] encrypt(int[] input) {
        plainText = input;
        keyGeneration();
        initialPermutation();
        f1Result = kFunction(k1);
        f1Result = functionK1(f1Result);

        //swap module stage
        System.arraycopy(f1Result, 0, right_nibble, 0, f1Result.length / 2);
        System.arraycopy(f1Result, 4, left_nibble, 0, f1Result.length / 2);

        for (int i = 4; i < IPP.length; i++) {
            IPP[i] = right_nibble[i-4]; //+4 to give us right most half
        }

        f2Result = kFunction(k2);
        f2Result = functionK2(f2Result);
        f2Result = inversePermutation(f2Result);

        return f2Result;
    }

    /**
     *
     * @param plaintext
     * @return
     */
    @Override
    public String encrypt(String plaintext) {
        String output;
        ArrayList<int[]> binaryValues = encryptWord(plaintext);

        output = convertBinaryArraysToBinaryString(binaryValues);
```

```
    logEncryption(plaintext, output);
    return output;
}

/**
 *
 * @param encryptedText
 * @return
 */
@Override
public String decrypt(String encryptedText) {
    List<int[]> binaryEntries = convertBinaryStringToBinaryArrays(encryptedText);

    String output = decryptWord(binaryEntries);

    logDecryption(encryptedText, output);
    return output;
}

/**
 * Decryption method
 * @param cText
 * @return
 */
public int[] decrypt(int[] cText) {
    cipherText = cText;
    applyPermutation(cipherText, ip, IPP);
    keyGeneration();
    f2Result = kFunction(k2);
    f2Result = functionK1(f2Result);
    System.arraycopy(f2Result, 0, right_nibble, 0, f2Result.length / 2);
    System.arraycopy(f2Result, 4, left_nibble, 0, f2Result.length / 2);

    for (int i = 4; i < IPP.length; i++) {
        IPP[i] = right_nibble[i-4]; //+4 to give us right most half
    }
    f1Result = kFunction(k1);
    f1Result = functionK2(f1Result);
    f1Result = inversePermutation(f1Result);

    return f1Result;
}

/**
 * Method to generation k1 & k2
 */
private void keyGeneration(){
    //take key and apply p10 permutation, store in temp10
    applyPermutation(key, p10, temp10);

    //then split into 2 groups of 5 bits and shift left 1 bit with wrap round
    shiftArrayElementsLeft(temp10, 0, 5, 1);
    shiftArrayElementsLeft(temp10, 5, 5, 1);

    //take temp and apply p8 permutation, store in key1
```

```
        applyPermutation(temp10, p8, k1);

        //k2
        //take key and apply p10 permutation, store in temp10
        applyPermutation(key, p10, temp10);

        //then split into 2 groups of 5 bits and shift left 3 bit with wrap round
        shiftArrayElementsLeft(temp10, 0, 5, 3);
        shiftArrayElementsLeft(temp10, 5, 5, 3);

        //take temp and apply p8 permutation, store in key 2
        applyPermutation(temp10, p8, k2);
    }

    /**
     * The initial Permutation
     */
    private void initialPermutation(){
        applyPermutation(plainText, ip, IPP);
    }

    /**
     * Method for k function for both rounds
     * @param key
     * @return
     */
    private int[] kFunction(int[] key){
        int[] rightIPP = new int[4];
        int[] temp1 = new int[8];
        int[] temp2 = new int[8];
        int[] sboxCombined = new int[4];

        for (int i = 0; i < rightIPP.length; i++) {
            rightIPP[i] = IPP[i + 4]; //+4 to give us right most half
        }

        //apply ep to right nibble, store in temp1 *only right most half*
        applyPermutation(rightIPP, ep, temp1);

        //XOR with key
        XOR(temp1, key, temp1);

        System.arraycopy(temp1, 4, temp2, 0, temp1.length / 2);
        int sbox1 = getSboxValues(sBox1,temp1);
        int sbox2 = getSboxValues(sBox2, temp2);

        String sboxBinary = String.format("%2s", Integer.toBinaryString(sbox1)).replace(' ', '0');
        sboxBinary += String.format("%2s", Integer.toBinaryString(sbox2)).replace(' ', '0');


        //combine the sbox values into array
        int count = 0;
        for (char ch: sboxBinary.toCharArray()) {
            sboxCombined[count] = Integer.parseInt(Character.toString(ch));
            count++;
        }
```

```
        //take result from sboxes and apply p4 permutation
        applyPermutation(sboxCombined, p4, p4Result);
        System.arraycopy(p4Result, 0, sboxCombined, 0, p4Result.length);
        return sboxCombined;
    }

    /**
     * Second part of round 1
     * @param sboxCombined
     * @return
     */
    private int[] functionK1(int[] sboxCombined){
        for (int i = 0; i < sboxCombined.length; i++) {
            sboxCombined[i] = (sboxCombined[i] ^ IPP[i]); //left half of initial permutation and XOR with p4result
        }

        int[] temp = new int [4];
        for (int i = 4; i < IPP.length; i++) {
            temp[i - 4] = IPP[i]; //take right half of initial permutation
        }

        int [] result = new int[8];
        System.arraycopy(sboxCombined, 0, result, 0, sboxCombined.length);
        System.arraycopy(temp, 0, result, 4, temp.length);

        return result;
    }

    /**
     * Second part of round 2
     * @param sboxCombined
     * @return
     */
    private int[] functionK2(int[] sboxCombined){
        for (int i = 0; i < sboxCombined.length; i++) {
            sboxCombined[i] = (sboxCombined[i] ^ left_nibble[i]); //left half of initial permutation and XOR with p4result
        }

        int[] combinedHalves = new int[8];
        System.arraycopy(sboxCombined, 0, combinedHalves, 0, sboxCombined.length);
        System.arraycopy(right_nibble, 0, combinedHalves, 4, right_nibble.length); //combine the XOR of above and
right nibble from earlier

        //Now inverse permutation of the combined halves
        return combinedHalves;
    }

    /**
     * Method to get the sbox values in the given row and column
     * @param sBox
     * @param rolCol
     * @return
     */
    private int getSboxValues(int[][] sBox, int[] rolCol){
        //First get sbox row
```

```
        String row = Integer.toString(rolCol[0]);
        row += Integer.toString(rolCol[3]);

        //Then get sbox column
        String col = Integer.toString(rolCol[1]);
        col += Integer.toString(rolCol[2]);

        int sboxCol = Integer.parseInt(col, 2);
        int sboxRow = Integer.parseInt(row, 2);
        return sBox[sboxRow][sboxCol];
    }

    /**
     * Method to apply the inverse permutation
     * @param array
     * @return
     */
    private int [] inversePermutation(int[] array){
        int[] destArray = new int[8];
        applyPermutation(array, inverse_ip, destArray);
        return destArray;
    }

    /**
     * Method to apply a give permutation
     * @param source
     * @param permutation
     * @param destination
     */
    private void applyPermutation(int[] source, int[] permutation, int[] destination) {
        for (int i = 0; i < permutation.length; i++) {
            destination[i] = source[permutation[i] - 1];
        }
    }

    /**
     * Method to shift array elements left
     * @param array
     * @param startElement
     * @param numElement
     * @param shiftLeftAmount
     */
    private void shiftArrayElementsLeft(int[] array, int startElement, int numElement, int shiftLeftAmount) {
        int[] arrayCopy = new int[array.length];
        System.arraycopy(array, 0, arrayCopy, 0, array.length);

        for (int i = startElement; i < startElement + numElement; i++) {
            array[i] = arrayCopy[((i + shiftLeftAmount) % 5) + startElement];
        }
    }

    /**
     * Method to perform XOR
     * @param input1
     * @param input2
     * @param destination
```

```
     * @return
     */
    private int[] XOR(int[] input1, int[] input2, int[] destination) {
        for (int i = 0; i < input1.length; i++) {
            destination[i] = (input1[i] ^ input2[i]);
        }
        return destination;
    }

    /**
     * Convert a binary strings representation from string to array.
     * It is assumed that each binary entry in the string is seperated by a " ".
     * @param binaryStrings The string containing many binary strings.
     * @return              The array representation.
     */
    private List<int[]> convertBinaryStringToBinaryArrays(String binaryStrings) {
        String[] words = binaryStrings.split("\\s+");

        List<int[]> binaryEntries = new ArrayList<>();
        int[] binaryEntry;

        // For all the binary strings
        for (String binaryString : words) {
            binaryEntry = new int[8];

            // For each character in the string - extract int value
            for (int i = 0; i < binaryString.length(); i++){
                String character = String.valueOf(binaryString.charAt(i));
                binaryEntry[i] = Integer.valueOf(character);
            }

            binaryEntries.add(binaryEntry);
        }

        return binaryEntries;
    }

    /**
     * Convert a list of binary array entries (where 1 array = 8 binary bits)
     * to plaintext binary strings
     * @param binaryArrays  The array's of binary
     * @return              The binary string
     */
    private String convertBinaryArraysToBinaryString(ArrayList<int[]> binaryArrays) {

        String binaryStrings = "";

        for (int[] binaryEntry : binaryArrays) {

            // Convert binary array to string
            String binaryString = "";
            for (int bit : binaryEntry) {
                binaryString = binaryString.concat(String.valueOf(bit));
            }

            // Concatenate the result
```

```
            binaryStrings = binaryStrings.concat(binaryString + " ");
        }

        return binaryStrings;
    }
}
```

2.1.3 AES

```
/**
* Cipher implementation for the AES algorithm.
*/
public class Cipher_AES extends BaseCipher {

    //region Fields

    static final String[][] DEFAULT_KEY = { {"2b", "7e", "15", "16"},
                                            {"28", "ae", "d2", "a6"},
                                            {"ab", "f7", "15", "88"},
                                            {"09", "cf", "4f", "3c"} };

    /**
     * S-Box used for sub bytes.
     * - Rijndael S-box.
     * - http://en.wikipedia.org/wiki/Rijndael_S-box
     */
    static final String S_BOX[][] = {   {"63", "7c", "77", "7b", "f2", "6b", "6f", "c5", "30", "01", "67", "2b", "fe", "d7",
"ab", "76"},
                                        {"ca", "82", "c9", "7d", "fa", "59", "47", "f0", "ad", "d4", "a2", "af", "9c", "a4", "72",
"c0"},
                                        {"b7", "fd", "93", "26", "36", "3f", "f7", "cc", "34", "a5", "e5", "f1", "71", "d8", "31",
"15"},
                                        {"04", "c7", "23", "c3", "18", "96", "05", "9a", "07", "12", "80", "e2", "eb", "27", "b2",
"75"},
                                        {"09", "83", "2c", "1a", "1b", "6e", "5a", "a0", "52", "3b", "d6", "b3", "29", "e3", "2f",
"84"},
                                        {"53", "d1", "00", "ed", "20", "fc", "b1", "5b", "6a", "cb", "be", "39", "4a", "4c", "58",
"cf"},
                                        {"d0", "ef", "aa", "fb", "43", "4d", "33", "85", "45", "f9", "02", "7f", "50", "3c", "9f",
"a8"},
                                        {"51", "a3", "40", "8f", "92", "9d", "38", "f5", "bc", "b6", "da", "21", "10", "ff", "f3",
"d2"},
                                        {"cd", "0c", "13", "ec", "5f", "97", "44", "17", "c4", "a7", "7e", "3d", "64", "5d", "19",
"73"},
                                        {"60", "81", "4f", "dc", "22", "2a", "90", "88", "46", "ee", "b8", "14", "de", "5e", "0b",
"db"},
                                        {"e0", "32", "3a", "0a", "49", "06", "24", "5c", "c2", "d3", "ac", "62", "91", "95", "e4",
"79"},
                                        {"e7", "c8", "37", "6d", "8d", "d5", "4e", "a9", "6c", "56", "f4", "ea", "65", "7a", "ae",
"08"},
                                        {"ba", "78", "25", "2e", "1c", "a6", "b4", "c6", "e8", "dd", "74", "1f", "4b", "bd", "8b",
"8a"},
                                        {"70", "3e", "b5", "66", "48", "03", "f6", "0e", "61", "35", "57", "b9", "86", "c1", "1d",
"9e"},
                                        {"e1", "f8", "98", "11", "69", "d9", "8e", "94", "9b", "1e", "87", "e9", "ce", "55", "28",
"df"},
```

```
                        {"8c", "a1", "89", "0d", "bf", "e6", "42", "68", "41", "99", "2d", "0f", "b0", "54", "bb",
"16"} };

    /**
     * Matrix used for mix columns.
     * - Galois Field
     */
    private final String[][] MIX_COLUMN_MATRIX = {  {"2", "3", "1", "1"},
                                                    {"1", "2", "3", "1"},
                                                    {"1", "1", "2", "3"},
                                                    {"3", "1", "1", "2"} };

    private final RoundKeyGenerator keyGen;

    private int blockSideLength = 0;

    private String[][] previousOutputState;
    private String[][] key;

    //endregion

    //region Constructors
    /**
     * Constructor - Use provided key.
     * @param key   The key to use for encryption.
     */
    public Cipher_AES(String[][] key) {
        this.key = key;
        this.keyGen = new RoundKeyGenerator(key);
    }

    /**
     * Default constructor - Use default key.
     */
    public Cipher_AES() {
        this(DEFAULT_KEY);
    }
    //endregion

    //region Getters
    public RoundKeyGenerator getKeyGenerator() {
        return keyGen;
    }

    /**
     * Getter for the previous encrypted output as a matrix.
     * @return  The matrix.
     */
    public String[][] getPreviousOutputState() {
        return this.previousOutputState;
    }

    /**
     * Getter for the key used for this AES instance.
     * @return  The key.
     */
```

```
public String[][] getKey() {
    return this.key;
}
//endregion

//region Interface Overrides
@Override
public String encrypt(String plaintext) {
    String[][] state = getInputBlock(plaintext);

    state = addRoundKey(state, keyGen.getFirstKey());
    state = performRounds(state);
    state = performFinalRound(state);

    previousOutputState = state;

    String output = getOutputString(state);
    logEncryption(plaintext, output);
    return output;
}

@Override
public String decrypt(String encryptedText) {
   return null; //TODO
}
//endregion

//region Encryption Helpers
/**
 * Perform the final processing step.
 * @param state The state to process.
 * @return      The processed state.
 */
private String[][] performFinalRound(String[][] state) {
    state = subBytes(state);
    state = shiftRows(state);
    state = addRoundKey(state, keyGen.getLastKey());
    return state;
}


/**
 * Perform a designated number of rounds.
 * @param state     The initial state to perform the rounds on.
 */
private String[][] performRounds(String[][] state) {
    for (int i = 1; i < keyGen.getNumRounds(); i++) {
        state = subBytes(state);
        state = shiftRows(state);
        state = mixColumns(state);
        state = addRoundKey(state, keyGen.getRoundKey(i));
    }

    return state;
}
```

```
/**
 * Perform the add round key operation on the current state.
 * @param state     The current state.
 * @param roundKey  The key to XOR with the state.
 */
private String[][] addRoundKey(String[][] state, String[][] roundKey) {
    for (int i = 0; i < blockSideLength; i++) {
        for (int j = 0; j < blockSideLength; j++) {
            state[i][j] = CipherUtils.hex_XOR(state[i][j], roundKey[i][j]);
        }
    }
    return state;
}

/**
 * Perform the substitute bytes operation using the declared s-box.
 * @param state The state to transform.
 */
private String[][] subBytes(String[][] state) {
    for (int i = 0; i < blockSideLength; i++) {
        for (int j = 0; j < blockSideLength; j++) {

            // Get the left and right piece
            String left = state[i][j].substring(0, 1);
            String right = state[i][j].substring(1, 2);

            // Get the row and column in the array
            int row = CipherUtils.hexToDecimal(left);
            int column = CipherUtils.hexToDecimal(right);

            // Get new value and overwrite old value in state
            String lookup = S_BOX[row][column];
            state[i][j] = lookup;
        }
    }

    return state;
}

/**
 * Perform the shift rows operation on the current state.
 * Shift row 0 0 places, shift row 1 1 place, shift row 2 2 places and shift row 3 3 places.
 * @param state The current state.
 */
private String[][] shiftRows(String[][] state) {
    // Invert the 2d array for row oriented processing
    String[][] invertedState = getInvertedArray(state);

    // For each row
    for (int i = 0; i < blockSideLength; i++) {
        // Shift the current row by the current index
        invertedState[i] = circularShiftRowLeft(invertedState[i], i);
    }

    // Re-invert the 2d array for further column oriented processing
    return getInvertedArray(invertedState);
```

```
}

/**
 * Perform a shift row operation, to the left.
 * @param array The array to shift.
 * @param shift The amount of places to shift.
 */
private String[] circularShiftRowLeft(String[] array, int shift) {
    String[] temp = new String[array.length];

    // Copy non shifted section to start of new array
    System.arraycopy(array, shift, temp, 0, array.length - shift);
    // Copy shifted section to end of new array
    System.arraycopy(array, 0, temp, array.length - shift, shift);

    return temp;
}

/**
 * Perform a mix columns operation on the current state.
 * @param state The state.
 */
private String[][] mixColumns(String[][] state) {
    // Invert the state to get columns
    String[][] mixedColumnState = new String[blockSideLength][blockSideLength];

    // For each column in the column state
    for (int i = 0; i < blockSideLength; i++) {
        String[] binaryColumn = new String[blockSideLength];

        // Convert current column to binary
        for (int j = 0; j < blockSideLength; j++) {
            binaryColumn[j] = CipherUtils.hexToBinaryString(state[i][j]);
        }

        String[] resultColumn = new String[blockSideLength];

        // For each entry in the column
        for (int j = 0; j < blockSideLength; j++) {

            // Get the matrix row for this entry
            String[] matrixRow = MIX_COLUMN_MATRIX[j];

            // Perform the mix column operation
            resultColumn[j] = performMixColumn(binaryColumn, matrixRow);
        }

        // Convert result column to hex and store
        for (int j = 0; j < blockSideLength; j++) {
            mixedColumnState[i][j] = CipherUtils.binaryStringToHex(resultColumn[j]);
        }
    }

    return mixedColumnState;
}
```

```
/**
 * Perform the mix column operation.
 * @param column       The column of binary strings.
 * @param matrixRow    The row from the galois matrix.
 * @return             The result of the operation.
 */
private String performMixColumn(String[] column, String[] matrixRow) {
   String[] results = new String[blockSideLength];

   // Populate results
   for (int i = 0; i < blockSideLength; i++) {
      String binaryValue = column[i];
      String matrixValue = matrixRow[i];

      switch (matrixValue) {
         case "1":   results[i] = binaryValue;
                     break;
         case "2":   results[i] = perform02(binaryValue);
                     break;
         case "3":   results[i] = perform03(binaryValue);
      }
   }

   // Perform XOR on results
   String result1 = CipherUtils.binary_XOR(results[0], results[1]);
   String result2 = CipherUtils.binary_XOR(result1, results[2]);
   String result3 = CipherUtils.binary_XOR(result2, results[3]);

   return result3;
}

/**
 * Perform the 03 operation.
 * @param binary    The binary string.
 * @return          The result of 03.
 */
private String perform03(String binary) {
   return CipherUtils.binary_XOR(binary, perform02(binary));
}

/**
 * Perform the 02 operation.
 * If left most bit is 0, shift left only.
 * If left most bit is 1, shift left and XOR with 27
 * @param binary    The binary string.
 * @return          The result of 02.
 */
private String perform02(String binary) {
   final String binary27 = CipherUtils.decimalToBinaryString(27);
   String result;

   if (binary.substring(0,1).equals("0")) {
      result = CipherUtils.shiftBinaryStringLeft(binary, 1);
   } else {
      String shifted = CipherUtils.shiftBinaryStringLeft(binary, 1);
      result = CipherUtils.binary_XOR(shifted, binary27);
```

```
    }

    return result;
}
//endregion

//region Shared helpers
/**
 * Transform a hex block into a string of text.
 * @param state The hex block
 * @return      The output text
 */
private String getOutputString(String[][] state) {
    StringBuilder output = new StringBuilder();
    for (int i = 0; i < blockSideLength; i++) {
        for (int j = 0; j < blockSideLength; j++) {
            output.append(String.valueOf((char)Integer.parseInt((state[i][j]), 16)));
        }
    }

    return output.toString();
}

/**
 * Transform a string of plaintext into a hex block
 * @param input The input text
 * @return      The hex block
 */
private String[][] getInputBlock(String input) {
    int charCount = input.length();
    blockSideLength = charCount / 4;

    String[][] inputBlock = new String[blockSideLength][blockSideLength];

    int charIndex = 0;
    for (int i = 0; i < blockSideLength; i++) {
        for (int j = 0; j < blockSideLength; j++) {
            inputBlock[i][j] = Integer.toHexString((int) input.charAt(charIndex));
            charIndex ++;
        }
    }

    // Account for small values, prefix zero
    for (int i = 0; i < blockSideLength; i++) {
        for (int j = 0; j < blockSideLength; j++) {
            String value = inputBlock[i][j];

            if (value.length() < 2) {
                value = 0 + value;
            }

            inputBlock[i][j] = value;
        }
    }

    return inputBlock;
```

```
}

/**
 * Invert the columns and rows of a 2d array.
 * @param array The array to invert.
 * @return      The inverted array.
 */
private String[][] getInvertedArray(String[][] array) {
    String[][] invertedArray = new String[blockSideLength][blockSideLength];

    for (int i = 0; i < blockSideLength; i++) {
        for (int j = 0; j < blockSideLength; j++) {
            invertedArray[i][j] = array[j][i];
        }
    }

    return invertedArray;
}
//endregion

//region Round Key Generator
/**
 * Inner class for round key generation.
 */
public class RoundKeyGenerator {

    private int numRounds;
    private List<String[][]> roundKeys; // 4x4 arrays in list - 11 of these
    private String[][] initialKey;

    /**
     * Constructor.
     * @param key   The key to generate round keys from.
     */
    private RoundKeyGenerator(String[][] key) {
        this.initialKey = key;
        roundKeys = performKeyExpansion();
    }

    public String[][] getFirstKey() {
        return roundKeys.get(0);
    }

    public String[][] getLastKey() {
        return roundKeys.get(roundKeys.size() -1);
    }

    public String[][] getRoundKey(int roundNumber) {
        return roundKeys.get(roundNumber);
    }

    public int getNumRounds() {
        return numRounds;
    }

    /**
```

```
        * Calculate the number of rounds to perform based on the set encryption key.
        */
       private int calculateNumRounds() {
           return 10; // TODO: Use key to calculate round count
       }

       /**
        * Generate a list of round keys based on the initial key.
        * @return  The list of round keys.
        */
       private List<String[][]> performKeyExpansion() {
           numRounds = calculateNumRounds();
           KeyExpansion_AES keyExpansion_aes = new KeyExpansion_AES();
           return keyExpansion_aes.keyExpansion(initialKey, 4, numRounds, 4);
       }
   }
   //endregion
}
```

2.1.4 AES - Key Expansion

```
public class KeyExpansion_AES {

   private static final int[] ROTATEPERMUTATION = {2, 3, 4, 1};

   /**
    * main key expansion method
    *
    * @param originalKey initial algorithm key
    * @param Nk number of keys
    * @param Nr Number of rounds
    * @param Nb Number of blocks
    * @return list of each key for each round
    */
   public List<String[][]> keyExpansion(String[][] originalKey, int Nk, int Nr, int Nb) {

       String parsedString = "";

       for (String[] strings : originalKey) {
           for (String string : strings) {
               parsedString += string;
           }
       }

       return keyExpansion(parsedString, Nk, Nr, Nb);
   }

   private List<String[][]> keyExpansion(String key, int Nk, int Nr, int Nb) {

       String w[] = new String[44];
       String temp;

       int i = 0;
       //LOAD INITIAL KEY
       while (i < Nk) {
```

```
            w[i] = key.substring(8 * i, (8 * i) + 8);
            i++;
        }

        i = Nk; //4

        String rcon = "";
        for (int j = 1; j < 10; j++) {
            rcon = generateRcon(i, Nk);
        }

        //num rounds*num blocks
        while (i < Nb * (Nr + 1)) {

            rcon = generateRcon(i, Nk);

            temp = w[i - 1];

            if (i % Nk == 0) {

                temp = applyRotation(temp, ROTATEPERMUTATION);

                temp = applySubWord(temp);

                temp = applyXorToSubwordAndRcon(temp, rcon);

            } else if ((Nk > 6) && (i % Nk == 4)) {
                //temp = SubWord(temp);
            }

            w[i] = bigBinaryToHex(xorStrings(temp, w[i - Nk]));

            i++;

        }

        return chopArrays(Nk, Nr, w);

    }

    private List<String[][]> chopArrays(int Nk, int Nr, String[] w) {

        String[][] w_final = new String[44][4];

        int count = 0;

        for (String w1 : w) {

            String tmp[] = new String[4];

            tmp[0] = w1.substring(0, 2);
            tmp[1] = w1.substring(2, 4);
            tmp[2] = w1.substring(4, 6);
            tmp[3] = w1.substring(6, 8);

            w_final[count] = tmp.clone();
```

```
            count++;
        }

        ArrayList<String[][]> res = new ArrayList<>();

        String[][] tmp;

        for (int j = 0; j < Nr + 1; j++) {
            tmp = new String[4][4];

            for (int k = 0; k < Nk; k++) {
                tmp[k][0] = w_final[(j * 4) + k][0];
                tmp[k][1] = w_final[(j * 4) + k][1];
                tmp[k][2] = w_final[(j * 4) + k][2];
                tmp[k][3] = w_final[(j * 4) + k][3];
            }

            res.add(tmp);
        }

        return res;
    }

    public String generateRcon(int round, int nK) {

        //to calculate do left-shift followed with a conditional XOR with a constant
        //rcon = (rcon left shift) ^ (0x11b & -(rcon>>7));
        //Convert to binary
        //• If leftmost bit is zero, shift left
        //• Iif leftmost bit is one, shift left and XOR with 00011011
        //• Convert back to hex
        String res = "";

        int val = round / nK;

        round = val;

        int tmp = (int) Math.pow(2, round - 1);

        String binString = Integer.toBinaryString(tmp);
        while (binString.length() < 8) {
            binString = "0" + binString;
        }

        if (round > 8) {

            //System.out.println("\tbinString = " + binString);
            if (binString.charAt(0) == '0') {
                binString = binString.substring(round - 8);
            } else {
                binString = binString.substring(round - 8);
                if (round == 10) {
                    binString = generateRcon((round - 1) * 4, nK);
                    //System.out.println("got previous rcon value = " + binString);
                    binString = hexToBinary(binString.substring(0, 2)) + "0";
```

```
                //System.out.println("updated to = " + binString);
                binString = binString.substring(1);
                //System.out.println("round 10 binstring now = " + binString);
            } else {
                binString = xorBinaryStrings(binString, "00011011");
            }
        }
    } else {
        //System.out.println("\tbinString = " + binString);
    }

    while (binString.length() < 8) {
        binString = "0" + binString;
    }

    String thp = binaryToHex(binString.substring(0, 4));
    res = thp + binaryToHex(binString.substring(4, 8)) + "000000";

    //System.out.println("\tres = " + res);

    return res;

}

private String xorStrings(String str1, String str2) {
    String res = "";
    String tmp1 = "";
    String tmp2 = "";
    for (int i = 0; i < 4; i++) {
        tmp1 = hexToBinary(str1.substring(i * 2, ((i * 2) + 2)));
        tmp2 = hexToBinary(str2.substring(i * 2, ((i * 2) + 2)));
        res = res + xorBinaryStrings(tmp1, tmp2);
    }
    return res;
}

private String applyRotation(String source, int[] rotatePermutation) {
    String res = "";
    String array[] = new String[4];
    String[] copy = new String[4];

    for (int i = 0; i < 4; i++) {
        array[i] = source.substring(i * 2, ((i * 2) + 2));
    }

    System.arraycopy(array, 0, copy, 0, array.length);

    for (int i = 0; i < rotatePermutation.length; i++) {
        array[i] = copy[rotatePermutation[i] - 1];
    }

    for (int i = 0; i < 4; i++) {
        res = res + array[i];
    }

    return res;
```

```
}

private String applySubWord(String source) {
    String res = "";

    int x;
    int y;

    for (int i = 0; i < 4; i++) {
        y = hexToDecimal(source.substring(i * 2, i * 2 + 1));
        x = hexToDecimal(source.substring((i * 2 + 1), (i * 2 + 1) + 1));

        res = res + Cipher_AES.S_BOX[y][x];
    }

    return res;
}

private String applyXorToSubwordAndRcon(String subword, String rcon) {
    String tmp = subword;

    String binaryRcon = hexToBinary(rcon.substring(0, 2));
    String binarySubword = hexToBinary(tmp.substring(0, 2));

    String Xor = "" + xorBinaryStrings(binaryRcon, binarySubword);

    subword = binaryToHex(Xor) + tmp.substring(2, 8);

    return subword;
}

private String xorBinaryStrings(String str1, String str2) {

    String res = "";

    for (int i = 0; i < str1.length(); i++) {
        if (((Integer.parseInt(str1.substring(i, i + 1))) ^ (Integer.parseInt(str2.substring(i, i + 1)))) == 1) {
            res = res + "1";
        } else {
            res = res + "0";
        }
    }

    return res;
}

private String hexToBinary(String hex) {

    String binary = Integer.toBinaryString(Integer.parseInt(hex, 16));
    int binLength = 8;
    for (int i = binary.length(); i < binLength; i++) {
        binary = "0" + binary;
    }

    return binary;
}
```

```
    private int hexToDecimal(String hex) {
        int decimal = Integer.parseInt(hex, 16);
        return decimal;
    }

    private String binaryToHex(String binary) {
        String hex = Integer.toString(Integer.parseInt(binary, 2), 16);
        return hex;
    }

    private String bigBinaryToHex(String binary) {
        String hex = "";
        for (int i = 0; i < binary.length(); i = i + 4) {
            hex = hex + "" + Integer.toString(Integer.parseInt(binary.substring(i, i + 4), 2), 16);
        }

        return hex;
    }
}
```

2.1.5 RSA

```
public class Cipher_RSA extends BaseCipher {

    //region Fields

    /**
     * The P value for RSA. - Chose a larger value of P to handle a larger number range.
     */
    private static final int p = 257;

    /**
     * The Q value for RSA. - Chose a larger value of Q to handle a larger number range.
     */
    private static final int q = 337;

    /**
     * The D value for RSA.
     */
    private static final int d = 17;

    /**
     * The value of E.
     */
    private int e;

    /**
     * The value of N.
     */
    private int n = p*q;

    /**
     * The value of W.
     */
    private int w = (p-1) * (q-1);
```

```
/**
 * The character map.
 */
private static final Map<Character , String> characterMap = new HashMap<Character, String>() {{
    put(' ', "00");
    put('A', "01");
    put('B', "02");
    put('C', "03");
    put('D', "04");
    put('E', "05");
    put('F', "06");
    put('G', "07");
    put('H', "08");
    put('I', "09");
    put('J', "10");
    put('K', "11");
    put('L', "12");
    put('M', "13");
    put('N', "14");
    put('O', "15");
    put('P', "16");
    put('Q', "17");
    put('R', "18");
    put('S', "19");
    put('T', "20");
    put('U', "21");
    put('V', "22");
    put('W', "23");
    put('X', "24");
    put('Y', "25");
    put('Z', "26");
}};

//region Abstract class override methods.

@Override
public String encrypt(String plaintext) {

    plaintext = plaintext.toUpperCase();

    String output = "";

    if (plaintext.length() % 2 != 0)
    {
        plaintext += " ";
    }

    e = calculateEFromEuclidGcd(d, w, w);

    for (int i = 0; i < plaintext.length(); i += 2) {

        char blockChar1 = plaintext.charAt(i);
        char blockChar2 = plaintext.charAt(i+1);

        String blockNum1 = convertCharToNumber(blockChar1);
```

```
            String blockNum2 = convertCharToNumber(blockChar2);

            String fullBlock = blockNum1 + blockNum2;

            long numBlock = Long.parseLong(fullBlock);

            long cipher = applyExponentiationBySquaringDivision(numBlock, e, n);

            output += Long.toString(cipher) + " ";

        }

        logEncryption(plaintext, output);

        output = output.trim();

        return output;
    }

    @Override
    public String decrypt(String encryptedText) {

        String plainText = "";

        e = calculateEFromEuclidGcd(d, w, w);

        String[] cipherTextBlocks = encryptedText.split("\\ ");

        for (String block : cipherTextBlocks) {

            long numberBlock = Long.parseLong(block);

            long message = applyExponentiationBySquaringDivision(numberBlock, d, n);

            String textBlock = Long.toString(message);

            textBlock = String.format("%04d", Long.parseLong(textBlock));

            String blockPart1 = textBlock.substring(0, 2);
            String blockPart2 = textBlock.substring(2, 4);

            char blockPart1Char = convertNumberToChar(blockPart1);
            char blockPart2Char = convertNumberToChar(blockPart2);

            String finalMessageBlock = Character.toString(blockPart1Char) + Character.toString(blockPart2Char);

            plainText += finalMessageBlock;

            int x = 3;

        }

        logDecryption(encryptedText, plainText);
        return plainText;
    }
```

```
//endregion

//region Methods

/**
 * Calculates (num^exponent) mod n using Exponentiation by Squaring/Dividing.
 * @param num The number.
 * @param exponent The power to raise to.
 * @param n The n number from the equation.
 * @return
 */
public long applyExponentiationBySquaringDivision(long num, long exponent, long n) {

    char[] binaryExponentChars = Long.toBinaryString(exponent).toCharArray();

    long c = 1;

    for (int i = 0; i < binaryExponentChars.length; i++)
    {
        long tmp = c * c;
        c = tmp % n;

        if (binaryExponentChars[i] == '1') {
            long tmp2 = c * num;
            c = tmp2 % n;
        }
    }

    return c;
}

/**
 * Converts a text block to numbers.
 * @param individualCharacter The character.
 * @return A numerical representation of the character.
 */
private String convertCharToNumber(char individualCharacter) {

    return characterMap.get(individualCharacter);

}

/**
 * Converts a text block to numbers.
 * @param individualCharacter The character.
 * @return A numerical representation of the character.
 */
private char convertNumberToChar(String individualCharacter) {

    char rtr = ' ';

    for (Map.Entry<Character,String> entry : characterMap.entrySet()) {

        if (entry.getValue().equals(individualCharacter)) {
            rtr =  entry.getKey();
        }
```

```
        }

        return rtr;
    }

    /**
     * Returns the GCD of 2 numbers, as calculated via Euclid's algorithm.
     * @param a The first number.
     * @param b The second number.
     * @return The greatest common divider of A and B.
     */
    public int calculateEFromEuclidGcd(int a, int b, int w) {

        int prevAi = 1;
        int ai = 0;

        int prevBi = 0;
        int bi = 1;

        while (b != 0)
        {
            int quotient = a / b;
            int remainder = a - quotient * b;

            int newAi = prevAi - quotient * ai;
            int newBi = prevBi - quotient * bi;

            prevAi = ai;
            ai = newAi;
            prevBi = bi;
            bi = newBi;

            a = b;
            b = remainder;
        }

        if (prevAi < 0)
        {
            return w + prevAi;
        }
        else
        {
            return prevAi;
        }

    }

    //endregion
}
```