RSAp – PROJECT REPORT

Valentin REVERSAT – AGH A2020

Summary

Application introduction

This application is intended to encrypt and decrypt text message (or text file) using the RSA algorithm encryption. It uses *Flutter*, the Google SDK to build Android and iOS application using the same code base and the *Dart* programming language, also developed by *Google.*

Architecture

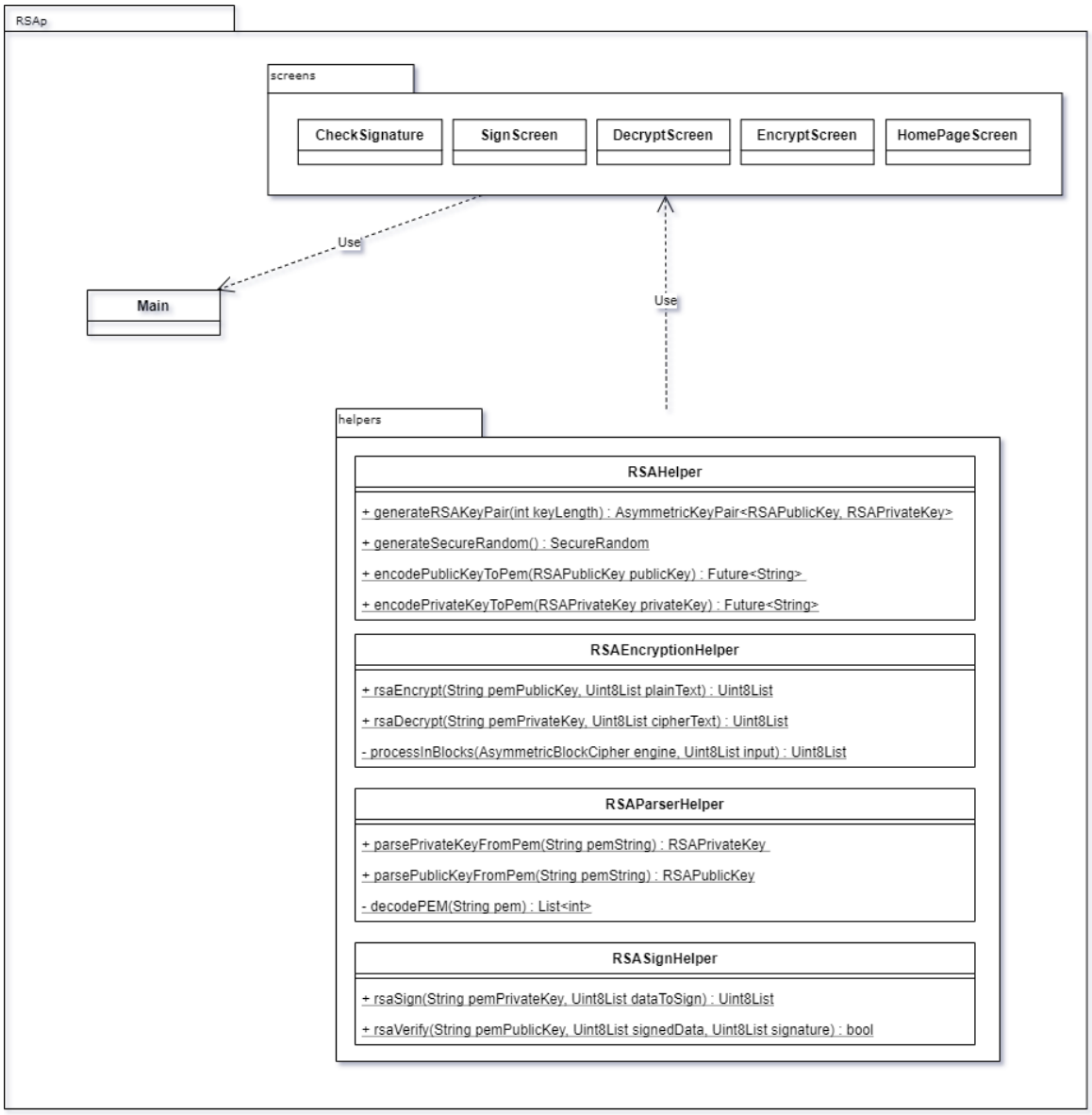


Figure 1 - Architecture diagram of the application

Class description

This application uses 2 main packages:

- ***screens*** which contains all the activities of the application:
    - *CheckSignature:* This activity let the user to check the signature of a file using the signature file, the signed text file and the public key.
    - *SignScreen:* This activity let the user signs a text file or an input text with the private key. It generates the signing file in the download directory of the phone.
    - *DecryptScreen:* This activity let the user to decrypt an encrypted message using the private key stored on the phone storage
    - *EncryptScreen:* This activity let the user to encrypt a text file or an input text using a public key file picked up by the user
    - *HomePageScreen:* Home page of the application where you can navigate through the different activities

- ***helpers*** which contains all the classes need to perform the encryption.
    - *RSAHelper:* Class used to generate the public/private key pair:
        - *generateRSAKeyPaire:* Generate the key pair according to the keyLength parameter
        - *generateSecureRandom:* Generate a random integer to perform the key generation
        - *encodePublicKeyToPEM:* Serialize the public key to be saved on a text file
        - *encodePrivateKeyToPEM:* Serialize the private key to be stored on the shared preferences
    - *RSAEncryptionHelper:* Class to perform the encryption and the decryption:
        - *rsaEncrypt:* Encrypt a text in ASCII format using the public key
        - *rsaDecrypt:* Decrypt a text in ASCII format using the private key
        - *processInBlocks:* Method to process the decryption and the encryption by block
    - *RSAParserHelper:* Class to perform the serialization and deserialization of the key pair:
        - *parsePrivateKeyFromPEM:* Deserialize the private key
        - *parsePublicKeyFromPEM:* Deserialize the public key
    - *RSASignHelper: Class to perform the signing of a text:*
        - *rsaSign: Sign a text in ASCII format using the private key*
        - *rsaVerify:* Check the signature using the public key, the signature file and the text file in ASCII format

External libraries

This application uses external libraries to perform various tasks like file opening or encryption processing:

- *pointycastle* **:** A Dart library for encryption and decryption
- asn1lib: Library to encode & decode ASN1 using BER encoding.
- permission_handler : Check and request the permission to access to the device storage
- file_picker: A package that allows you to use the native file explorer to pick single or multiple files, with extensions filtering support.
- shared_preferences: Wraps platform-specific persistent storage for simple data (NSUserDefaults on iOS and macOS, SharedPreferences on Android)

Building information

If you want to build the app directly on your computer, you need to follow these steps:

1- **Clone the repository on your machine:**

git clone https://github.com/UTBM-AGH-courses/agh-rsap.git

2- **Install the flutter SDK (documentation here)**

3- **Run the following command in the project folder:**

flutter build apk --profile –split-per-abi

4- **Voilà!**

Building the app for iOS device is not yet supported

You can also directly download the APK file here

User documentation

Home activity

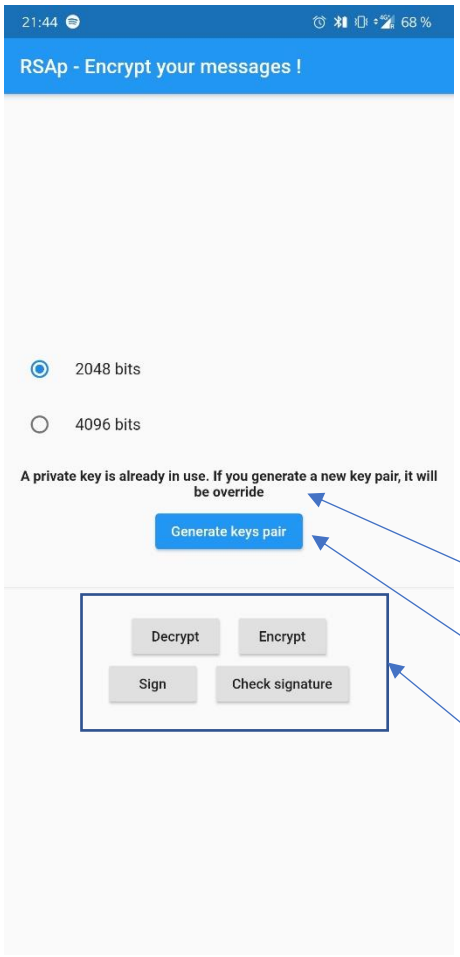


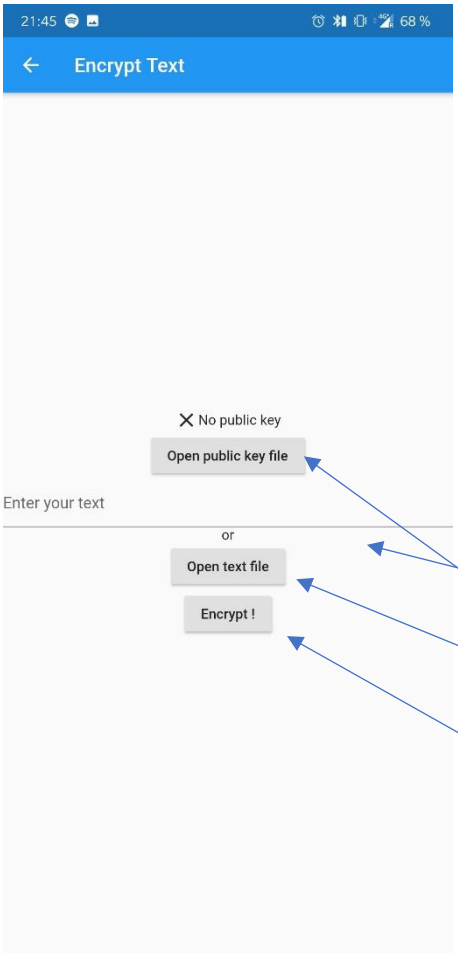
Figure 2 - Home activity screenshot

This activity allows the user to navigate between the different features of the application (encrypt, decrypt, sign and check the signature). It also allows the user to generate a RSA key pair with the button “Generate keys pair”. The user can choose the size of the key pair: 2048 or 4096 bits. If a private key is already saved in the device, a waring message is display.

Warnings:

- **When the key pair in generating, the application cannot respond to any input. Don’t spam the button, it’s working**
- **The generation of a 4096 bits key pair is quite long.**

Encrypt activity



This activity allows the user to encrypt its message. First of all, the user has to open a public key file to perform any encryption. The button “Open public key file” will open the file explorer to pick the appropriate file. If the file matches the PEM public key format, the message “*Public key loaded*” appear. Then the user has the choice either to choose a text file to encrypt or only an input text. When the correct input is filled, he can press on “*Encrypt!*”. This will generate an encrypted file into the download directory of the device

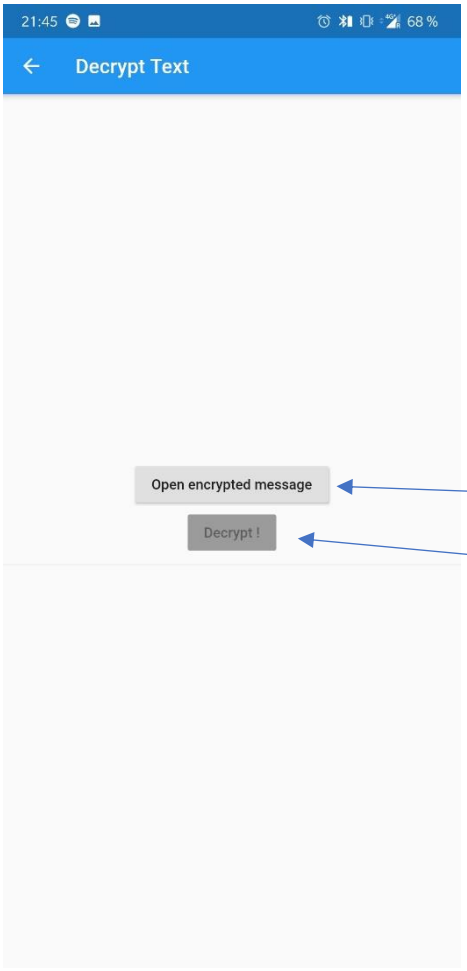
Input text to be encrypted

Buttons to open the file picker

Buttons to encrypt the text file or the input text

Figure 3 - Encrypt activity screenshot

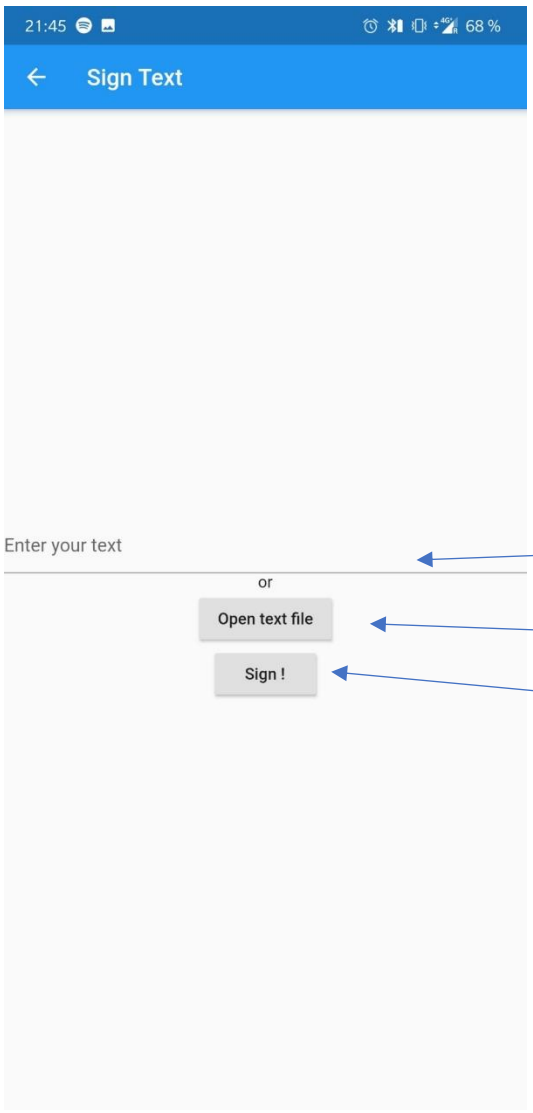
Decrypt activity



With this activity, the user can decrypt a message encrypted by the previous activity. He has just to open the encrypted text file via the "*Open encrypted message*" button which will open the file explorer. Then the button "*Decrypt!*" will became clickable.

Figure 4 - Decrypt activity screenshot

Sign activity



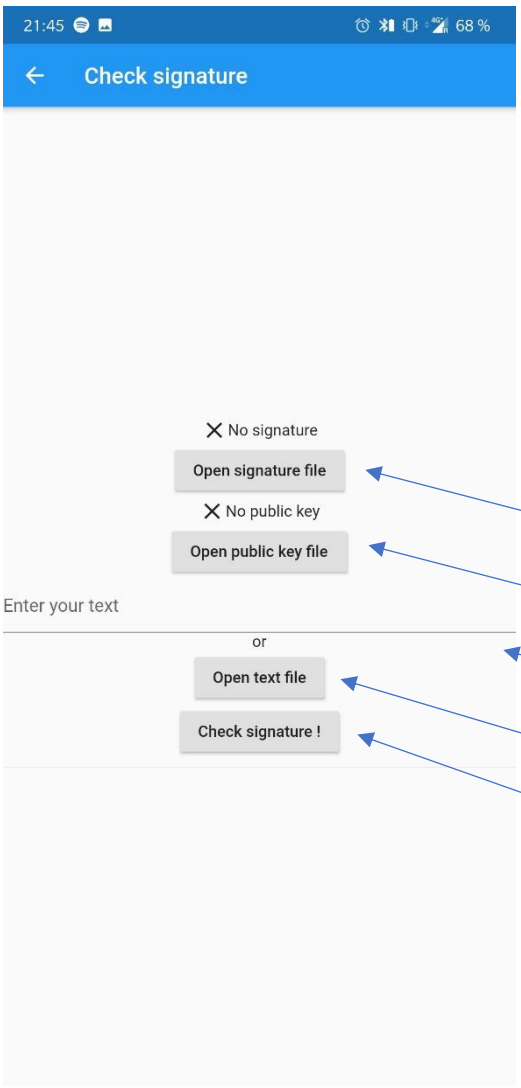
This activity generates a signature file according to the private key stored on the device. The user can either enter a text via the input or pick a text file to be sign. When done, he has just to press on "*Sign!*". This will generate a signature file and store it into the download directory of the device.

Input text to enter a text to sign

Button to open the file explorer

Button to sign the chosen input

Signing verification activity



This activity is intended to check the signature of a text file or a simple text. The user has to open the signature file and the public key used for the signing process. If the three inputs match, the message **"OK"** appears at the bottom of the screen. If not, the message **"NOK"** appears.

Button to open the signature file

Button to open the public key

Input text to enter a text to check

Button to open the file explorer to pick the corresponding text file

Button to check the signature

Theory of RSA encryption

Invention

The main idea behind the RSA encryption is the use of two separate keys. The private one is kept by the sender of the message and must not be disclose. It will be used to decrypt the messages. The public one is shared with everyone one and used to encrypt a given message.

> *RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem that is widely used for secure data transmission. It is also one of the oldest. The acronym RSA comes from the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman, who publicly described the algorithm in 1977.*
>
> *Source – Wikipedia*

Key generation

1- In a first place, we need to choose 2 distinct prime number p and q.

2- Then compute $n = pq$ and $\phi(n) = (p-1)(q-1)$

3- Choose 1 random integer e which verify $1 < e < \phi(n)$

4- Compute $d \equiv e^{-1} \, mod \, \phi(n)$

5- The coup (e, n) is the public key and the (d, n) is the private key

Encryption and decryption

To encrypt a message, you need to use this formula:

$$M \equiv m^e \, mod \, n$$

To decrypt:

$$m \equiv M^d \, mod \, n$$

with:

M the encrypted message and m the plain text message.

Limit of RSA encryption

This algorithm is widely used on the computer world nowadays but may be quite expensive in terms of key computation especially for large keys (4096 bits) on mobile devices. Large keys take also more space on the internal storage.

The solution: ECC algorithm

The ECC (Elliptic Curve Cryptography) algorithm is used for website certificates signing but also as the key stone for the Bitcoin. In fact, this cryptosystem is more efficient in terms of computation use and reach the same level of security of RSA but with a shorter key length as in the chart below.

ECC	**RSA**
163	*1024*
233	*2048*
283	*3072*
409	*7680*
571	*15360*

This made the cryptosystem well fitted for the mobile devices and websites by being more efficient and allowing shorter key with the same level of security.

The theory of ECC encryption

ECC uses a mathematical object called **elliptic curve:**

Like RSA, ECC is based on a trapdoor function. In other words, it is easy to get B from A, but much harder to get A from B. When we choose a point A on the curve and multiply it by 2 (by doting), we get another point on the same curve.

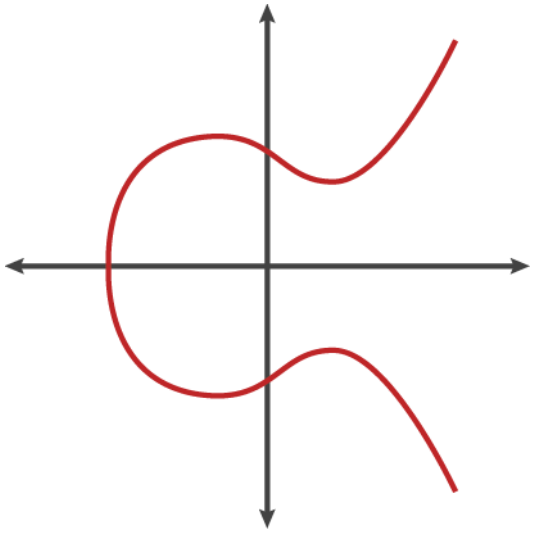


Figure 5 - Example of elliptic curve

Example:

$$A\ dot\ A = 2A = B$$

$$A\ dot\ B = 3A = C$$

$$A\ dot\ C = 4A\ = D$$

By getting D (the public key) it's very hard to find the number 4 (the private key). This computation is called discrete logarithm problem. Despite almost three decades of research, mathematicians still haven't found an algorithm to solve this problem that improves upon the naive approach.

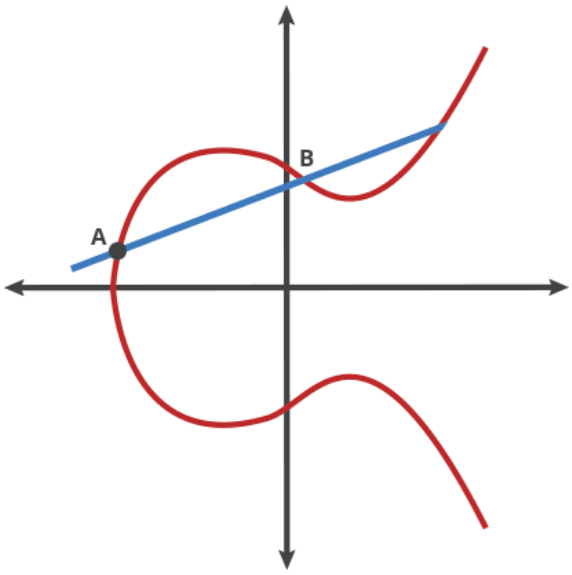


Figure 6 - Example of point dotting

Key generation

1- First, we need to pick an elliptic curved function shaped like: $y^2 = x^3 + ax + b$

2- Choose two integers, a and b

3- Pick a point called generator on the curve: (Gx, Gy)

4- Pick a random prime number p

5- Pick a random integer n

6- Pick a random generated integer d which verify $d < n$

7- Compute $Q = dG$

8- Q is the public key and d the private key

Limits of ECC encryption

Even if ECC may be better than RSA, the choice of the parameters for the key generation may have a consequence on the strength of the keys. If d is too low, it may be easy to guess it by brute force.

Bitcoin example

The bitcoin block chain algorithm uses these parameters:

$$a = 0$$

$$b = 8$$

$$Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798$$

$$Gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8$$

$$p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F$$

$$n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141$$

$$d = 0x51897B64Z85C3F714BBA707E867914295A1377A7463A9DAE8EA6A8B914246319$$