Term Project Documentation for "Security Toolbox. No more, No Less"

Vladyslav Furda

September 17, 2023

Contents

1 Introduction

This project was chosen by me for several reasons. First of all I wanted to gain knowledge about cryptography functions of C# language. Second I was struggling to implement all those algorithms i was tought in practice. So, this implies main theme of my project: toolbox with several most needed cryptography function including Steganography and Crypto Primitives implementations all coverd with fancy command line interface.

In a nutshell, Security Toolbox is a .NET application that provides a suite of tools for enhancing your digital security. It includes features such as password generation, file encryption/decryption, file hashing, digital signatures, and steganography.

2 Project Overview

Security Toolbox is a Command-Line Interface (CLI) application that was specially designed just to provide users with a suite of basic tools to enhance the security of their everyday usage.

With the increasing threats in out fast pace digital world, having a reliable set of security providing tools is truly paramount, so my program aims to be a one-stop solution for most security-related tasks.

2.1 Objective

The primary objective of the this project is to equip users, especially system administrators and security amateurs, with a comprehensive set of utilities that can help in identifying, mitigating, and preventing different range of cybersecurity threats.

3 System Requirements

3.1 Hardware Requirements

- **Processor:** Intel Core i3 or equivalent, 2.0 GHz or faster.
- **Memory:** Minimum 2 GB RAM.
- **Hard Disk:** 200 MB of free space.

3.2 Software Requirements

- **Operating System:** Windows 10 or later with .NET Framework 4.7.2 or newer.
- **Dependencies:** Microsoft Visual C# Redistributable for Visual Studio 2019 (Windows only).

3.3 Network Requirements

- An active internet connection may be required for initial setup from online repository, future updates, and accessing online features (if any will be).

3.4 Additional Requirements

- **Permissions:** The application may require administrative privileges for operations such as accessing protected directories and files.

3.5 Features

- **Generate Password:** Generate full customizable strong and complex passwords that adhere to security best practices.
- **Encrypt file (AES):** Encrypt sensitive data with military standart AES-256 to protect it from unauthorized access.
- **Decrypt file (AES):** Decrypt encrypted files from previous clause.
- **Encrypt file (AES+RSA):** More advanced encryptiong approach that involves not only using US Military Advances Encryption Standard (AES), but encryptiong its key with another comprehensive assymetric encrypting algorithm RSA.
- **Decrypt file (AES+RSA):** Decrypting key and file from previous clause.
- **Hashing file:** Hashing file using most popular digesting algorithms that allow users to check integrity of their files.
- **Digital Signature:** Sign your documents with cutting-adge digital signing standarts.

- **Steganography:** Hide your messages aroung image and audio files to keep your information away from away from prying eyes.

3.6 Target Audience

While my project (Security Toolbox) is designed mostly for studying purposes, it is quite user-friendly and can be used by anyone, its primary target audience may include:

- System Administrators
- Network Engineers
- Cybersecurity Professionals
- IT Consultants
- Tech-savvy individuals concerned about their digital security

3.7 Future Enhancements

The development team (only me) is continuously working on adding more features to the "Security Toolbox." Future versions will likely include advanced threat detection algorithms or even integration with cloud-based security solutions, and support for another OS.

4 Menu implementations

4.1 Main menu

The program's Main method contains an infinite loop that constantly displays main menu using a new library called Spectre.Console and ShowMainMenu method that is responsible for rendering main menu interface with pretty self-made ASCII art and a list of options.

This menu includes options for generating passwords, encrypting and decrypting files using various algorithms, hashing files, working with digital signatures, and performing steganography. It prompts the user to select an option and then calls the appropriate sub-menu or exits the program based on the user's choice. Users can select options simly by pressing arrows up and down.

Program continues to loop, displaying the main menu after each user action until the user chooses to exit.

```
static void Main()
{
    while (true)
    {
        AnsiConsole.Clear();
```

```
        ShowMainMenu();
    }
}

static void ShowMainMenu()
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine(@"
    __________                          ________
    \______   \_____    ______ ______ / _____/  ____   ____
     |     ___/\__  \   /  ___/ / ___//  \  ___ _/ __ \ /    \
     |    |     / __ \_ \___ \ \___ \ \  \_\  \\  ___/ |  |  \
     |____|    (____  //____ >/____ > \______ / \___ >|___| /
                    \/     \/     \/        \/     \/      \/

                 Security Toolbox. No more, No Less.

   \-----------------------------------------------------------------/
    ");

    Console.ResetColor();

    var MainMenuOption = AnsiConsole.Prompt(
        new SelectionPrompt<string>()
        .HighlightStyle(Spectre.Console.Color.Green)
            .PageSize(10)
            .AddChoices(new[]
            {
                "[[0]] Generate Password",
                "[[1]] Encrypt file (AES)",
                "[[2]] Decrypt file (AES)",
                "[[3]] Encrypt file (AES+RSA)",
                "[[4]] Decrypt file (AES+RSA)",
                "[[5]] Hashing file",
                "[[6]] Digital Signature",
                "[[7]] Steganography",
                "[[99]] Exit"
            }));

    switch (MainMenuOption)
    {
        case "[[0]] Generate Password":
            GeneratePasswordMenu();
            break;
        case "[[1]] Encrypt file (AES)":
           EncryptFileAESMenu();
            break;
        case "[[2]] Decrypt file (AES)":
            DecryptFileAESMenu();
            break;
```

```
            case "[[3]] Encrypt file (AES+RSA)":
                EncryptFileMenu();
                break;
            case "[[4]] Decrypt file (AES+RSA)":
                DecryptFileMenu();
                break;
            case "[[5]] Hashing file":
                HashFileMenu();
                break;
            case "[[6]] Digital Signature":
                SignatureMenu();
                break;
            case "[[7]] Steganography":
                SteganographyMenu();
                break;
            case "[[99]] Exit":
                AnsiConsole.Clear();
                System.Environment.Exit(0);
                break;
        }
    }
```

4.2 Password generation menu

Code snippet below represents a nice looking menu for generating passwords. It starts by prompting the user for the desired password length and options to include lowercase letters, uppercase letters, numbers, and special characters. It then uses a password generator with given parameters to create a random password and assess its strength. Entropy calculator used to show user about created password's strength.

Optionally, code allows the generated password to be copied to the clipboard for a limited time (using TextCopy), displaying a progress bar (using Spectre.Console rendering features) during the copying process.

```
    //AnsiConsole.Clear();

    AnsiConsole.MarkupLine($"[rapidblink green] You're now in password
        generating mode [/]\n");


    var length = AnsiConsole.Prompt(
                new TextPrompt<int>("[bold] 1. Enter password length:
                    [/]")
                    .Validate(value =>
                    {
                        return value >= 8 ? ValidationResult.Success() :
```

```
                    ValidationResult.Error("[red]The password
                    length must be at least 8[/]");
            }));

Console.WriteLine();

var useLowercase = AnsiConsole.Confirm("- Include lowercase
    letters?");
var useUppercase = AnsiConsole.Confirm("- Include uppercase
    letters?");
var useNumbers = AnsiConsole.Confirm("- Include numbers?");
var useSpecial = AnsiConsole.Confirm("- Include special
    characters?");

PasswordGenerator passwordGenerator = new PasswordGenerator();
EntropyCalculator entropyCalculator = new EntropyCalculator();
//var password = "1234124";
//var password = GeneratePassword(false, false, true, true, length);
var password = PasswordGenerator.GeneratePassword(useLowercase,
    useUppercase, useNumbers, useSpecial, length);

Console.WriteLine();
AnsiConsole.MarkupLine($"[bold]2. Generated Password:[/] [bold
    yellow]{password}[/]\n");
// This will display the storage size in bits based on the
    assumption each character is encoded in 8 bits.
AnsiConsole.MarkupLine($"- Storage size of generated password is
    {password.Length * 8} bits");
double entropyPerCharacter =
    EntropyCalculator.CalculateStringEntropy(password);
AnsiConsole.MarkupLine($"- Entropy of generated password is
    {entropyPerCharacter * password.Length}");
Console.WriteLine();

var Clipboard = AnsiConsole.Confirm("[bold] 3. Do you want to copy
    password to clipboard for 10 seconds? [/]");
if (Clipboard)
{
    ClipboardService.SetText(password);
    AnsiConsole.Progress()
    .Columns(new ProgressColumn[] { new SpinnerColumn(), new
        ProgressBarColumn(), new PercentageColumn(), new
        RemainingTimeColumn() })
    .Start(ctx =>
    {

        var task = ctx.AddTask("[green]Counting...[/]", new
            ProgressTaskSettings
        {
            MaxValue = 10
```

```
        });

        for (int i = 0; i < 10; i++)
        {
            if (task.IsFinished)
            {
                break;
            }

            // Sleep for one second to simulate work
            Thread.Sleep(1000);

            // Increment the task's current value
            task.Increment(1);
        }
    });

    AnsiConsole.MarkupLine($"[bold green]Progress bar finished.
        Clearing clipboard...[/]");

    ClipboardService.SetText(string.Empty); // Empty the clipboard

    AnsiConsole.MarkupLine($"[bold green]Clipboard emptied![/]");
}

Console.ReadKey();
```

4.3 AES Encryption menu

This menu is for AES encryption. It prompts the user to input a file path for encryption, offers options to either generate a password or input if user already has it, then proceeds to encrypt the specified file using the chosen password.

```
static void EncryptFileAESMenu()
{
    //AnsiConsole.Clear();
    Encrypt encrypt = new Encrypt();

    AnsiConsole.MarkupLine($"[rapidblink green] You're now in
        encryption mode [/]\n");

    var filePath = AnsiConsole.Prompt(
              new TextPrompt<string>("[bold] 1. Enter the path to
                  the file you want to encrypt: [/]"));

    if (File.Exists(filePath))
    {
```

```
    string encryptedFilePath = filePath + ".aes";

    var passMode = AnsiConsole.Prompt(
   new SelectionPrompt<string>()
       .Title("Choose way to enter the password:")
       .PageSize(10)
       .HighlightStyle(Spectre.Console.Color.Green)
       .AddChoices(new[]
       {
            "[[0]] Generate password",
            "[[1]] Enter password"
       }));

    string password = "hardcodedtemppassword";

    switch (passMode)
    {
        case "[[0]] Generate password":
            password = PasswordGenerator.GeneratePassword(true,
                true, true, true, 50);
            AnsiConsole.MarkupLine($"[bold] 2. Generated password
                is: {password} [/]");

            break;
        case "[[1]] Enter password":
            password = AnsiConsole.Prompt(
            new TextPrompt<string>("[bold] 2. Enter the password
                for encryption: [/]"));
            break;
    }

    Encrypt.EncryptFile(filePath, encryptedFilePath, password);

    AnsiConsole.MarkupLine($"[bold] 3. File succesfully encrypted
        and saved to {encryptedFilePath} [/]\n");
}
else
{
    Console.WriteLine();
    AnsiConsole.MarkupLine("[red]File not found![/]\n");
}


AnsiConsole.Prompt(
    new SelectionPrompt<string>()
        .Title("Choose an action:")
        .PageSize(10)
        .AddChoices(new[]
        {
            "Return to main menu"
```

```
            }));
    }
```

4.4 AES Decryption menu

DecryptFileAESMenu handles a menu for decrypting AES-encrypted files. It first prompts the user to enter the path to the encrypted file and a decryption password. If the file exists, it decrypts it and displays a success message. If the file doesn't exist, it informs the user about it.

```
    static void DecryptFileAESMenu()
    {
        //AnsiConsole.Clear();
        Decrypt decrypt = new Decrypt();

        AnsiConsole.MarkupLine($"[rapidblink red] You're now in
            decryption mode [/]\n");

        var filePath = AnsiConsole.Prompt(
                  new TextPrompt<string>("[bold] 1. Enter the path to
                      the encrypted file you want to decrypt: [/]"));

        if (File.Exists(filePath))
        {
            string originalExtension =
                Path.GetExtension(Path.GetFileNameWithoutExtension(filePath));
            string baseName =
                Path.GetFileNameWithoutExtension(Path.GetFileNameWithoutExtension(filePath));
            string directoryPath = Path.GetDirectoryName(filePath);
            string decryptedFileName =
                $"{baseName}.decrypted{originalExtension}";
            string decryptedFilePath = Path.Combine(directoryPath,
                decryptedFileName);

            var password = AnsiConsole.Prompt(
                new TextPrompt<string>("[bold] 2. Enter decryption
                    password: [/]"));

            Decrypt.DecryptFile(filePath, decryptedFilePath, password);

            AnsiConsole.MarkupLine($"[bold] 3. File succesfully
                decrypted! [/]\n");
        }
        else
        {
            AnsiConsole.MarkupLine("[red]File not found![/]\n");
        }
```

```
    AnsiConsole.Prompt(
        new SelectionPrompt<string>()
            .Title("Choose an action:")
            .PageSize(10)
            .AddChoices(new[]
            {
                "Return to main menu"
            }));
}
```

4.5 File Hashing menu

This menu allows users to hash files using different hashing algorithms. The user is presented with a menu to choose a hashing algorithm, such as popular SHA-1, SHA-2, and MD5.

The user is prompted to enter the path to the file they want to hash. Depending on the selected hashing algorithm, code calls corresponding methods from the Hashing class (e.g., Hashing.HashFileWithMD5, Hashing.HashingFileWithSHA256, or Hashing.HashFileWithSHA1) to hash the specified file. The resulting hash value is displayed to the user.

```
static void HashFileMenu()
{
    //AnsiConsole.Clear();
    Hashing hashing = new Hashing();

    AnsiConsole.MarkupLine($"[rapidblink green] You're now in
        hashing mode [/]\n");

    var hashMode = AnsiConsole.Prompt(
       new SelectionPrompt<string>()
           .Title("[bold] 1. Choose hashing alrogithm: [/]")
           .HighlightStyle(Spectre.Console.Color.Green)
           .PageSize(10)
           .AddChoices(new[]
           {
               "[[1]] SHA-1",
               "[[2]] SHA-2",
               "[[3]] MD5",
           }));

    AnsiConsole.MarkupLine($"[bold] 1. Choose hashing alrogithm:
        {hashMode.Substring(5)}[/]");
```

```
        var filePath = AnsiConsole.Prompt(
                   new TextPrompt<string>("[bold] 2. Enter the path to
                       the file you want to hash: [/]"));

        switch (hashMode)
        {
            case "[[3]] MD5":
                string md5Hash = Hashing.HashFileWithMD5(filePath);
                AnsiConsole.MarkupLine($"[bold] 3. File hashed
                    succesfully:[/][bold green] {md5Hash} [/]\n");
                break;

            case "[[2]] SHA-2":
                string sha2hash = Hashing.HashingFileWithSHA256(filePath);
                AnsiConsole.MarkupLine($"[bold] 3. File hashed
                    succesfully:[/][bold green] {sha2hash} [/]\n");
                break;

            case "[[1]] SHA-1":
                string sha1Hash = Hashing.HashFileWithSHA1(filePath);
                AnsiConsole.MarkupLine($"[bold] 3. File hashed
                    succesfully:[/][bold green] {sha1Hash} [/]\n");
                break;
        }

        AnsiConsole.Prompt(
            new SelectionPrompt<string>()
                .Title("Choose an action:")
                .PageSize(10)
                .AddChoices(new[]
                {
                    "Return to main menu"
                }));
    }
```

4.6 Steganography menu

This is menu for performing steganography operations, which is the true art of hiding information within other data, such as images or audio files, without visibly altering them. You can't put this in the Louvre or the Hermitage, but it's art of the highest order.

Here's simple overview:

User-friendly menu allows user to choose from four steganography options: Hiding text in an image using the Least Significant Bit (LSB) method (only for .png files). Reading hidden text from an image using the LSB method (only for .png files). Hiding text in an audio file (by modifying every 16th byte, tested with .wav files). Extracting text from an audio file (again, by considering every

16th byte, tested with .wav files).

Depending on the user's choice, the code executes one of the four steganography operations: For hiding text in an image, it prompts the user for the path to the image file and the text to be hidden. It then calls ImageSteganography.embedText(text, filePath) to embed the text within the image. For reading hidden text from an image, it prompts the user for the path to the image file, and it calls ImageSteganography.extractText(imageFrom) to extract and display the hidden text. For hiding text in an audio file, it prompts the user for the path to the audio file and the text to be hidden. It then calls AudioSteganography.HideMessageInAudio(filePath, text) to hide the text within the audio file. For extracting text from an audio file, it prompts the user for the path to the audio file and calls AudioSteganography.ExtractMessageFromAudio(filePath) to extract and display the hidden text.

```
static void SteganographyMenu()
{
    //AnsiConsole.Clear();
    ImageSteganography steganography = new ImageSteganography();
    AudioSteganography audioSteganography = new AudioSteganography();

    AnsiConsole.MarkupLine($"[rapidblink green] You're now in
        steganography mode [/]\n");

    var steganoMode = AnsiConsole.Prompt(
       new SelectionPrompt<string>()
           .Title("Choose variants:")
           .PageSize(10)
           .HighlightStyle(Spectre.Console.Color.Green)
           .AddChoices(new[]
           {
                "[[0]] Hiding text in Image (LSB, .png only)",
                "[[1]] Reading hidden text from Image (LSB, .png
                    only)",
                "[[2]] Hiding text in audio file (every 16th byte to
                    hide the message, .wav tested)",
                "[[3]] Extracting text from audio file (every 16th
                    byte to hide the message, .wav tested)",
           }));

    switch (steganoMode)
    {
        case "[[0]] Hiding text in Image (LSB, .png only)":
            //Console.Clear();
            AnsiConsole.MarkupLine($"[rapidblink green] You're now in
                writing mode [/]\n");
            var filePath = AnsiConsole.Prompt(
                new TextPrompt<string>("[bold] 1. Enter the path to
```

```
            the file you want to hide text into: [/]"));
        var text = AnsiConsole.Prompt(
            new TextPrompt<string>("[bold] 2. Enter the text to be
                hidden: [/]"));


        ImageSteganography.embedText(text, filePath);
        break;

    case "[[1]] Reading hidden text from Image (LSB, .png only)":
        //Console.Clear();
        AnsiConsole.MarkupLine($"[rapidblink red] You're now in
            reading mode [/]\n");
        filePath = AnsiConsole.Prompt(
            new TextPrompt<string>("[bold] 1. Enter the path to
                the file you want to extract text: [/]"));

        Bitmap imageFrom = new Bitmap(filePath);

        //var hiddenText =
            Steganography.ExtractingTextInImage(filePath);
        var hiddenText =
            ImageSteganography.extractText(imageFrom);

        Console.WriteLine($" 2. Hidden text was succesfully
            extracted: {hiddenText}\n");

        break;

    case "[[2]] Hiding text in audio file (every 16th byte to
        hide the message, .wav tested)":
        //Console.Clear();
        AnsiConsole.MarkupLine($"[rapidblink green] You're now in
            writing mode [/]\n");
        filePath = AnsiConsole.Prompt(
            new TextPrompt<string>("[bold] 1. Enter the path to
                the file you want to hide text into: [/]"));
        text = AnsiConsole.Prompt(
            new TextPrompt<string>("[bold] 2. Enter the text to be
                hidden: [/]"));
        AudioSteganography.HideMessageInAudio(filePath, text);
        break;

    case "[[3]] Extracting text from audio file (every 16th byte
        to hide the message, .wav tested)":
        //Console.Clear();
        AnsiConsole.MarkupLine($"[rapidblink red] You're now in
            reading mode [/]\n");

        filePath = AnsiConsole.Prompt(
```

```
                new TextPrompt<string>("[bold] 1. Enter the path to
                    the file you want to extract text: [/]"));
            string extractedMessage =
                AudioSteganography.ExtractMessageFromAudio(filePath);

            Console.WriteLine($" 2. Hidden text was succesfully
                extracted: {extractedMessage}\n");
            //throw new NotImplementedException();
            break;
        }

        AnsiConsole.Prompt(
            new SelectionPrompt<string>()
                .Title("Choose an action:")
                .PageSize(10)
                .AddChoices(new[]
                {
                    "Return to main menu"
                }));
    }
```

4.7 Digital Signature menu

The code snippet defines a menu system for handling digital signatures. Inside the menu, users have two primary options:

1. "Signing document": This option allows users to create a digital signature for a file. It includes sub-options for generating an RSA key pair or uploading a private key.

2. "Verifying sign": This option allows users to verify the authenticity of a signed document. Users need to provide the path to the document, the signature to be verified, and their public key for verification.

Key features and algorithms involved: 1. RSA key pair generation: Users can generate an RSA key pair for digital signing.

2. Digital signature creation: Users can sign a document using their private key.

3. Signature verification: Users can verify the signature of a document using their public key.

4. Interactive menu: The menu system provides an interactive command-line interface for users to navigate and perform actions.

```
    static void SignatureMenu()
    {
        DigitalSignature digitalSignature = new DigitalSignature();

        var signMode = AnsiConsole.Prompt(
           new SelectionPrompt<string>()
```

```
        .Title("Choose variants:")
        .PageSize(10)
        .HighlightStyle(Spectre.Console.Color.Green)
        .AddChoices(new[]
        {
            "[[0]] Signing document",
            "[[1]] Verifying sign"
        }));

switch (signMode)
{
    case "[[0]] Signing document":

        //Console.Clear();
        AnsiConsole.MarkupLine($"[rapidblink green] You're now in
            signing mode [/]\n");

        var keyGenMode = AnsiConsole.Prompt(
   new SelectionPrompt<string>()
        .Title("Choose variants:")
        .PageSize(10)
        .HighlightStyle(Spectre.Console.Color.Green)
        .AddChoices(new[]
        {
            "[[0]] Generate RSA keypair",
            "[[1]] Upload private key"
        }));

        switch (keyGenMode)
        {
            case "[[0]] Generate RSA keypair":
                var keypairPath = AnsiConsole.Prompt(
                 new TextPrompt<string>("[bold] 1. Enter the path
                     where you want to save the keypair file:
                     [/]"));
                DigitalSignature.GenerateKeys(keypairPath);

                break;

            case "[[1]] Upload private key":
                var privateKeyPath = AnsiConsole.Prompt(
                new TextPrompt<string>("[bold] 1. Enter the path
                    to the privateKey.xml file: [/]"));
                DigitalSignature.UploadKeys(privateKeyPath);

                break;
        }


        var filePath = AnsiConsole.Prompt(
```

```
                    new TextPrompt<string>("[bold] 2. Enter the
                        path to the file you want to sign: [/]"));

            DigitalSignature.SignDocument(filePath);

            break;

        case "[[1]] Verifying sign":

            AnsiConsole.MarkupLine($"[rapidblink red] You're now in
                verifying mode [/]\n");

            var docPath = AnsiConsole.Prompt(
                    new TextPrompt<string>("[bold] 1. Enter the
                        path to the document: [/]"));

            var sigPath = AnsiConsole.Prompt(
                    new TextPrompt<string>("[bold] 2. Enter the
                        path to the signature you want to verify:
                        [/]"));

            var keyPath = AnsiConsole.Prompt(
                    new TextPrompt<string>("[bold] 3. Enter the
                        path to your publicKey.xml: [/]"));

            DigitalSignature.VerifyDocumentSignature(docPath,
                sigPath, keyPath);

            break;
    }

    AnsiConsole.Prompt(
        new SelectionPrompt<string>()
            .Title("Choose an action:")
            .PageSize(10)
            .AddChoices(new[]
            {
                "Return to main menu"
            }));

}
```

4.8 AES + RSA Encryption menu

This menu is used for encrypting files using AES and RSA encryption techniques. It generates a key pair, saves it to specified files, takes a file path for encryption, and then encrypts the file, saving the encrypted version and en-

crypted symmetric key into a new files.

```
static void EncryptFileMenu()
{
    AesRsaEncryption aesRsaEncryption = new AesRsaEncryption();
    RSAParameters publicKey;

    AnsiConsole.MarkupLine($"[rapidblink green] You're now in
        encryption mode [/]\n");

    var keypairPath = AnsiConsole.Prompt(
                    new TextPrompt<string>("[bold] 1. Enter the path
                        where you want to save keypair: [/]"));

    using (RSA rsa = new RSACryptoServiceProvider(2048))
    {
        publicKey = rsa.ExportParameters(false);
        RSAParameters privateKey = rsa.ExportParameters(true);

        var publicKeyFullPath = Path.Combine(keypairPath,
            "publicKey.xml");
        var privateKeyFullPath = Path.Combine(keypairPath,
            "privateKey.xml");

        // Save the public key and private key to files (for
            demonstration purposes)
        File.WriteAllText(publicKeyFullPath, rsa.ToXmlString(false));
        File.WriteAllText(privateKeyFullPath, rsa.ToXmlString(true));
    }

    var filePath = AnsiConsole.Prompt(
                    new TextPrompt<string>("[bold] 2. Enter the path
                        to the file to be encrypted: [/]"));

    string encryptedFilePath = filePath + ".bin";
    string encryptedKeyFilePath = encryptedFilePath + ".key";

    // Step 2: Encrypt the file using the EncryptFile method
    AesRsaEncryption.EncryptFile(filePath, encryptedFilePath,
        publicKey);

    AnsiConsole.MarkupLine($"[bold green] File succesfully encrypted
        and saved to: {encryptedFilePath} [/]\n");

    AnsiConsole.Prompt(
              new SelectionPrompt<string>()
                  .Title("Choose an action:")
                  .PageSize(10)
```

```
                .AddChoices(new[]
                {
            "Return to main menu"
                }));
}
```

4.9 AES+RSA Decryption menu

This is menu for decrypting files using AES and RSA encryption. It begins by prompting the user to enter the paths to a private asymmetric key, an encrypted file, and an encrypted asymmetric key file. After obtaining these inputs, it decrypts the file using the specified keys and saves the decrypted result to a new file. The user is then informed of the successful decryption and the location of the decrypted file.

```
static void DecryptFileMenu()
{
    AesRsaEncryption aesRsaEncryption = new AesRsaEncryption();
    RSAParameters privateKey;

    AnsiConsole.MarkupLine($"[rapidblink red] You're now in
        decryption mode [/]\n");

    var rsaKeypairPath = AnsiConsole.Prompt(
                new TextPrompt<string>("[bold] 1. Enter the path
                    to private assymetric key (.xml): [/]"));

    using (RSA rsa = new RSACryptoServiceProvider())
    {
        rsa.FromXmlString(File.ReadAllText(rsaKeypairPath));
        privateKey = rsa.ExportParameters(true);
    }

    var encryptedFilePath = AnsiConsole.Prompt(
                new TextPrompt<string>("[bold] 2. Enter the path
                    to encrypted file (.bin): [/]"));

    var encryptedKeyFilePath = AnsiConsole.Prompt(
                new TextPrompt<string>("[bold] 3. Enter the path
                    to encrypted assymetric key file (.key):
                    [/]"));

    string originalExtension =
        Path.GetExtension(Path.GetFileNameWithoutExtension(encryptedFilePath));
    string baseName =
        Path.GetFileNameWithoutExtension(Path.GetFileNameWithoutExtension(encryptedFilePath));
```

```
        string directoryPath = Path.GetDirectoryName(encryptedFilePath);
        string decryptedFileName =
            $"{baseName}.decrypted{originalExtension}";
        string decryptedFilePath = Path.Combine(directoryPath,
            decryptedFileName);

        AesRsaEncryption.DecryptFile(encryptedFilePath,
            encryptedKeyFilePath, decryptedFilePath, privateKey);

        AnsiConsole.MarkupLine($"[bold green] File succesfully encrypted
            and saved to: {decryptedFilePath} [/]\n");
    }
```

5 Function implementations

5.1 Dependencies

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;
using System.Text;
using System.Drawing;
using System.Drawing.Imaging;
using Spectre.Console;
```

5.2 Password Generation

Here is defined two classes: PasswordGenerator and EntropyCalculator. First class, PasswordGenerator is responsible for generating passwords based on certain criteria.

It takes as input a set of boolean flags (useLowercase, useUppercase, useNumbers, and useSpecial) to specify which character types should be included in the generated password, and an integer length to specify the password's length. It first checks if the input parameters are valid and throws an exception if not. Then, it constructs a character pool based on the selected character types. Next, it generates random bytes to create a password of the specified length. Finally, it maps these random bytes to characters in the charPool and returns the resulting password as a string.

EntropyCalculator calculates the entropy of a given string, which measures the unpredictability or randomness of the string. It takes a string input as input and initializes a dictionary (charCounts) to count the occurrence of each character in the input string. Then calculates the probabilities of each character

occurring in the string and uses these probabilities to compute the Shannon entropy.

```
public class PasswordGenerator
{
    private static readonly string LowercaseChars =
        "abcdefghijklmnopqrstuvwxyz";
    private static readonly string UppercaseChars =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private static readonly string NumberChars = "0123456789";
    private static readonly string SpecialChars =
        "!@#$%^&*()-_=+=|;:,.<>?";

    public static string GeneratePassword(bool useLowercase, bool
        useUppercase, bool useNumbers, bool useSpecial, int length)
    {
        if (length <= 0 || (!useLowercase && !useUppercase &&
            !useNumbers && !useSpecial))
        {
            throw new ArgumentException("Invalid password settings.");
        }

        string charPool = (useLowercase ? LowercaseChars : "") +
                          (useUppercase ? UppercaseChars : "") +
                          (useNumbers ? NumberChars : "") +
                          (useSpecial ? SpecialChars : "");

        var bytes = new byte[length];
        RandomNumberGenerator.Fill(bytes);
        return new string(bytes.Select(b => charPool[b %
            charPool.Length]).ToArray());
    }
}

public class EntropyCalculator
{
    public static double CalculateStringEntropy(string input)
    {
        if (string.IsNullOrEmpty(input))
            return 0.0; // entropy is zero

        // counting the occurrence of each char
        Dictionary<char, int> charCounts = new Dictionary<char,
            int>();
        foreach (char c in input)
        {
            if (charCounts.ContainsKey(c))
                charCounts[c]++;
```

```
            else
                charCounts[c] = 1;
        }

        // probabilities and then the entropy computing
        double entropy = 0.0;
        int totalChars = input.Length;
        foreach (var count in charCounts.Values)
        {
            double probability = (double)count / totalChars;
            entropy += probability * Math.Log2(probability);
        }

        return -entropy;
    }
}
```

5.3 File Encryption (AES)

First we create instance of aes under using block. Generate salt and set needed parameters from password derivation function. After we write salt to the beginning of the file from very first byte and then open input file and encrypt it. In essence, this inner loop goes through each byte in the input file, encrypts it, and writes the encrypted byte to the output file. When the entire input file has been read and processed, the encrypted content has been written to the output file. Here is a short overlook of features:

1. Salt Generation: A random 128-bit (16-byte) salt is generated using a cryptographic random number generator. This salt adds randomness to the encryption process and ensures that the same input file will produce different encrypted outputs each time.

2. Key Derivation: The user-provided password and the generated salt are used as inputs to the Rfc2898DeriveBytes class to derive encryption key and initialization vector (IV) for AES. The Rfc2898DeriveBytes class uses PBKDF2 (Password-Based Key Derivation Function 2) with SHA-256 as the underlying hash algorithm. All of this strengthens the encryption by making it very intensive to derive the key (brute force resist).

```
public class Encrypt
{
    public static void EncryptFile(string inputFile, string
        outputFile, string password)
    {
        using (Aes aes = Aes.Create())
        {
            byte[] salt = new byte[16]; // 128 bits
```

```
                RandomNumberGenerator.Fill(salt);

                int iterations = 100000;
                Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(password,
                    salt, iterations, HashAlgorithmName.SHA256);
                aes.Key = pdb.GetBytes(32);
                aes.IV = pdb.GetBytes(16);

                using (FileStream fsOutput = new FileStream(outputFile,
                    FileMode.Create))
                {
                    // writing the salt at the begining
                    fsOutput.Write(salt, 0, salt.Length);

                    using (FileStream fsInput = new FileStream(inputFile,
                        FileMode.Open))
                    using (CryptoStream cs = new CryptoStream(fsOutput,
                        aes.CreateEncryptor(), CryptoStreamMode.Write))
                    {
                        int data;
                        while ((data = fsInput.ReadByte()) != -1)
                        {
                            cs.WriteByte((byte)data);
                        }
                    }
                }
            }
        }
    }
```

5.4 File Decryption (AES)

This method is used to decrypt an input file and save the decrypted content to an output file.

Inside the DecryptFile method we can see Aes instance is created, which will be used for decryption. A 16-byte salt is read from the beginning of the input file using FileStream stream which points to inputFile.

The user-provided password, salt, and other parameters are used to derive a 32-byte encryption key and a 16-byte IV using the PBKDF2 algorithm with SHA-256 as the hash algorithm.

A CryptoStream is created to perform the actual decryption in Read mode, using the derived key and IV. The input file is read, decrypted, and the decrypted data is written to the output file in a streaming fashion (just like in encryption mode, but in decrypting way, still byte to byte).

```
    public class Decrypt
```

```
{
    public static void DecryptFile(string inputFile, string
        outputFile, string password)
    {
        using (Aes aes = Aes.Create())
        {
            byte[] salt = new byte[16];

            using (FileStream fsInput = new FileStream(inputFile,
                FileMode.Open))
            {
                fsInput.Read(salt, 0, salt.Length);

                int iterations = 100000;
                Rfc2898DeriveBytes pdb = new
                    Rfc2898DeriveBytes(password, salt, iterations,
                    HashAlgorithmName.SHA256);
                aes.Key = pdb.GetBytes(32);
                aes.IV = pdb.GetBytes(16);

                using (CryptoStream cs = new CryptoStream(fsInput,
                    aes.CreateDecryptor(), CryptoStreamMode.Read))
                using (FileStream fsOutput = new
                    FileStream(outputFile, FileMode.Create))
                {
                    int data;
                    while ((data = cs.ReadByte()) != -1)
                    {
                        fsOutput.WriteByte((byte)data);
                    }
                }

            }
        }
    }
}
```

5.5 File Encryption (AES+RSA)

This method is used to encrypt a file using a combination of symmetric and asymmetric encryption techniques. Here's how it works:

1. The EncryptFile method takes three parameters: - inputFilePath: The path to the file that needs to be encrypted. - outputFilePath: The path where the encrypted file will be saved. - publicKey: RSA public key parameters.

2. Inside the method, a random symmetric key is generated using the Advanced Encryption Standard (AES) algorithm. This symmetric key is used for encrypting the file's data, and a random Initialization Vector (IV) is also generated for AES encryption.

3. The file to be encrypted is opened for reading (inputFileStream), and a new file is created for writing the encrypted data (outputFileStream). These streams are wrapped within a CryptoStream, which enables data to be encrypted as it is written to the output file.

4. The code reads the file in chunks of 1024 bytes, encrypts each chunk using the symmetric key and IV, and writes the encrypted data to the output file.

5. After encrypting the file data, the symmetric key is encrypted using the recipient's RSA public key (publicKey). This is done to securely share the symmetric key, which is faster for encrypting large files compared to asymmetric encryption.

6. The encrypted symmetric key and the IV are combined into a single byte array (encryptedKey.Concat(aes.IV).ToArray()) and saved to a separate file with a ".key" extension. This file contains the encrypted information necessary for decrypting the file.

```
public class AesRsaEncryption
{

    public static void EncryptFile(string inputFilePath, string
        outputFilePath, RSAParameters publicKey)
    {
        // generating a random symmetric key
        using (Aes aes = Aes.Create())
        {
            aes.GenerateKey();
            aes.GenerateIV();

            // encrypting the file data with the symmetric key
            using (FileStream inputFileStream = new
                FileStream(inputFilePath, FileMode.Open,
                FileAccess.Read))
            using (FileStream outputFileStream = new
                FileStream(outputFilePath, FileMode.Create,
                FileAccess.Write))
            using (CryptoStream cryptoStream = new
                CryptoStream(outputFileStream, aes.CreateEncryptor(),
                CryptoStreamMode.Write))
            {
                byte[] buffer = new byte[1024];
                int bytesRead;
                while ((bytesRead = inputFileStream.Read(buffer, 0,
                    buffer.Length)) > 0)
                {
                    cryptoStream.Write(buffer, 0, bytesRead);
                }
            }
```

```
            // encrypting the symmetric key with the recipient's RSA
                public key
            using (RSA rsa = new RSACryptoServiceProvider())
            {
                rsa.ImportParameters(publicKey);
                byte[] encryptedKey = rsa.Encrypt(aes.Key,
                    RSAEncryptionPadding.Pkcs1);

                // encrypted symmetric key and the IV together
                File.WriteAllBytes(outputFilePath + ".key",
                    encryptedKey.Concat(aes.IV).ToArray());
            }
        }
    }
    }
```

5.6 File Decryption (AES+RSA)

1. Code defines a class named AesRsaEncryption with a static method called DecryptFile. This method takes four parameters: inputFilePath (the path to the encrypted file), keyFilePath (the path to the encrypted symmetric key and IV), outputFilePath (the path where the decrypted file will be saved), and privateKey (RSA private key used for decrypting the symmetric key).

2. The encrypted data from the keyFilePath is read into a byte array called encryptedData. This data contains both the encrypted symmetric key and the initialization vector (IV) used for AES encryption.

3. The code separates the encrypted symmetric key and IV from encryptedData using the Take and Skip LINQ methods. The symmetric key is stored in the encryptedKey byte array, while the IV is stored in the iv byte array.

4. Next, the RSA private key provided in the privateKey parameter is used to decrypt the encryptedKey. This is done using the RSA decryption method with the specified padding (PKCS1).

5. After obtaining the decrypted symmetric key, an AES encryption instance is created (Aes.Create()). The decrypted key and IV are set as the AES key and IV, respectively.

6. The code then opens the input file (inputFilePath) and creates an output file (outputFilePath) to write the decrypted data. It also sets up a CryptoStream that will handle the decryption process using the AES decryptor.

7. The code reads data from the input file in chunks, decrypts each chunk using AES, and writes the decrypted data to the output file via the CryptoStream.

```
    public class AesRsaEncryption
    {
        public static void DecryptFile(string inputFilePath, string
            keyFilePath, string outputFilePath, RSAParameters privateKey)
```

```
    {
        byte[] encryptedData = File.ReadAllBytes(keyFilePath);
        byte[] encryptedKey = encryptedData.Take(encryptedData.Length
            - 16).ToArray();
        byte[] iv = encryptedData.Skip(encryptedData.Length -
            16).ToArray();

        // RSA private key to decrypt the symmetric key
        using (RSA rsa = new RSACryptoServiceProvider())
        {
            rsa.ImportParameters(privateKey);
            byte[] decryptedKey = rsa.Decrypt(encryptedKey,
                RSAEncryptionPadding.Pkcs1);

            // using decrypted symmetric key to decrypt the file data
            using (Aes aes = Aes.Create())
            {
                aes.Key = decryptedKey;
                aes.IV = iv;

                using (FileStream inputFileStream = new
                    FileStream(inputFilePath, FileMode.Open,
                    FileAccess.Read))
                using (FileStream outputFileStream = new
                    FileStream(outputFilePath, FileMode.Create,
                    FileAccess.Write))
                using (CryptoStream cryptoStream = new
                    CryptoStream(outputFileStream,
                    aes.CreateDecryptor(), CryptoStreamMode.Write))
                {
                    byte[] buffer = new byte[1024];
                    int bytesRead;
                    while ((bytesRead = inputFileStream.Read(buffer,
                        0, buffer.Length)) > 0)
                    {
                        cryptoStream.Write(buffer, 0, bytesRead);
                    }
                }
            }
        }
    }
```

5.7 Hashing

Here is several simple methods each for exact digesting function. In every of them instance of hash creating, using FileStream to open the file and than just (sha256.ComputeHash(stream)). General features are:

1. Hashing Algorithms: Three different hashing algorithms - SHA-256, SHA-1, and MD5. These algorithms are commonly used for generating fixed-size hash values from input data.

2. Method Structure: Each hash algorithm has a corresponding method: HashingFileWithSHA256, HashFileWithSHA1, and HashFileWithMD5. These methods accept a file path as input and return the computed hash value as a lowercase hexadecimal string.

3. Hash Computation: Inside each method, the respective hash algorithm is applied to the file stream using the ComputeHash method. The resulting hash value is a byte array.

4. Formatting the Hash: The byte array is then converted to a hexadecimal string using BitConverter.ToString(). The resulting string contains hyphens, which are removed using Replace("-", ""). Finally, the string is converted to lowercase using .ToLower().

```
public class Hashing
{
    public static string HashingFileWithSHA256(string filePath)
    {
        using SHA256 sha256 = SHA256.Create();
        using FileStream stream = File.OpenRead(filePath);
        return
            BitConverter.ToString(sha256.ComputeHash(stream)).Replace("-",
            "").ToLower();
    }
    public static string HashFileWithSHA1(string filePath)
    {
        using SHA1 sha1 = SHA1.Create();
        using FileStream stream = File.OpenRead(filePath);
        return
            BitConverter.ToString(sha1.ComputeHash(stream)).Replace("-",
            "").ToLower();
    }

    public static string HashFileWithMD5(string filePath)
    {
        using MD5 md5 = MD5.Create();
        using FileStream stream = File.OpenRead(filePath);
        return
            BitConverter.ToString(md5.ComputeHash(stream)).Replace("-",
            "").ToLower();
    }
}
```

5.8 Image Steganography

Image steganography is a technique of hiding text data within an image, allowing you to hide information within the image without easily detecting it by the naked eye.

The code defines a class called ImageSteganography, which contains methods for embedding text into an image and extracting hidden text from an image. It performs image steganography by manipulating the least significant bits of an image's RGB channels to hide and retrieve text data. It uses a simple algorithm to embed and extract text, making it relatively easy to implement for basic steganography purposes.

The embedText method takes two parameters: the text to be hidden and the path to the input image. It opens the input image, processes each pixel, and embeds the binary representation of the text into the least significant bits (LSBs) of the RGB color channels of the image. It uses a state machine (State) to determine whether it is in the process of hiding text or filling with zeros to mark the end of the text.

The extractText method takes a Bitmap object (representing an image) as input and extracts the hidden text from the LSBs of the image's color channels. It iterates through each pixel, retrieves the LSBs of the RGB channels, and reconstructs the hidden text character by character.

```
public class ImageSteganography
{

    public enum State
    {
        Hiding,
        Filling_With_Zeros
    };

    public static void embedText(string text, string inputImagePath)
    {

        string originalFilePath = inputImagePath;
        string directory = Path.GetDirectoryName(originalFilePath);
        string originalFileName = Path.GetFileName(originalFilePath);
        string newFileName = "hidden_" + originalFileName;
        string newFilePath = Path.Combine(directory, newFileName);

        Bitmap bmp = new Bitmap(inputImagePath);

        State state = State.Hiding;

        // index of the character that is being hidden
        int charIndex = 0;
```

```
// value of the character converted to integer
int charValue = 0;

// index of the color element (R or G or B) that is currently
    being processed
long pixelElementIndex = 0;

// number of trailing zeros that have been added when
    finishing the process
int zeros = 0;

int R = 0, G = 0, B = 0;

for (int i = 0; i < bmp.Height; i++)
{
    for (int j = 0; j < bmp.Width; j++)
    {
        System.Drawing.Color pixel = bmp.GetPixel(j, i);

        // clearing the least significant bit (LSB) from each
            pixel
        R = pixel.R - pixel.R % 2;
        G = pixel.G - pixel.G % 2;
        B = pixel.B - pixel.B % 2;

        // pass through RGB
        for (int n = 0; n < 3; n++)
        {
            // if 8 bits has been processed
            if (pixelElementIndex % 8 == 0)
            {
                // finished when 8 zeros are added
                if (state == State.Filling_With_Zeros && zeros
                    == 8)
                {
                    // apply the last pixel on the image
                    if ((pixelElementIndex - 1) % 3 < 2)
                    {
                        bmp.SetPixel(j, i,
                            System.Drawing.Color.FromArgb(R, G,
                            B));
                    }
                    bmp.Save(newFilePath, ImageFormat.Png);
                }

                if (charIndex >= text.Length)
                {
                    // zeros to mark the end of the text
                    state = State.Filling_With_Zeros;
                }
```

```
    else
    {
        charValue = text[charIndex++];
    }
}

// check which pixel element has the turn to hide
    a bit in its LSB
switch (pixelElementIndex % 3)
{
    case 0:
        {
            if (state == State.Hiding)
            {
                R += charValue % 2;

                // removes the added rightmost bit
                    of the character
                // such that next time we can reach
                    the next one
                charValue /= 2;
            }
        }
        break;
    case 1:
        {
            if (state == State.Hiding)
            {
                G += charValue % 2;

                charValue /= 2;
            }
        }
        break;
    case 2:
        {
            if (state == State.Hiding)
            {
                B += charValue % 2;

                charValue /= 2;
            }

            bmp.SetPixel(j, i,
                System.Drawing.Color.FromArgb(R, G,
                B));
        }
        break;
}
```

```
                pixelElementIndex++;

                if (state == State.Filling_With_Zeros)
                {
                    // increment the value of zeros until it is 8
                    zeros++;
                }
            }
        }
    }
    bmp.Save(newFilePath, ImageFormat.Png);
}

public static string extractText(Bitmap bmp)
{
    int colorUnitIndex = 0;
    int charValue = 0;

    // text that will be extracted
    string extractedText = String.Empty;

    for (int i = 0; i < bmp.Height; i++)
    {
        for (int j = 0; j < bmp.Width; j++)
        {
            System.Drawing.Color pixel = bmp.GetPixel(j, i);

            for (int n = 0; n < 3; n++)
            {
                switch (colorUnitIndex % 3)
                {
                    case 0:
                        {
                            // get the LSB from the pixel element
                                (will be pixel.R % 2)
                            // then add one bit to the right of the
                                current character
                            charValue = charValue * 2 + pixel.R % 2;
                        }
                        break;
                    case 1:
                        {
                            charValue = charValue * 2 + pixel.G % 2;
                        }
                        break;
                    case 2:
                        {
                            charValue = charValue * 2 + pixel.B % 2;
                        }
                        break;
```

```
                    }

                    colorUnitIndex++;

                    if (colorUnitIndex % 8 == 0)
                    {
                        charValue = reverseBits(charValue);

                        if (charValue == 0)
                        {
                            return extractedText;
                        }

                        char c = (char)charValue;

                        extractedText += c.ToString();
                    }
                }
            }
        }
        return extractedText;
    }

    public static int reverseBits(int n)
    {
        int result = 0;

        for (int i = 0; i < 8; i++)
        {
            result = result * 2 + n % 2;

            n /= 2;
        }
        return result;
    }
}
```

5.9 Audio Steganography

Audio Steganography allows you to hide a text message within an audio file (specifically, a WAV file) and also extract a hidden message from such a file. Here's how it works:

1. Header and Byte Interval: The code defines two constants: HEADER_SIZE (set to 4, representing 32 bits) and BYTE_INTERVAL (set to 16). The header size is used to store the length of the hidden message, and the byte interval specifies how often to change the audio data to hide the message.

2. Hiding Message in Audio: The HideMessageInAudio method takes the path of an input audio file and a message as parameters. It first converts the

message into an array of bytes and reads the audio file into an array of bytes as well.

3. Hiding Message Length: It starts at a specific index (44) to skip the header of the WAV file. Then, it iterates through the message length in bits, one bit at a time, and modifies the least significant bit (LSB) of certain bytes in the audio data to store the message length.

4. Hiding Message Itself: After storing the message length, the code proceeds to hide the actual message in the audio data. It iterates through the message byte by byte and bit by bit, similarly modifying the LSB of certain bytes in the audio data to hide the message.

5. Saving the Modified Audio: Once the message is hidden, the modified audio data is written to a new audio file with a name prefixed by "hidden_."

6. Extracting Message from Audio: The ExtractMessageFromAudio method takes the path of an audio file as input. It reads the audio data and starts at index 44 to skip the header.

7. Extracting Message Length: It extracts the message length from the LSB of certain bytes in the audio data, following the same bit-by-bit approach used for hiding the length.

8. Extracting Message Itself: After determining the message length, the code extracts the message bit by bit, similar to how it was hidden, and reconstructs the hidden message.

9. Returning the Extracted Message: Finally, the extracted message is converted from bytes to a string and returned.

```
class AudioSteganography
{

    const int HEADER_SIZE = 4; // 32 bits to store message length
    const int BYTE_INTERVAL = 16; // change every 16th byte

    public static void HideMessageInAudio(string inputAudioPath,
        string message)
    {
        byte[] messageBytes = Encoding.ASCII.GetBytes(message);
        byte[] audioBytes = File.ReadAllBytes(inputAudioPath);

        int audioIndex = 44; // start at the data part of the WAV
            file while skipping the header (thanks to Jezek's
            lectures)

        // hiding message length in first few bytes
        for (int i = 0; i < HEADER_SIZE; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                byte bit = (byte)((messageBytes.Length >> (i * 8 + j))
```

```
                & 1);
            audioBytes[audioIndex] =
                (byte)((audioBytes[audioIndex] & 0xFE) | bit); //
                masking lsb and writing bit
            audioIndex += BYTE_INTERVAL;
        }
    }

    // hiding message itself
    for (int i = 0; i < messageBytes.Length; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            byte bit = (byte)((messageBytes[i] >> j) & 1);
            audioBytes[audioIndex] =
                (byte)((audioBytes[audioIndex] & 0xFE) | bit);
            audioIndex += BYTE_INTERVAL;
        }
    }

    string originalFilePath = inputAudioPath;
    string directory = Path.GetDirectoryName(originalFilePath);
    string originalFileName = Path.GetFileName(originalFilePath);
    string newFileName = "hidden_" + originalFileName;
    string newFilePath = Path.Combine(directory, newFileName);

    File.WriteAllBytes(newFilePath, audioBytes);
}

public static string ExtractMessageFromAudio(string
    inputAudioPath)
{
    byte[] audioBytes = File.ReadAllBytes(inputAudioPath);

    int audioIndex = 44; // skipping the header again

    // extract message length (32 bits) from first few bytes
    int messageLength = 0;
    for (int i = 0; i < HEADER_SIZE; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            byte bit = (byte)(audioBytes[audioIndex] & 1);
            messageLength |= (bit << (i * 8 + j));
            audioIndex += BYTE_INTERVAL;
        }
    }

    byte[] messageBytes = new byte[messageLength];
```

```
        // Extract message
        for (int i = 0; i < messageLength; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                byte bit = (byte)(audioBytes[audioIndex] & 1);
                messageBytes[i] |= (byte)(bit << j);
                audioIndex += BYTE_INTERVAL;
            }
        }

        return Encoding.ASCII.GetString(messageBytes);
    }
}
```

5.10 Digital Signature

1. Key Generation: we have instance of the RSACryptoServiceProvider with a 2048-bit key size to generate RSA public and private key pairs. The GenerateKeys method exports these keys as XML strings and saves them to specified files (publicKey.xml and privateKey.xml).

2. Key Upload: The UploadKeys method allows you to read a private key from a file, enabling you to sign documents using a previously generated key pair.

3. Document Signing: The SignDocument method takes the path to a document as input, checks if a private key is available, and signs the document's data using the private key. The resulting digital signature is saved in a file with a ".sig" extension appended to the original document's filename.

4. Signature Verification: The VerifyDocumentSignature method is used to verify the authenticity of a signed document. It takes the paths to the document, the signature file, and the path to the corresponding public key as input. It reads the public key, verifies the signature against the document data, and returns whether the signature is valid or not.

5. RSA Signature Generation and Verification: The SignData method is used internally to sign data using the private key, and the VerifyData method is used for verifying the signature against the data using the corresponding public key. Both methods employ the RSA algorithm with SHA-256 hashing and PKCS1 padding.

```
public class DigitalSignature
{
    static RSACryptoServiceProvider rsa = new
        RSACryptoServiceProvider(2048);
    static RSAParameters publicKey;
    static RSAParameters privateKey;
```

```
public static void GenerateKeys(string publicKeyPath)
{
    publicKey = rsa.ExportParameters(false);
    privateKey = rsa.ExportParameters(true);
    var publicKeyXml = rsa.ToXmlString(false);
    var privateKeyXml = rsa.ToXmlString(true);

    var publicKeyFullPath = Path.Combine(publicKeyPath,
        "publicKey.xml");
    var privateKeyFullPath = Path.Combine(publicKeyPath,
        "privateKey.xml");
    File.WriteAllText(publicKeyFullPath, publicKeyXml);
    File.WriteAllText(privateKeyFullPath, privateKeyXml);

    AnsiConsole.MarkupLine($"[bold green invert] RSA keys
        generated and keypair saved to
        publicKey.xml/privateKey.xml [/]\n");
}

public static void UploadKeys(string privateKeyPath)
{
    var privateKeyXml = File.ReadAllText(privateKeyPath);
    rsa.FromXmlString(privateKeyXml);

    privateKey = rsa.ExportParameters(true);
}

public static void SignDocument(string documentPath)
{
    if (privateKey.D == null)
    {
        AnsiConsole.MarkupLine($"[bold red] Please generate RSA
            keys first. [/]\n");
        return;
    }

    if (!File.Exists(documentPath))
    {
        AnsiConsole.MarkupLine($"[bold red] File does not exist.
            [/]\n");
        return;
    }

    var data = File.ReadAllBytes(documentPath);
    var signature = SignData(data, privateKey);
    var signaturePath = documentPath + ".sig";
    File.WriteAllBytes(signaturePath, signature);

    AnsiConsole.MarkupLine($"[bold] 2. Document signed. Signature
```

```
            saved to {signaturePath} [/]\n");
    }

    public static void VerifyDocumentSignature(string documentPath,
        string signaturePath, string publicKeyPath)
    {

        if (!File.Exists(publicKeyPath))
        {
            Console.WriteLine("Public key file does not exist.");
            return;
        }

        var publicKeyXml = File.ReadAllText(publicKeyPath);
        rsa.FromXmlString(publicKeyXml);

        //var signaturePath = documentPath + ".sig";
        if (!File.Exists(signaturePath))
        {
            Console.WriteLine("Signature file does not exist.");
            return;
        }

        var data = File.ReadAllBytes(documentPath);
        var signature = File.ReadAllBytes(signaturePath);
        var isVerified = VerifyData(data, signature);

        AnsiConsole.MarkupLine($"[bold green] Signature Verified:
            {isVerified} [/]\n");
    }

    public static byte[] SignData(byte[] data, RSAParameters
        privateKey)
    {
        rsa.ImportParameters(privateKey);
        var signedData = rsa.SignData(data, HashAlgorithmName.SHA256,
            RSASignaturePadding.Pkcs1);
        return signedData;
    }

    static bool VerifyData(byte[] data, byte[] signature)
    {
        return rsa.VerifyData(data, signature,
            HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    }
}
```

6 Conclusions

After all the theory and practical implementation, i can do several conclusions about the project:

1. **Relevance:** In our rapidly evolving digital landscape, demand for comprehensive security solutions has never been greater. The "Security Toolbox" aims to be relevant tool for both professionals and tech-savvy individuals.
2. **Ease of Use:** With its Command-Line Interface (CLI), the toolbox makes complex cryptographic functions accessible even to those without advanced technical knowledge.
3. **Comprehensive Suite:** The toolbox has a wide range of features, from encryption and decryption to steganography.
4. **Scalability:** The modular design of my project facilitates the integration of future enhancements. This makes it not only a solution for present challenges but also adaptable to forthcoming security requirements.
5. **Educational Value:** Beyond its practical application, the project was an excellent educational purpose by providing hands-on experience with cryptographic functions in the C# language. This aids in the understanding and real-world implementation of theoretical concepts.

7 Not implemented features

I haven't done any kind of error and exception handling neither secure way for parsing private and public key, not certificate support etc.

8 References

ChatGPT, StackOverflow (image steganography mostly, it was rly hard for the first time to get into it), Google