Introduction

Cryptography, the science of securing communication and information, has played a pivotal role throughout history in safeguarding sensitive data from unauthorized access. In the digital age, the significance of cryptographic techniques has only intensified with the increasing reliance on electronic communication and data storage. This report delves into the realm of cryptography, shedding light on various cryptographic primitives that form the backbone of secure communication.

Cryptographic Primitives:

1. **Caesar Cipher:** The Caesar cipher, one of the earliest known encryption techniques, involves shifting the letters of the alphabet by a fixed number of positions. While simple, it laid the foundation for more sophisticated encryption methods.
2. **Affine Cipher:** The affine cipher is a type of substitution cipher that combines mathematical functions to transform the plaintext. It introduces a greater level of complexity compared to the Caesar cipher.
3. **Vigenère Cipher:** The Vigenère cipher, an extension of the Caesar cipher, employs a keyword to encrypt text. This method enhances security by introducing variability based on the chosen keyword.
4. **AES (Advanced Encryption Standard):** AES, a widely adopted symmetric encryption algorithm, is renowned for its efficiency and security. Employed in various applications, including data encryption and secure communication, AES ensures robust protection against unauthorized access.
5. **RSA (Rivest–Shamir–Adleman):** RSA, a widely used asymmetric encryption algorithm, facilitates secure data transmission by utilizing a pair of public and private keys. Its mathematical foundation makes it a cornerstone of modern secure communication.
6. **SHA-256 (Secure Hash Algorithm 256-bit):** As a cryptographic hash function, SHA-256 generates a fixed-size hash value, providing data integrity and authentication. Widely employed in digital signatures and certificate generation, SHA-256 enhances the security of various cryptographic applications.
7. **Digital Signature:** Digital signatures use cryptographic techniques to authenticate the origin and integrity of digital messages or documents. They provide a mechanism for non-repudiation and verification of the sender's identity.
8. **HMAC (Hash-based Message Authentication Code):** HMAC combines a cryptographic hash function with a secret key to authenticate data integrity and origin. Commonly used in network security protocols, HMAC ensures secure communication through message authentication.

JavaFX:

In addition to cryptographic techniques, this report explores the integration of JavaFX, a robust and user-friendly framework for building interactive and visually appealing graphical user interfaces (GUIs). JavaFX facilitates the development of cryptographic applications with a seamless and engaging user experience.

Description TL;DR

To explain how the application works, we have the class diagram that shows the interactions between the different classes.

App class

This is the entry for our app; it extends the **Application** class from JavaFX, which contains the `start` method.

```java
public void start(Stage primaryStage) throws Exception {
    // Initialize the global store
    globalStore.createInstance();
    FXRouter.bind(this, primaryStage, "Crypto App");
    FXRouter.when("home", "./pages/home/page.fxml");
    FXRouter.when("caesar", "./pages/caesar/page.fxml");
    FXRouter.when("aes", "./pages/aes/page.fxml");
    FXRouter.when("rsa", "./pages/rsa/page.fxml");
    FXRouter.when("veginar", "./pages/veginar/page.fxml");
    FXRouter.when("sha256", "./pages/sha256/page.fxml");
    FXRouter.when("hmac", "./pages/hmac/page.fxml");
    FXRouter.when("affine", "./pages/affine/page.fxml");
    FXRouter.when("key generation", "./pages/keyGen/page.fxml");
    FXRouter.when("digital signature", "./pages/digSign/page.fxml");
    FXRouter.goTo("home");
}
```

Here we are doing four important things:

1. Binding the app stage to `FXRouter`.
2. Connecting each route to its corresponding FXML file.
3. Navigating to the home page right after starting the app.
4. Creating the global store instance.

Fxrouter class

This is a class provided by the **fxrouter package** with a couple of static methods. The once with interest are :

- `bind`: Used to bind the app stage to `FXRouter`.
- `when`: Used to link a route with an FXML file. Simply speaking, render FXML when visiting this route.
- `goto`: Used to navigate to different routes.

Global store class

This is a class that follows the singleton pattern to ensure creating data only once when starting the application.

It holds two properties: the `random iv` used in AES CBC and an `errormap` used by the error class.

```
private globalStore() throws Exception {
    errorsMap = error.init();
    iv = symmetric.genIV(128);
}

public static void createInstance() throws Exception {
    if (instance == null)
        instance = new globalStore();
}
```

The `createInstance` will be called at the start of the app, which uses the constructor to initialize the two properties.

Fxml view class

This is an abstraction of the FXML file that will be called by `FXRouter`. The UI content will be different from route to route but still using the same global `app.css`. It binds a `controller Java class` to handle the logic.

Controller abstract class

It's the class that handles the logic of a specific route (FXML file). It holds references to UI elements by their IDs using **FXML annotations**, for example:

```
public class page {
    @FXML
    Button myBtn;
}
```

Here we have a reference to a button on the FXML file with the ID **myBtn**.

Note It goes without saying that the controller for each route will be different, as we're going to see later on.

Home class

This is a controller for the home route; it contains only one function that takes care of routing to the different primitives routes.

```
public void switchRoute(ActionEvent event) throws IOException {
    // This returns the source of the click event
    Object source = event.getSource();
    if (source instanceof Button) {
        Button castSource = (Button) source;
        String route = castSource.getText().toLowerCase();
        FXRouter.goTo(route);
    }
}
```

Confidentiality primitives abstract class

This class abstracts every controller that handles confidentiality, which are `caesar`, `affine`, `veginar`, `aes`, `rsa` because they need to implement the **encrypt** and **decrypt** functions.

Here is an example that handles `caesar`:

```
public class page {
    @FXML
    TextField key;
    @FXML
    TextField input;
    @FXML
    TextArea output;
    @FXML
    Button encrypt;
    @FXML
    Button decrypt;

    public void encrypt() {
        String rawInput = input.getText();
        String rawKey = key.getText();
        if (!error.verify("CAESAR", rawInput, rawKey, null)) return;
        String cipher = classic.ceasar(rawInput, Integer.parseInt(rawKey));
        output.setText(cipher);
    }
    public void decrypt() {
        String rawInput = input.getText();
        String rawKey = key.getText();
        if (!error.verify("CAESAR", rawInput, rawKey, null)) return;
        String message = classic.decryptCeasar(rawInput,
           Integer.parseInt(rawKey));
        output.setText(message);
    }
}
```

so the **encrypt** and **decrypt** changes from primitive to another but the concept is the same take the input from the `input text field` and the key from `key text field` then check for errors , apply the primitive and set `output text field` with the result.

Notes

- For `affine` we retrieve two keys from two different text fields.
- For `rsa` and `aes` the encryption result will get encoded in **base64** why ? check `encode` function in the utils section.
- For `rsa` the process is more strict where you need to generate keys from the keyGen route first.
- check the verify method in utils section.

Integrity abstract class

This represents both `hmac` and `sha256` controllers that have the **toDigest** function.

Hmac's toDigest function.

```
public void toDigest() {
    String rawInput = input.getText();
    String rawKey = key.getText();
    if (!error.verify("HMAC", rawInput, rawKey, null))
          return;
    Mac mac = Mac.getInstance("HmacSHA256");
    SecretKeySpec secretKey = new SecretKeySpec(rawKey.getBytes("UTF-8"),
       "HmacSHA256");
    mac.init(secretKey);
    byte[] digest = mac.doFinal(rawInput.getBytes());
    output.setText(utils.encode(digest));
}
```

As you can see the process is pretty straightforward we get the message , get the key (in case of hmac) , create the digest and set `output text field` with the digest.

Notes

- No key is needed with `sha256` primitive.
- We also need to encode the `sha256` and `hmac` results to **base64**.

Digital signature class

This obviously gets linked to the `digital signature` FXML and contains two functions: **verify** and **sign**.

The way **sign** function works without going into the implementation details, it takes the user input and a private key and sign it with java built-in method.

The **verify** method takes the signature and the public key and verify it with java built-in method. Here is the twist we don't need the message to verify java includes it inside the signature.

KeyGen class

This generates RSA public and private key pair to be used either in RSA or digital signature route.

It contains only one function **generates**:

```
public void generate() throws Exception {
    Pair<PrivateKey, PublicKey> keys = asymmetric.KeyGen(2048);
    Pair<String, String> encodedKeys = asymmetric.encodeKeys(keys);
    publicKey.setText(encodedKeys.x);
    privateKey.setText(encodedKeys.y);
}
```

Check **Utils** section for the `encodeKeys` description.

Primitive abstract class

All primitives need to follow this class by including the **goHome** function which, as the name suggests, routes you back to the home page (`FXRouter` calls home FXML).

```
public void goHome(ActionEvent event) throws IOException {
    FXRouter.goTo("home");
}
```

This class also forces all primitives to handle errors using the global class `error`.

Finally, the error class

This class is responsible for handling errors by binding states to alert messages using a map. The map gets created by the `global store`.

```
HashMap<Integer, String> errorsMap = new HashMap<Integer, String>();
errorsMap.put(1, "You need to enter an input");
errorsMap.put(2, "For affine, you need to enter both keys, and they must be numbers");
errorsMap.put(3, "Key must be a number.");
errorsMap.put(4, "Vegin

ar cipher needs a key that consists of alphabet letters");
errorsMap.put(5, "AES needs a key with a length equal to 16");
errorsMap.put(6, "The input needs to contain only alphabet letters with this method");
errorsMap.put(7, "The second affine key needs to be reversible");
errorsMap.put(8, "Invalid RSA public key. You should probably generate the RSA keys using either the ke
errorsMap.put(9, "Invalid RSA private key. You should probably generate the RSA keys using either the k
errorsMap.put(10, "Hmac key can be anything but not empty");
```

Then we have the `alert` method:

```
public static boolean alert(Integer alertId) {
    HashMap<Integer, String> errorsMap = globalStore.instance.getErrorMap();
    String errorMessage = errorsMap.get(alertId);
    Alert alert = new Alert(AlertType.ERROR);
    alert.setContentText(errorMessage);
    alert.show();
    return false;
}
```

Which gets called by the `verify` method or directly from the controller in case of digital signature because the error is caught by the `try-catch block`.

```java
public static boolean verify(String methodStr, String input, String key1, String key2) {
    if (methodStr.equals("HMAC") && key1.isEmpty())
        return alert(10);
    if (methodStr.equals("AFFINE") && (!isNumber(key1) || !isNumber(key2)))
        return alert(2);
    else if (methodStr.equals("CAESAR") && !isNumber(key1))
        return alert(3);
    else if (methodStr.equals("VEGINAR") && !containsOnlyEnglishAlphabets(key1))
        return alert(4);
    else if (methodStr == "AES" && key1.length() != 16)
        return alert(5);
    else if (input.equals(""))
        return alert(1);
    else if ((methodStr.equals("AFFINE") || methodStr.equals("VEGINAR") ||
            methodStr.equals("CAESAR")) && !containsOnlyEnglishAlphabets(input))
        return alert(6);
    return true;
}
```

Utils

We have certain utility functions we need to talk about.

`encode` This function encodes data to **base64**. But why are we re-encoding to base64 in the first place? Well, the result of certain primitives like `aes` does not respect **utf-8** that gets used by `new String(byte[])` constructor so the only way to show the result to screen is to encode it to base64.

```java
public static String encode(byte[] data) {
    return Base64.getEncoder().encodeToString(data);
}
```

`decode` If we have encoding, we need decoding.

```java
public static byte[] decode(String data) {
    return Base64.getDecoder().decode(data);
}
```

`encodeKeys` To put it simply, this function takes the **PKCS8 encoding** for the public key and the **X509 encoding** of the private key and re-encodes them with base64. This is needed by the keyGen controller.

```java
public static Pair<String, String> encodeKeys(Pair<PrivateKey, PublicKey> keys) {
    byte[] privateEncoded = keys.x.getEncoded();
    byte[] publicEncoded = keys.y.getEncoded();
    return new Pair<String, String>(utils.encode(publicEncoded),
            utils.encode(privateEncoded));
}
```

`decodePublic` This takes the **base64** encoding, decodes it, recreates the **X509 encoding** object and refactors the public key.

```java
public static PublicKey decodePublic(String encodedPublic) {
    KeyFactory factory = KeyFactory.getInstance("RSA");
    X509EncodedKeySpec publicEncoded = new
            X509EncodedKeySpec(utils.decode(encodedPublic));
    PublicKey publicKey = factory.generatePublic(publicEncoded);
    return publicKey;
}
```

`decodePrivate` This takes the base64 encoding, decodes it, recreates the **PKCS8 encoding** object and refactors the public key.

```
public static PrivateKey decodePrivate(String encodedPrivate) {
    PKCS8EncodedKeySpec privateEncoded = new
            PKCS8EncodedKeySpec(utils.decode(encodedPrivate));
    KeyFactory factory = KeyFactory.getInstance("RSA");
    PrivateKey privateKey = factory.generatePrivate(privateEncoded);
    return privateKey;
}
```

`keyGen` This one is straightforward; it takes the user input and turns it into an `aes` key.

```
public static SecretKey keyGenStr(String keyStr) {
    byte[] decodedKey = keyStr.getBytes();
    return new SecretKeySpec(decodedKey, 0, decodedKey.length, "AES");
}
```

Conclusion

In conclusion, the cryptographic application presented in this report exhibits a well-organized and modular structure, leveraging the capabilities of JavaFX for user interface design and FXRouter for efficient navigation between different views. The class diagram provides a comprehensive overview of the key components and their relationships within the application.