Table of Contents

Chapter 1: **Appendix A Software Development Methodology**

1.1 Appendix A.1 Software Development Approach

I intend to follow the full software development lifecycle with adherence to standard software engineering principles (Test-driven Development (TDD) etc). This approach encompasses the five stages of (1) requirements specification, (2) software design that serves as a blueprint for development, (3) implementation with a focus on code quality and functionality, (4) testing to guarantee software reliability and performance, and (5) documentation, for both programmers and users to enhance understanding and usability of the software.

I have chosen to use the Agile methodology due to its precision and adaptability, which could prove crucial for handling the complex cryptographic proofs at the core of signature scheme security. The iterative and flexible nature of Agile allows me to constantly reassess and integrate theoretical insights into my coding, ensuring accurate implementation and prompt identification of discrepancies. In TDD, where coding is a byproduct of testing, I make sure each component functions correctly, a critical aspect in cryptographic software where errors can have serious security consequences. This approach is particularly beneficial for the extended scope of my project (beyond PKCS#1-v1.5), which includes implementing the full suite of deterministic RSA signatures. Agile's iterative sprints and milestones help me in making accurate estimations of time and effort, preventing overruns and ensuring systematic progress. TDD complements this by ensuring that each part of the project is thoroughly tested before moving on, which is particularly important in the integration of the multiple signature schemes.

With this, I consider each of the constituent software development phases as fluid and adaptable. The plan is not to adhere strictly to them, but rather to remain open to revisiting previous phases as necessary. This might occur where I may have mis-estimated tasks, or received new feedback. To aid this adaptive process, I conduct bi-weekly meetings with my project supervisor to discuss ongoing progress, address any concerns, and define short-term objectives. Moreover, I have established review reflection points to regularly evaluate progress and make adjustments as needed.

*Respective TDD unit test classes can be found in the tests subdirectory of its corresponding module. All distinct features can be found in the application/modules directory. The top level application/tests directory houses integration tests.

1.2 Appendix A.2 Technology Choices

When considering the implementation of an RSA digital signature program, multiple programming languages offer distinct features that can influence the efficacy. The choice of language is crucial, as it dictates the available libraries and tools, as well as the overall robustness and performance of the cryptographic operations.

Considering various languages, C and C++ offer high performance and control over system resources, which can be advantageous in cryptography. However, they do not have built-in

support for big integers, relying instead on libraries like GMP [1] for bigint operations. While these libraries are powerful, they introduce additional dependencies and complexities.

Python, with its simplicity and readability, offers native big integer support and is capable of modular exponentiation, random integer generation, and other necessary operations [2] . However, Python's BigInt functionalities are scattered across its standard library, and its cryptographic capabilities are still maturing, with many updates and changes since its initial support in 2009.

After evaluating these options, I decided to use Java for the project. The Java BigInteger class [3] is particularly well-suited for RSA. In practice it can handle arbitrarily long integers, provides essential arithmetic operations on these integers, and offers RSA-specific methods. Notably, the maturity and reliability of the BigInteger class add to its suitability. Being a part of Java since version 1.1 (1997), the class has undergone extensive testing and refinement. These factors, along with the robustness and reliability of the Java programming language, contribute to its suitability for cryptographic applications. The ongoing support and maintenance of the BigInteger class offers more confidence in consideration of its potential application to cryptographic processes.

Chapter 2: **Appendix B Software Development of Proof of Concept program**

2.1 Appendix B.1 Requirements and Analysis

The overall goal of the system is to provide primarily, a basic implementation of the PKCS signature scheme from which a user can interact with via a user interface to perform relevant actions. The program will also include the other considered schemes, to be integrated once the implementation for the PKCS scheme has been established. The core actions comprise, the generation of keys, creation of signatures and finally verification of previously created signatures. The program will form the foundation for the eventual delivery of the benchmarking program used to examine the discussed provable security overhead.

2.1.1 Description of Actors

Table 2.1: Description of Actors for the POC Digital Signature Program

Actor / Role Name	**Role Description and Objective**
User/Signer	Individual who wishes to digitally sign a piece of content. The signer generates key pairs, can input content, and create a digital signature using their private key. Their main goal is to ensure that the content they're signing is authenticated and its integrity is maintained, proving that it hasn't been tampered with.
Verifier	Entity that needs to validate the authenticity and integrity of a digitally signed piece of content. The verifier inputs signed content, a corresponding public key, and attempts to verify a specified digital signature. Their primary objective is to ascertain that the content hasn't been altered post-signing and to confirm the identity of the signer.

2.1.2 User Stories

Essential Requirements

1. Potential signer should be able to generate and retrieve a public-private key pair having provided a key size.
    - User should be presented with a text box to input the key size.
    - The system should handle any exceptions or errors during key generation, displaying to the user of any issues.
    - The system should notify the signer once the key generation process is successful.
    - Once the key is generated the user should have the option to save it to a file.
2. Having provided a message and private key, the signer should be able to retrieve the resulting computed digital signature.

- The signer should be presented with a text box to input the message intended for signing.
- The signer should be able to specify and input the private key using file selection via a browse option.
- The system should handle any exceptions or errors during signature generation, displaying to the signer of any issues.
- The system should notify the signer once the signing process is successful.
- Once the signature is generated the signer should have the option to copy the signature to the clipboard or save it to a file.

3. Having provided a message, its corresponding digital signature, and a public key, the verifier should be able to verify the authenticity of the signature.

   - The verifier should be presented with a text box or file browse option to input the message corresponding to the signature intended for verifying.
   - The verifier should be presented with a file browse option to input the signature intended for verifying.
   - The system should handle any exceptions or errors during verification process, displaying to the verifier of any issues.
   - The system should notify the verifier once the verification process is successful.

4. The signer should be able to sign messages using the PKCS#1 v1.5 Signature Scheme.
5. The verifier should be able to verify messages using the PKCS#1 v1.5 Signature Scheme.
6. The signer should be able to sign messages using the ANSI X9.31 rDSA Signature Scheme.
7. The verifier should be able to verify messages using the ANSI X9.31 rDSA Signature Scheme.
8. The signer should be able to sign messages with partial or full recovery using the ISO/IEC 9796-2:2010 Signature Scheme 1.
9. The verifier should be able to verify messages with partial or full recovery using the ISO/IEC 9796-2:2010 Signature Scheme 1.

Non Essential Requirements

Performance

7. User shoulder be able view a measurement of time taken for each of the signature related operations i.e., Key generation, Signature creation and verification.

Non Functional

8. The program should generate keys within a reasonable timeframe.
9. The program should create signatures within a reasonable timeframe.
10. The program should verify signatures within a reasonable timeframe.

2.1.3 UML Use Case

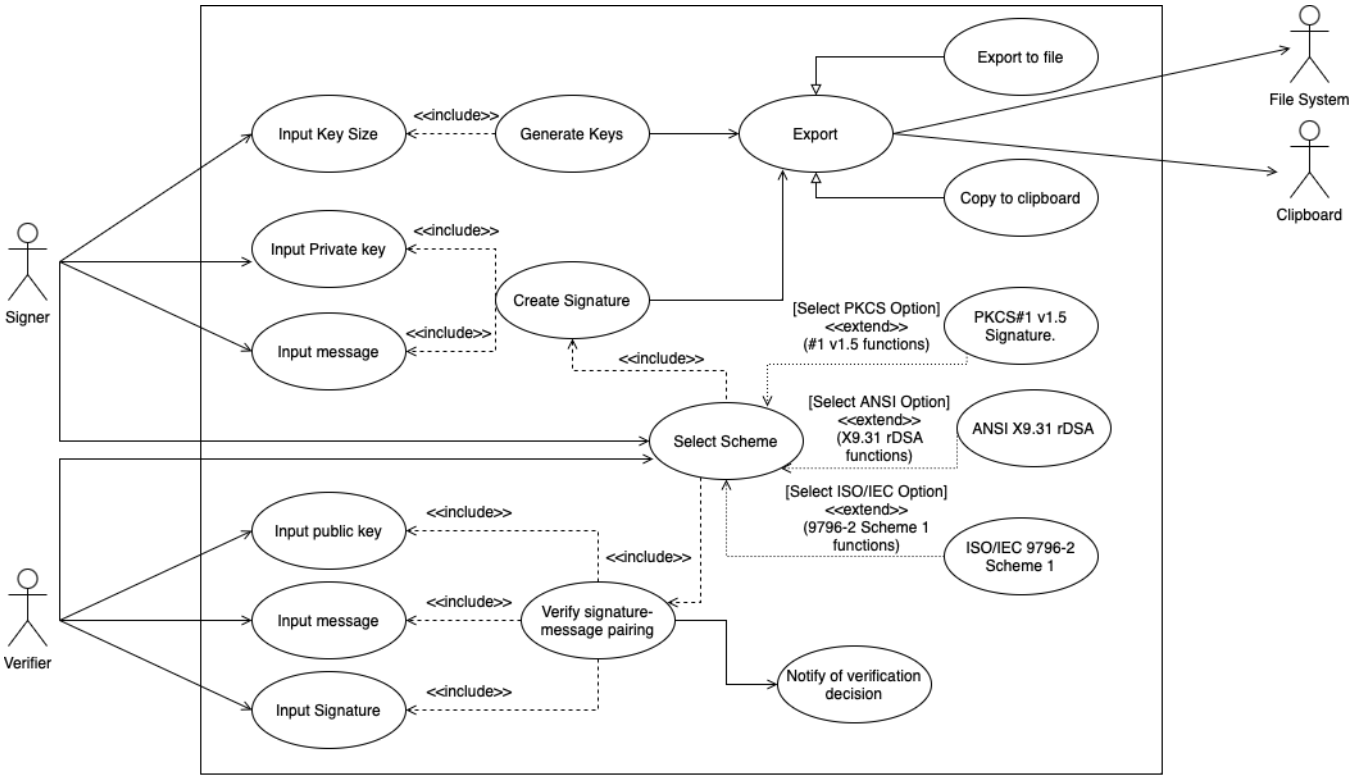

Figure 2.1: UML Use Case Diagram

Generate Keys Use Case

Flow of Events:

1. User selects "Generate Key" from the main menu options panel.
2. User is presented with an input box labeled "Input Key Size".
3. User inputs desired key size into the box.
4. System processes the request and generates the public-private key pair.
5. System displays a notification informing the user that the key generation process was successful.
6. User is presented with options "Export to file" and "Copy to clipboard" for the signature.
7. User selects desired option to either save the keys to a file or copy them to clipboard.

Alternative flows:

3a. User inputs an invalid key size.

3a1. System warns user about the invalid input and prompts them to enter a valid key size again.

5a. System encounters an error during key generation.

5a1. System displays an error message and prompts the user to try again.

Create Signature Use Case

Flow of Events:

1. User selects "Sign message" from the main menu options panel.
2. User is presented with text boxes labeled "Input Private Key" and "Input Message".
3. User inputs their private key and the message they wish to sign.
4. User selects the desired signature scheme from options like "PKCS#1 v1.5 Signature", "ANSI X9.31 rDSA", etc.
5. System processes the input and computes the digital signature.
6. System displays a notification informing the signer that the signing process was successful.
7. User is presented with options "Export to file" and "Copy to clipboard".
8. User selects desired option to either save the keys to a file or copy them to clipboard.

Alternative flows:

3a. User inputs an invalid or mismatched private key.

3a1. System warns user about the invalid input and prompts them to enter a valid key.

5a. System encounters an error during signature creation.

5a1. System displays an error message and prompts the user to try again.

7a. User selected an ISO/IEC 9796-2 scheme in step 4.

7a1. User is presented with options "Export to file" and "Copy to clipboard" for the computed signature and additionally if applicable. a computed non recoverable portion of their initially submitted message.

Verify Signature Use Case

Flow of Events:

1. User selects "Verify Signature" from the main menu options panel.
2. User selects the desired signature scheme from options like "PKCS#1 v1.5 Signature", "ANSI X9.31 rDSA", etc..
3. User is presented with options to input the message, its corresponding signature, and the public key
4. User provides all required inputs.
5. System processes the information and verifies the authenticity of the signature.
6. System displays a notification with the result, either confirming the authenticity or notifying of a mismatch.

Alternative flows:

3a. User selected the ISO/IEC 9796-2 scheme 1 with full message recovery option scheme in step 2.

3a1. System greys out box requiring message input so that user cannot input a message.

3b. User selected the ISO/IEC 9796-2 scheme 1 with partial message recovery option scheme in step 2.

3b1. System changes the displayed label for message input to non-recoverable message portion.

4a. User inputs mismatched or incorrect information.

4a1. System warns user about the incorrect input and suggests rechecking the inputs.

5a. System encounters an error during verification.

5a1. System displays an error message and prompts the user to try again.

6a. User selected an ISO/IEC 9796-2 scheme in step 2.

6a1. User is presented with options "Export to file" and "Copy to clipboard" for the computed signature and additionally if applicable a recovered portion of a message submitted some time in the past to the signature generation process.

2.1.4 Acceptance Tests

1. Key Pair Generation:

1. Open the application and locate the key generation section.
2. Input a valid key size into the provided text box and submit.
3. If key size is invalid no key is issued and the user is informed to try again.
4. Observe that no exceptions or errors are displayed during the key generation process.
5. If there are errors during key generation the user is informed to try again.
6. Confirm that a notification is presented to the user upon successful key generation.
7. Check if there is an option to save the generated key pair to a file and perform a successful save.

2. Digital Signature Generation:

1. Locate the signature generation section in the application.
2. Input a test message into the provided text box.
3. Use the browse option to provide a valid private key file.
4. If empty message or invalid file is provided, the user is informed to try again.
5. Ensure no errors or exceptions are displayed during the signing process.

6. Confirm that a notification is presented to the signer upon successful signature generation.
7. Check for options to either copy the signature to clipboard or save it to a file and verify both functionalities.

3. Digital Signature Verification:

1. Locate the signature verification section in the application.
2. Use the text box or file browse option to input the original test message.
3. Use the browse option to provide the generated signature file.
4. If empty message or invalid file is provided, the user is informed to try again.
5. Ensure no errors or exceptions are displayed during the verification process.
6. Confirm that a notification is presented to the verifier upon successful verification.

4. Signature and Verification with PKCS#1 v1.5:

1. Set the application to use the PKCS#1 v1.5 Signature Scheme.
2. Sign a test message and verify its signature using the previous steps. Ensure both processes succeed.

5. Signature and Verification with ANSI X9.31 rDSA:

1. Set the application to use the ANSI X9.31 rDSA Signature Scheme.
2. Sign a test message and verify its signature using the previous steps. Ensure both processes succeed.

6. Signature Generation with ISO/IEC 9796-2:2010 Scheme 1:

1. Set the application to use the ISO/IEC 9796-2:2010 Signature Scheme 1.
2. Locate the signature generation section in the application.
3. Input a test message into the provided text box.
4. Use the browse option to provide a valid private key file.
5. If empty message or invalid file is provided, the user is informed to try again.
6. Ensure no errors or exceptions are displayed during the signing process.
7. Confirm that a notification is presented to the signer upon successful signature generation.
8. Check and verify separate options (copying to clipboard and saving to a file) for the signature.
9. Check and verify separate options (copying to clipboard and saving to a file) for non message portion if if user submitted a sufficiently short message relative to modulus (if message is too short there is no non recoverable portion).

6. Signature Verification with ISO/IEC 9796-2:2010 Scheme 1

1. Set the application to use a variant of ISO/IEC 9796-2:2010 Signature Scheme 1.
2. Locate the signature verification section in the application.
3. Use the browse option to provide the generated signature file.
4. Use the text box or file browse option to input the original test message if user did received a non recoverable message from previous signature generation process
5. Ensure no errors or exceptions are displayed during the verification process.
6. Confirm that a notification is presented to the verifier upon successful verification.
7. Check and verify separate options (copying to clipboard and saving to a file) for recoverable message portion if user submitted a legitimate non recoverable message portion to the verification process in step 4.

2.2 Appendix B.2 Design

2.2.1 Program packages

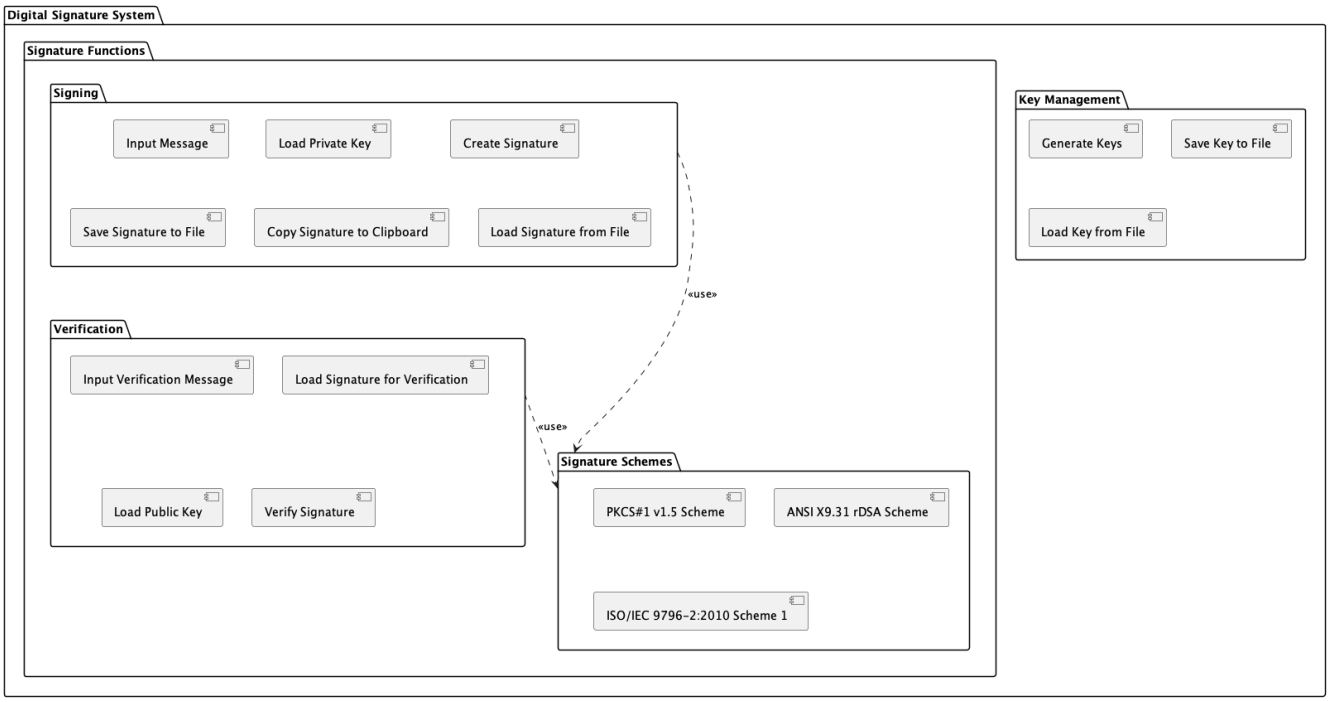

Figure 2.2: POC program Packages

Figure 2.2 depicts the core functionality of the POC program and is in direct alignment with previously elaborated on (see requirements) user activities of signing, creating keys, and verifying, using a specified scheme like PKCS#1-v1.5.

2.2.2 UML sequence diagram

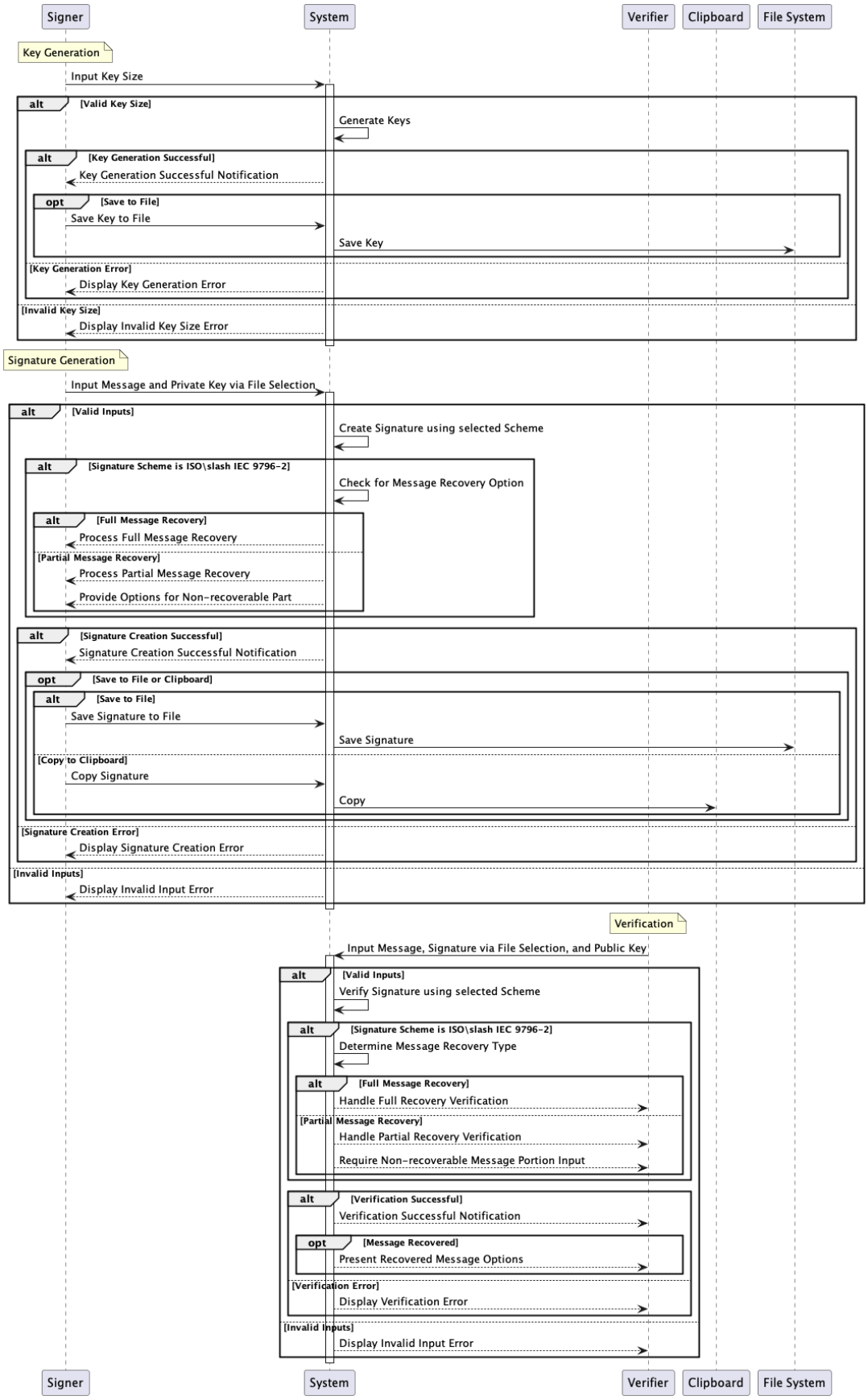

Figure 2.3: UML Sequence Diagram

The above diagram is mostly a high level view of the core behaviour that can be expected to performed by a user of the proof of concept of program. The point at which the diagram departs from this, is the specialised functionality related to the message recovery signature schemes of the ISO standard. These schemes require special consideration because their behaviour differs from the standard digital signature process. The ISO/IEC 9796-2 schemes incorporate message recovery features, where part or all of the original message can be reconstructed from the signature itself. This necessitates additional logic in both the signing and verifying processes.

For partial message recovery, the signer needs to manage the non-recoverable portion of the message, ensuring that it is correctly returned alongside the signature. Subsequently they may then input the non-recoverable portion as part of their interaction with the verification process in attempt to recover the remaining portion of message.

For full message recovery, the entire message is embedded in the signature, eliminating the need for a separate message input during verification but requiring careful handling to extract and validate the message from the signature.

These nuances demand specialised user interfaces and system checks, making the ISO schemes distinct in their interaction and processing requirements within the application.

Initially, the user is prompted to generate cryptographic keys, providing a key size that, if valid, leads to the creation of a private and public key pair. The user can then opt to save these keys onto their file system.

Once keys are in place, the user can sign a message. They input the message into the system and load their private key. If the inputs are correct, the system employs a chosen signature algorithm to create a digital signature, which the user can save or copy to their clipboard. In case of invalid input or an error during signature creation, the user is informed with an error message.

For verification, the user inputs a message, loads the digital signature and the corresponding public key. The system checks the signature against the message using the public key. If the signature is valid, a success notification is displayed; otherwise, the user is alerted to a verification error. Throughout this process, the system guides the user with notifications or error messages based on the success or failure of the operations performed.

2.3 Appendix B.3 Implementation: An Extensible Signature Scheme framework

The foundation of this framework is provided by SigSchemeInterface, an interface that outlines the essential functionalities required for any digital signature scheme. This interface serves as a contract for implementing classes, ensuring that fundamental operations such as signing, verifying, and data conversion between different formats are standardised and consistent.

```
1 public interface SigSchemeInterface {
2
3   byte[] sign(byte[] M) throws SignatureException, DataFormatException;
4
5   boolean verify(byte[] data, byte[] signature) throws SignatureException,
      DataFormatException;
```

```
6
7   BigInteger OS2IP(byte[] EM);
8
9   byte[] I2OSP(BigInteger m) throws IllegalArgumentException;
10
11  BigInteger RSASP1(BigInteger m);
12
13  BigInteger RSAVP1(BigInteger s);
14 }
15
```

Listing 2.1: SigSchemeInterface Interface code

Signing a message m (sign(m)) works as follows:

1. Encode the message: encodeMessage(m).
2. Convert the message to representative integer form: OS2IP(m).
3. Using a private key sk, calculate the signature representative s using RSA: RSASP1(sk,m).
4. Convert the signature (integer) representative back to a byte string: I2OSP(s).

Verification is the reverse, adapted accordingly. Given a signature s (verify(s)):

1. Convert s into an integer representative: OS2IP(s).
2. Using public key vk, calculate the message representative m: RSAVP1(vk,s)
3. Convert m to a byte string: I2OSP(m).
4. Verify the byte string using verifyMessage.

We also note two further methods: 'encodeMessage', an algorithm to format data before performing a cryptographic signing operation on its integer representation, and 'verifyMessage', an algorithm that verifies the signature after a cryptographic verification operation has been performed on its integer representation.

Building upon the SigSchemeInterface, the abstract class SigScheme is introduced as a concrete implementation of the Interface for implementing RSA-based signature schemes. This class provides standardised implementations of the interface's methods and includes additional functionality specific to RSA operations. It encapsulates key components like the RSA modulus, exponent, and message digest, contributing to a modular and extensible design suitable for various RSA-based signature schemes.

```
1
2   /**
3    * Converts a BigInteger to an octet string of length emLen is the byte
     length of the RSA modulus.
4    */
5   public byte[] I2OSP(BigInteger m) throws IllegalArgumentException {
6     return ByteArrayConverter.toFixedLengthByteArray(m, this.emLen);
7   }
8
9   /**
```

```
10     * Calculates the RSA signature of a given message representative by
       raising it to the power of
11     * the private exponent as outlined by the RSA algorithm.
12     */
13    public BigInteger RSASP1(BigInteger m) {
14      BigInteger s = m.modPow(this.exponent, this.modulus);
15      return s;
16    }
17
18    /**
19     * Facilitates the verification of RSA signature by raising it to the power
         of the public exponent
20     * as outlined by the RSA algorithm.
21     */
22    public BigInteger RSAVP1(BigInteger s) {
23      return this.RSASP1(s);
24    }
25
26  }
27
```

Listing 2.2: Implementation of RSA primitives

We note that the implementation of RSAVP1 re-uses RSASP1. This design choice is viable because the exponent field in the SigScheme class is generic and can represent either the public or private exponent, depending on the type of key the scheme is instantiated with. This reuse simplifies the code and centralises the modular exponentiation logic in one place.

The implementation of signing and verification processes essentially mirrors the pseudocode from the start of the section.

```
1   // Abstract method to be implemented by derived classes for encoding
2   protected abstract byte[] encodeMessage(byte[] M) throws
      DataFormatException;
3
4   public boolean verifyMessage(byte[] M, byte[] S)
5       throws DataFormatException {
6     BigInteger s = OS2IP(S);
7     BigInteger m = RSAVP1(s);
8     byte[] EM;
9     try {
10      EM = I2OSP(m);
11    } catch (IllegalArgumentException e) {
12      return false;
13    }
14
15    byte[] EMprime = encodeMessage(M);
16
17    return Arrays.equals(EM, EMprime);
18  }
19
```

Listing 2.3: Signature Scheme specialisation methods

The SigScheme class provides two further additional methods as compared to the interface with encodeMessage and verifyMessage. The encodeMessage method is left abstract because

the encoding of messages is what defines different signature scheme. On the other hand, verifyMessage has a standardised implementation because, in many cases, it simply reverses the signing process. Since the core logic of this reversal often follows a standard procedure that is largely independent of the specifics of the message encoding, it is feasible to provide a standardised implementation for verifyMessage within the SigScheme class.

Message Recovery

Message recovery schemes, such as ISO/IEC 9796-2 Scheme 1, represent special, thereby challenging the notion of standardised implementations for key methods like sign and verifyMessage.

```
1  public abstract class SigScheme implements SigSchemeInterface {
2
3    byte[] nonRecoverableM;
4
5    byte[] recoverableM;
6  }
7  public class ISO_IEC_9796_2_SCHEME_1 extends SigScheme {
8    @Override
9    public byte[] sign(byte[] M) throws DataFormatException {
10     byte[] S = super.sign(M);
11     // Extract m2 from the original message M using the computed m2's length
12     if (m2Len > 0) {
13       nonRecoverableM = Arrays.copyOfRange(M, m1Len - m2Len, m1Len);
14     }
15     return S;
16   }
17 }
18
```

Listing 2.4: Implementation changes for Message Recovery Schemes

The sign method for these schemes needs to be able to set the non recoverable message field of the class to allow the respective portion message to be returned to the user as part of the conclusion to the signature creation process. Hence, the ISO/IEC 9796-2 Scheme 1 class includes an overridden implementation of sign modified accordingly to account for this.

Along these same lines the verifyMessage method which we omit for brevity has to incorporate logic related to the separation of the message into recoverable/non-recoverable parts into the verification process. During verification, the verifyMessage method must correctly reconstruct the original message from these two parts.

2.4 Appendix B.4 Testing

2.4.1 Appendix B.4A Integration Testing

My approach towards integration testing was tailored to ensure that each of the application modules functioned correctly within their respective Model-View-Controller (MVC) frameworks. Utilising TestFX [4], a testing framework for JavaFX applications, the testing concentrated on the internal workings of each module, examining how well the MVC components within a single module interacted with each other.

The first step of this testing was to ensure that the main controller effectively managed transitions from the application-level main menu into the different functional modules. This is the only form of inter-module communication that the application utilises.

The primary role of TestFX in this scenario was to automate interactions within each module, testing the cohesion between the Model, View, and Controller layers. For example, in the key generation module, TestFX helped ensure that the user input in the View layer was accurately processed by the Controller, and the resulting data was correctly managed and reflected by the Model. This pattern was replicated in the signature module that encapsulates the signature creation and verification functionalities as well.

2.4.2 Appendix B.4B System Testing

In the context of this proof of concept, my approach to system testing has been intentionally focused and concise, targeting primarily the core functional requirements. Understanding that exhaustive testing is not the goal at this stage, I concentrated on verifying the key operations that are crucial for my application: key generation, signature creation, and signature verification.

During the system testing phase, I employed a targeted strategy, prioritising Critical Path Testing. This involved ensuring that the essential functions of the application – generating keys, creating signatures, and verifying them – performed accurately under normal use scenarios. Alongside this, I conducted Happy Path Testing to confirm that the application behaves as expected in ideal conditions.

Given the scope of this project, comprehensive testing for errors and edge cases was not a primary focus. However, I did engage in some basic Negative Testing to ensure that the system could handle common user input errors gracefully. This approach provided me with a sufficient level of confidence in the application's operational integrity and usability, particularly for a proof of concept.

Type of Testing	**Module Scope**	**Goal of tests**	**Test Objective**	**Technique**	**Completion Criteria**
Functional Testing	All Modules.	The goals of these tests are to verify acceptance of data, its retrieval, and the correct adoption of requirement related logic	Ensure entry and retrieval of data along expected navigation of an application	Execute function, using valid/invalid data, ensuring: • When valid /invalid data is inputted, respectively, the expected results occur, or corresponding error message is displayed. • Each requirement is met.	All planned tests have been executed

Table 2.2: **Main Menu Test Cases**

Test ID	**Prerequisites**	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**
MainMenu-001	Application is launched and the user is presented with the main menu.	1. Click on the "[K] Generate Keys" button.	N/A	The application should navigate to the key generation page without errors.	Pass
MainMenu-002	Application is launched and the user is presented with the main menu.	1. Click on the "[S] Sign Document" button.	N/A	The application should navigate to the signature creation page without errors.	Pass
MainMenu-003	Application is launched and the user is presented with the main menu.	1. Click on the "[V] Verify Signature" button.	N/A	The application should navigate to the signature verification page without errors.	Pass

Table 2.3: **Key Generation Test Cases**

Test ID	**Prerequisites**	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**
KeyGen-001	Application is installed and operational; the user is on the Key Generation page.	1. Navigate to the "Generate Keys" section. 2. Enter a valid bit size in the input field. 3. Click the "Generate Keys" button.	1024, 1024	The system should generate a key pair using the specified bit sizes without errors.	Pass

KeyGen-002	Application is installed and operational; the user is on the Key Generation page.	1. Navigate to the "Generate Keys" section. 2. Enter a string of special characters in the input field.	@#%&*[($	The system should not accept the input and display an error message indicating that only numerical bit sizes are valid	Pass
KeyGen-003	Application is installed and operational; the user is on the Key Generation page.	1. Navigate to the "Generate Keys" section. 2. Enter an excessively long string of numbers in the input field. 3. Click the "Generate Keys" button.	A string of numbers exceeding normal bit size lengths (e.g., 1000 digits).	The system should reject the input and display an error message indicating that the bit size is too long and not valid.	Fail.

KeyGen-004	Application is installed and operational; the user is on the Key Generation page.	1. Navigate to the "Generate Keys" section. 2. Enter alphanumeric characters in the input field. 3. Click the "Generate Keys" button.	abc123	The system should not accept the input and should display an error message that only numeric values are valid.	Pass
KeyGen-201	Application is installed and operational; the user has successfully generated keys using the "Generate Keys" feature.	1. After key generation, click on the "Export Private Key" button. 2. Check the application's default save location for the presence of the new signature file.	N/A (The action uses the application's UI)	The signature file is automatically saved to the default location specified by the application. The file should contain the correct signature data, formatted as expected for a digital signature.	Pass

Table 2.4: **Signature Creation Test Cases**

Test ID	**Prerequisites**	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**
Sign-001	User is on the "Sign" page of the application.	1. Click the "Import Text..." button. 2. Select a valid text file to import for signing. 3. Verify that the text box is replaced with the name of the imported file and a green check-mark is displayed. 4. Click the "Import Private Key" button and select a valid private key. 5. Choose a signature scheme from the dropdown menu if available. 6. Click "Create Signature".	Valid text file for import, valid private key file.	The application should successfully complete the signature generation process	Pass. Screen transitions to notification panel with success message

Continued on the next page

Table 2.4 (continued): Signature Creation Test Cases

Test ID	Prerequisites	Test Steps	Test Data	Expected Result	Actual Result
Sign-002	User is on the "Sign" page of the application without any key or text pre-loaded.	1. Manually enter text into the "ENTER TEXT TO SIGN:" field. 2. Click "Create Signature" without importing a private key.	"Example text to sign"	The application should prompt the user to import a private key before allowing the signature creation to proceed.	Pass. Error pop-up informing input must be provided for all fields is displayed
Sign-003	User is on the "Sign" page of the application. A valid private key is already imported.	1. Click the "Import Text..." button. 2. Select an invalid file format or a corrupted text file. 3. Attempt to create a signature.	Invalid or corrupted file.	The application should display an error message indicating the file is not valid for import.	Pass. File chooser limits its choice of file to be imported to text files.
Sign-004	User is on the "Sign" page with a valid text and scheme selected.	1. Click "Import Private Key" and select an invalid or corrupted private key file. 2. Attempt to sign the message.	Invalid or corrupted private key file.	The application should display an error message indicating the private key file is not valid for import.	Fail.

Continued on the next page

Table 2.4 (continued): Signature Creation Test Cases

Test ID	Prerequisites	Test Steps	Test Data	Expected Result	Actual Result
Sign-005	User is on the "Sign" page with a valid text, and private key imported	1. Attempt to create a signature without choosing a signature scheme.		The application should display an error message indicating signature scheme needs to be selected.	Pass
Sign-Export-001	User has attempted to create a digital signature but the process failed due to an invalid key or other errors.	1. Attempt to click on the "Export Signature" or "Copy to Clipboard" button after a failed signature creation attempt.	N/A (The action uses the application's UI)	The application should either disable the export/copy functionality	Pass. Error pop-up informing input must be provided for all fields is displayed, so screen does not transition to panel where export/copy functionality is provided.

Continued on the next page

Table 2.4 (continued): Signature Creation Test Cases

Test ID	Prerequisites	Test Steps	Test Data	Expected Result	Actual Result
Sign-Export-002	User has successfully created a digital signature on the "Sign" page.	1. After signature creation, check for any UI indication that the signature is ready to be exported (such as a confirmation message or an enabled "Export" button). 2. If a confirmation message or similar indicator is part of the design, confirm its presence. 3. Proceed with the export or copy operation as designed.	N/A (The action uses the application's UI)	Any UI indicators or messages that should appear post-signature creation to guide the user to export or copy the signature should be present and correct according to the application design.	Pass. Screen transitions to notification panel with success message and export/copy functionality provided

Continued on the next page

Table 2.4 (continued): Signature Creation Test Cases

Test ID	Prerequisites	Test Steps	Test Data	Expected Result	Actual Result
Sign-Export-003	User has successfully created a digital signature on the "Sign" page.	1. After signature creation, click on the "Export Signature" button. 2. Wait for the application to perform the export operation automatically. 3. Check the application's default save location or the location indicated by the application for the presence of the new signature file. 4. Open the signature file with a text editor to verify that it contains the correct signature data.	N/A (The action uses the application's UI)	The signature file is automatically saved to the default location specified by the application.	Pass. File saved to same directory application was run in.

Table 2.5: **Verification Test Cases**

Test ID	**Prerequisites**	**Test Steps**	**Expected Result**	**Actual Result**
Verify-002	User is on the "Verify" page of the application with no files pre-loaded.	1. Manually input text into "ENTER TEXT TO VERIFY:". 2. Click "Import Public Key". 3. Select a valid public key file. 4. Manually input a signature into "ENTER SIGNATURE:". 5. Select the appropriate signature scheme, if applicable. 6. Click "Verify Signature".	The application should accept manual input and the imported public key, perform verification on "Verify Signature" click, and display the result.	Pass. Screen transitions to notification panel with result of verification.
Verify-003	User is on the "Verify" page of the application.	1. Attempt to import an invalid or corrupted text file by clicking "Import Text...". 2. Attempt to verify the signature.	The application should display an error message indicating the file is not valid for import.	Pass. File chooser limits choice of file to be imported to text files.

Continued on the next page

Table 2.5 (continued): Verification Test Cases

Test ID	Prerequisites	Test Steps	Expected Result	Actual Result
Verify-004	User is on the "Verify" page with a valid text and signature imported.	1. Click "Import Public Key" and select an invalid or corrupted public key file. 2. Attempt to verify the signature.	The application should display an error message indicating the public key file is not valid for import.	Pass.
Verify-005	User is on the "Verify" page with a valid text, signature imported and public key imported	1. Attempt to verify the signature without choosing a signature scheme.	The application should display an error message indicating signature scheme needs to be selected.	Fail.

2.4.3 Bug Report

Table 2.6: **Bug Report Table**

Bug ID	Test ID	Test Steps	Expected Result	Actual Result	Solution
Bug-001 Key Generation causes application crash when a number that exceeds what can be accepted as an integer is inputted Bit causes crash .	KeyGen-003	1. Navigate to "Generate Keys". 2. Enter 11111111111. 3. Click "Generate Keys".	System should reject input and display error for long bit size.	Fail: System crashes.	Catch and handle exception thrown by Integer.parseInt call so that error is displayed as intended

Continued on next page

Table 2.6 continued from previous page

Bug ID	Test ID	Test Steps	Expected Result	Actual Result	Solution
Bug-002 Invalid or Corrupted Private Key File causes crash	Sign-004	1. On "Sign" page, click "Import Private Key". 2. Select an invalid or corrupted private key file. 3. Attempt to sign the message.	Error message indicating invalid private key file.	Fail: System crashes.	Correct regex pattern used to identify regex pattern so that error is displayed as intended
Bug-003 Not selecting signature causes application crash	Verify-005	1. On "Verify" page, import valid text, signature, and public key. 2. Attempt to verify without selecting a signature scheme.	Error message for not selecting a signature scheme.	Fail: System crashes.	Modify check for signature scheme drop value to use "!=" to check for equality with null rather than .equals().

Chapter 3: **Appendix B Diary**

Diary Entry: Week of 18th - 24th September 2023

This week was dedicated to writing a first draft of the abstract for my project (PKCS signature scheme) and researching arbitrary precision arithmetic for a library I may look to implement as part of the aims for the project

On Monday, I started writing about its importance, touching on its widespread use and history. Tuesday was a continuation, emphasising why it's such a vital system.

By Wednesday, I added details on potential security issues, particularly focusing on something called the Bleichenbacher attacks. I was initially puzzled about how these attacks affected the signature scheme.

On Thursday and Friday, after more research, I figured out the difference between how these attacks affect encryption and signature aspects of the system. This helped clarify some of my earlier confusion.

Over the weekend, I added insights on why, despite some concerns, many still prefer the PKCS system. I also touched upon a new research finding that supports its use. By Sunday, I detailed my main goals for this project, hoping to create a useful tool that compares different signature schemes.

Next I will clarify whether I should consider implementing a self-made big number library as part of the project and hopefully advance significantly in the creation of the project plan.

Diary Entry - Week of 25th September - 1st October 2023

Met with my supervisor for initial meeting. Discussed potential extensions to the original project specifications and in general what the project entails. Refocused and refined the project plan, emphasising deterministic RSA hash-and-sign schemes, especially PKCS#1 v1.5. Made structural changes to the introduction and abstract, enhancing clarity. Set up the Maven project directory on GitLab and further developed the project timeline. Transitioned all documentation from Microsoft Word to latex, drafting the literature review in the process. By week's end, automated referencing in latex for enhanced efficiency.

Diary Entry - Week of 2nd October - 8th October 2023

This week, I refined and expanded the Risks and mitigation section, established a risk quantification table, and deepened my understanding of digital signature schemes. The literature review for the interim report was integrated, and significant progress was made in drafting the cryptographic foundation of the report. By the weekend, focus was channeled into classifying digital signature schemes and laying out a clear structure for detailed exploration of specific signature schemes in upcoming sessions. Next I will begin writing the introductory section on digital signatures for my report.

Diary Entry - Week of 9th October - 15th October 2023

I Started the week with supervisor meeting, confirming my focus on the POC PKCS Signature for term 1 and was given advice to potentially using the top 1000 English words for the signature program when I sought guidance on the type of data I could provide to be signed. I delved into textbook RSA, highlighting its vulnerabilities. By Friday, I had expanded on RSA's role in digital signatures, introducing potential attacks and Hashed RSA signatures. The weekend saw me laying the foundation for all three schemes considered in the project by

formally defining them. I then began to explore the motivation of provably secure signature schemes.

Diary Entry - Week of 16th October - 22nd October 2023

I started the week attempting to try and understanding trapdoor permutations, especially how they tie into RSA. Following this I began work on enumerating the requirements for the proof of concept program. By the end Friday, I had detailed the user stories and actors for the program with a corresponding a UML use case diagram. During the weekend I first focussed on expanding the motivation for provable security section with subsections on real world implications and limitations. I finished off the week on Sunday by trimming down the report to make it more concise.

Diary Entry - Week of 23rd October - 29th October 2023

The week started with a meeting where I received constructive feedback on my project plan, specifically that I had spent too much time on PKCS#1 v1.5 encryption scheme and Bleichenbacher attacks, which were deemed beyond the project's scope. We clarified the implications of the interim report's word limit, and I was reassured that my report's structure was on the right track, though I was advised against including full software design documents.

I primarily focused on refining my project plan based on feedback. After restructuring my report and creating an appendix for the software requirements of the proof of concept program, I turned my attention to conceptualising and beginning the implementation of the RSA key generation process, culminating in a complete first draft by Friday. The weekend was dedicated to initiating a new chapter on security proof in the report, laying down the foundational concepts and starting to weave them into the project's larger narrative.

Diary Entry - Week of 30th October - 5th November 2023

Focused on enhancing the clarity and structure of my project report, I began by unifying the background concepts for security proofs. I refined the introduction, dividing it into clear aims and objectives. I then pruned excess information from several sections for brevity and clarity, particularly around RSA concepts. I started work on the design phase for the proof of concept program kicking off with a draft outline of the MVC architecture and factory patterns, which I later formalised into a UML class diagram. By the week's end, I finalised design diagrams, integrated them into the report, and refined the section on provable security. The upcoming weeks are now poised for the implementation stage.

Diary Entry - Week of 6th November - 12th November 2023

This week, I had the 4th meeting with my supervisor, where we clarified the required content for the security proofs in my report, focusing on the practical implications for the signature schemes. My progress on the interim report was positively noted, and I announced my intention to submit a draft shortly. I began conceptualising (Wednesday) and then coding the PKCS#1 v1.5 signature scheme (Thursday) with a focus on modularity. By the end of the week, I had not only implemented this scheme but also completed the security proof chapter of my report, emphasising the implications for practical parameter choices. I also improved the key generation process to be more parametrisable, setting a foundation for term 2 work where this is required.

Diary Entry - Week of 13th November - 19th November 2023

This week, I focused on implementing various signature schemes, starting with conceptualising and drafting the ANSI X9.31 signature scheme. Using Test-Driven Development, I developed and refined this implementation, leveraging the modular code structure from the earlier PKCS

scheme.

A significant part of the week involved troubleshooting and resolving issues related to signature verification. In the ANSI implementation, legitimate signatures occasionally failed to verify. The problem was traced to the message encoding method and was fixed by adjusting the first padding byte.

When implementing the ISO/IEC 9796-2 scheme, I encountered a similar issue with signature verification failures. This time, it was due to the first padding byte causing the encoded message's big integer representation to sometimes exceed the modulus size, leading to verification failures. After thorough research and comparison with open-source implementations, I realised the necessity of prepending an initial 0x00 byte to the encoded message array, a Java-specific implementation detail.

The week concluded with a substantial refactoring of the ISO scheme's class structure, simplifying it to a single class that automatically adjusts the recovery mode based on the user's message length.

Diary Entry - Week of 20th November - 26th November 2023

This week, I made substantial progress in both my report and the development of the proof of concept program. I sent a draft of my report to my supervisor and rescheduled our meeting to Friday for feedback. I then focused on developing models for the proof of concept program, beginning with the key generation model and applying the state design pattern effectively. By midweek, I completed all essential models for the program, setting the stage for controller development.

I then moved on to developing the program's views, ensuring they supported the observer design pattern, and completed the implementation of all application views. After receiving positive feedback and suggestions for minor improvements on my report from my supervisor, I advanced to developing the controllers, completing the GenController and initiating the SignatureController.

Over the weekend, I finished the SignatureController, integrating it seamlessly with the views, and developed the MainController to manage the application flow. The application was functionally complete, albeit pending more rigorous testing. I concluded the week by reorganising the project and code directory to better reflect the MVC pattern and separate different functional modules.

Diary Entry - Week of 27th November - 3rd December 2023

This week, I focused on finalising my project for the interim submission I started with integration testing for the mainMenu and Sign view features using TestFX, ensuring the UI and model-view-controller interactions worked correctly. By Tuesday, I had completed all integration testing, including tests for the verify view, and updated the appendix with detailed test cases.

Midweek, I shifted to preparing my project presentation, developing introductory slides and key concept overviews. During this period, I also enhanced the JavaDoc documentation across the project and detailed the system testing results in the appendix.

On Friday, I create a new launch class for the application and generated a fat jar containing the full application (with all classes and dependencies needed to run it). Additionally, I started recording demo videos for the presentation and the final project submission.

Over the weekend, I put the finishing touches on the presentation slides and the demo videos.

I also updated the project's README with detailed run instructions and refined the report to incorporate specifics from my implementation of the signature schemes.

Next, I will organise everything in a manner appropriate for the final submission and clean up any remaining loose ends, ensuring that all elements of the project are polished.

Bibliography

[1] F. S. Foundation, *Gnu multiple precision arithmetic library*, version 6.3.0, 2023. [Online]. Available: `https://gmplib.org/`.

[2] Python Software Foundation, *Python 3 standard library documentation*, 2023. [Online]. Available: `https://docs.python.org/3/library/`.

[3] Oracle Corporation, *Java se 17 & jdk 17: Class biginteger*, 2023. [Online]. Available: `https://docs.oracle.com/en/java/javase/21/docs//api/java.base/java/math/class-use/BigInteger.html`.

[4] TestFX, *Testfx: A javafx testing library*, version 4.0.17, 2023. [Online]. Available: `https://testfx.github.io/TestFX/`.