UNIVERSIDADE
DE TRÁS-OS-MONTES
E ALTO DOURO

Criptografia e Segurança de Sistemas Informáticos

Mestrado em Engenharia Informática - 1.º ano

João Paulo Sousa
Pedro Miguel Mestre Alves da Silva

Trabalho Prático 1

Desenvolvimento de Ferramenta Criptográfica

Diogo Medeiros (70633) Tomás Silva (70680)

abril 2023

Resumo

Este trabalho descreve o desenvolvimento de uma ferramenta criptográfica em C# segundo a arquitetura MVVM, utilizando a plataforma .NET e a API System.Security.Cryptography. Este projeto foi desenvolvido no âmbito da Unidade Curricular de Criptografia e Segurança de Sistemas Informáticos.

A ferramenta inclui diversas funcionalidades, tais como compressão e descompressão de ficheiros, hashing de mensagens e checksum de ficheiros, cifragem e decifragem de ficheiros com cifras simétricas e assimétricas, assinaturas digitais, e chat seguro com recurso a WebSockets.

Foi desenvolvido um servidor Web em ASP.NET para suportar a integração com o chat seguro, a integridade e assinatura digital de ficheiros. O chat seguro utiliza o algoritmo RSA para a troca segura de mensagens, em detrimento do EC Diffie-Hellman, devido ao menor impacto no desempenho e na transmissão de dados.

O trabalho atingiu os objetivos propostos de confidencialidade, integridade e disponibilidade de sistemas de informação.

Palavras-chave – criptografia, segurança de sistemas informáticos, compressão de arquivos, síntese de mensagens, checksum de ficheiros, cifra simétrica, cifra assimétrica, assinatura digital, troca segura de mensagens, WebSockets, .NET

Conteúdo

Lista de Figuras

Lista de Excertos de Código

Capítulo 1

Introdução

No âmbito da Unidade Curricular de Criptografia e Segurança de Sistemas Informáticos, foi solicitado o desenvolvimento de uma ferramenta criptográfica, que incorporasse alguns dos conceitos mais comuns e relevantes na área da Criptografia - desde síntese e autenticação de mensagens (*hashing* e MAC), a algoritmos de cifra simétrica e assimétrica (AES, RSA, etc.), assinaturas digitais, e até mesmo comunicações seguras.

Para o efeito, foi deixado ao critério do aluno a escolha da plataforma desejada, nomeadamente, Java ou .NET, devendo o mesmo cingir-se às APIs de segurança disponibilizadas pela plataforma, sem recorrer a bibliotecas de terceiros ou ferramentas externas.

Com estas considerações em mente, foi desenvolvida uma aplicação desktop multifacetada, com recurso à *framework* .NET e, em particular, ao modelo de criptografia disponibilizado pela API System.Security.Cryptography, que permite ao seu utilizador uma variedade de operações, entre as quais:

- A compressão e descompressão de arquivos;
- A síntese (*hashing*) de mensagens;
- A soma de verificação (*checksum*) de ficheiros;
- A cifragem e decifragem de ficheiros;
- A criação e validação de assinaturas digitais;
- A troca segura de mensagens numa sala de chat.

Capítulo 2

Enquadramento tecnológico

O Manual de Segurança Informática do NIST (National Institute of Standards and Technology) [4] define o termo segurança informática da seguinte forma:

Segurança Informática: proteção fornecida a um sistema de informação automatizado com o objetivo de alcançar os objetivos aplicáveis de preservar a integridade, disponibilidade e confidencialidade dos recursos do SI (incluindo hardware, software, firmware, informação/dados e telecomunicações).

Com base nesta definição, o padrão FIPS 199 [7] lista a confidencialidade, integridade e disponibilidade (também conhecida por Tríade da CIA) como os três objetivos de segurança para SI, fornecendo a seguinte caracterização:

- Integridade: Proteger contra modificações ou destruição inadequadas de informação, incluindo a garantia de não repudiação e autenticidade da informação.
- Confidencialidade: Preservar as restrições autorizadas no acesso e divulgação de informação, incluindo meios para proteger a privacidade pessoal e informação proprietária.
- Disponibilidade: Garantir o acesso e uso oportunos e confiáveis de informação.

2.1 Integridade

As funções de síntese (ou hash) são uma das ferramentas criptográficas mais importantes e amplamente utilizadas na segurança de dados. Elas são capazes de gerar uma impressão digital única para qualquer conjunto de dados, independentemente do tamanho ou tipo de informação, e são projetadas para serem unidirecionais, o que significa que é virtualmente impossível reverter o processo de hash de forma a recuperar os dados originais. [9]

As funções hash são usadas em muitas aplicações, incluindo autenticação, integridade de dados, armazenamento de senhas, assinaturas digitais e sistemas de deteção de intrusões. O SHA-256 [6] é uma das funções hash mais comuns e amplamente utilizadas.

2.2 Confidencialidade

As cifras são algoritmos matemáticos que são usados para cifrar e decifrar dados para garantir a segurança e privacidade das informações transmitidas. Elas funcionam através de um processo de transformação dos dados originais em uma forma que seja ininteligível para terceiros, que é conhecida como cifragem. A decifragem é o processo inverso, que transforma os dados cifrados na sua forma original. [1]

Existem dois tipos principais de cifras: simétricas e assimétricas, com a principal diferença sendo o número de chaves usadas. Na cifra simétrica, a mesma chave é usada para cifrar e decifrar os dados, enquanto na cifra assimétrica, são usadas duas chaves diferentes: uma chave pública para cifrar os dados e uma chave privada para decifrar os dados. A escolha de qual cifra usar dependerá do uso específico e dos requisitos de segurança do sistema. [5]

As cifras simétricas são geralmente mais rápidas e eficientes do que as assimétricas, pois usam apenas uma chave, o que as torna mais adequadas para cifrar grandes quantidades de dados. Elas são amplamente utilizadas em aplicações de criptografia de dados em que a velocidade é um fator importante, como proteção de arquivos em um sistema de arquivos ou criptografia de discos rígidos. Alguns exemplos incluem o Advanced Encryption Standard (AES) [3], Data Encryption Standard (DES) e Blowfish.

As cifras assimétricas, por outro lado, são mais seguras do que as simétricas, pois usam duas chaves diferentes para cifrar e decifrar os dados, tornando-as mais adequadas para transações seguras pela Internet, onde a segurança é uma prioridade. As cifras assimétricas são frequentemente usadas em aplicações que envolvem autenticação de utilizadores e assinaturas digitais. Alguns exemplos de cifras assimétricas incluem o Rivest-Shamir-Adleman (RSA) [8], o Elliptic Curve Cryptography (ECC) e o Diffie-Hellman [2].

2.3 Disponibilidade

As assinaturas digitais são uma técnica criptográfica usada para verificar a autenticidade de uma mensagem ou documento digital, bem como a identidade do remetente. Para tal, são utilizados algoritmos criptográficos assimétricos, como o RSA e o DSA, que geram uma assinatura digital única que pode ser verificada usando a chave pública correspondente. A assinatura digital garante que a mensagem não foi alterada e que o remetente é quem afirma ser. [5]

As assinaturas digitais são semelhantes aos códigos de autenticação de mensagem (MAC) e baseados em hash (HMAC), com a principal diferença sendo que as assinaturas digitais usam chaves assimétricas, já os últimos usam uma chave partilhada. As assinaturas digitais são amplamente usadas em várias aplicações, tais como documentos eletrónicos, transações financeiras e contratos legais, onde a autenticidade e integridade da informação são cruciais. [9]

Capítulo 3

Conceção

A ferramenta criptográfica descrita nos seguintes capítulos foi desenvolvida em C#, com recurso à plataforma .NET e, em particular, ao namespace System.Security.Cryptography, e suporta uma variedade de funcionalidades que englobam os três objetivos base da segurança de um SI: confidencialidade, integridade, e disponibilidade.

Assim, foi criada uma solução .NET, composta por três projetos com característica e propósitos distintos:

- CryptoTools: a aplicação desktop WPF, segundo a arquitetura MVVM (Model-View-ViewModel), que disponibiliza uma interface gráfica para todas as funcionalidades da ferramenta criptográfica;

- CryptoServer: o servidor Web ASP.NET, que disponibiliza serviços ao cliente, desde o registo e login na plataforma, integridade e assinatura digital de ficheiros, e chat encriptado com recurso a WebSockets

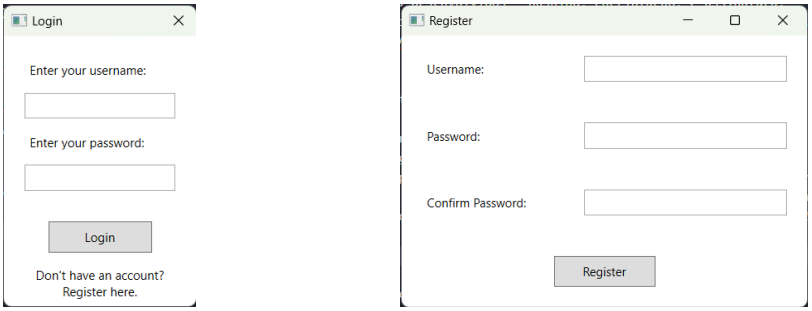
- CryptoLib: a biblioteca de classes, que inclui extensões de classes nativas como SecureString e ZipArchive, e modelos de dados para ORM (Object-Relational Mapping) da base de dados SQL Server

Capítulo 4

Implementação

Tal como foi referido no capítulo 3, o utilizador dispõe de uma interface gráfica para poder explorar e experimentar alguns dos conceitos mais importantes na área da criptografia.

Para isto, é necessário iniciar sessão no servidor remoto (Fig. 4.1a) ou, caso ainda não se encontre registado, efetuar o registo na plataforma (Fig. 4.1b).



(a) Login (b) Registo

Figura 4.1: Janelas de login e registo

No caso do registo, o servidor recebe como parâmetros o nome de utilizador e a palavra-passe, gera um *salt* aleatório com a classe RandomNumberGenerator (Alg. 1), computa a hash da palavra-passe com SHA-256 (Alg. 2), e, por fim, adiciona o novo registo à base dados.

```
private static string GenerateSalt(int length = 16)
{
    var salt = RandomNumberGenerator.GetBytes(length);
    return Convert.ToBase64String(salt);
}
```

Excerto de código 1: Geração do *salt* aleatório

```
public static string GenerateHash(string password, string salt)
{
    var saltBytes = Convert.FromBase64String(salt);
    var passwordBytes = Encoding.UTF8.GetBytes(password);
    var combinedBytes = new byte[saltBytes.Length
                                 + passwordBytes.Length];
    Buffer.BlockCopy(saltBytes, 0, combinedBytes, 0,
        saltBytes.Length);
    Buffer.BlockCopy(passwordBytes, 0, combinedBytes,
        saltBytes.Length, passwordBytes.Length);
    var hashBytes = SHA256.HashData(combinedBytes);
    return Convert.ToBase64String(hashBytes);
}
```

Excerto de código 2: Cálculo do hash da palavra-passe

Já no caso do login, o servidor valida os dados do utilizador e gera um JSON Web Token (JWT), assinado com uma chave privada de 32 bytes usando o algoritmo HMAC-SHA256 (Alg. 3), devolvendo esse mesmo token para futuros pedidos.

```
public static string GenerateAccessToken(string userName)
{
    // Set up the JWT security settings
    var tokenHandler = new JwtSecurityTokenHandler();
    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new Claim[]
        {
            new(ClaimTypes.Name, userName),
            new(ClaimTypes.Role, "User")
        }),
        Expires = DateTime.UtcNow.AddHours(1),
        SigningCredentials = new SigningCredentials(
            new SymmetricSecurityKey(Key),
            SecurityAlgorithms.HmacSha256Signature
        )
    };
    var token = tokenHandler.CreateToken(tokenDescriptor);
    var accessToken = tokenHandler.WriteToken(token);
    return accessToken;
}
```

Excerto de código 3: Geração do token de acesso

Após iniciar sessão e armazenar o token de acesso enviado pelo servidor, o cliente pode finalmente aceder à ferramenta criptográfica CryptoTools, cujas funcionalidades serão apresentadas e descritas em maior detalhe de seguida.

4.1 Compressão de ficheiros

O primeiro separador da aplicação, visível na Fig. 4.2, concerne a operação mais vulgar: a compressão e descompressão de arquivos ZIP, com o recurso ao namespace System.IO.Compression.

Neste separador, o utilizador dispõe de várias opções, desde adicionar e remover diretórios e ficheiros à lista de entradas a comprimir, criar o arquivo ZIP de acordo com o nível de compressão, e descomprimir arquivos.

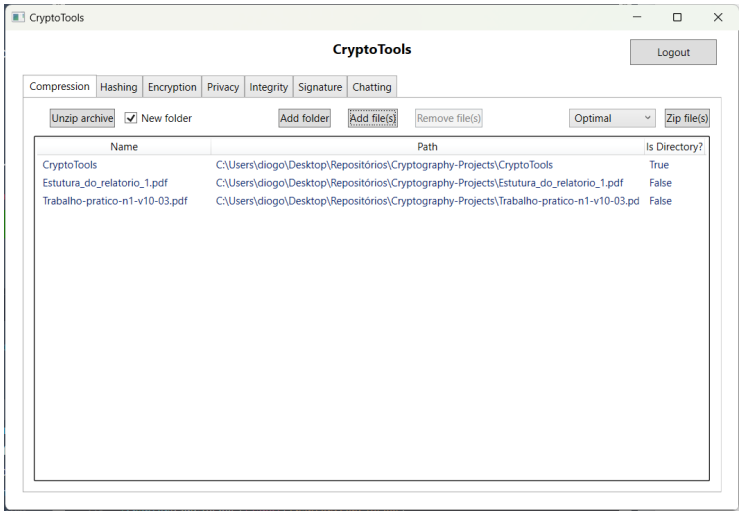


Figura 4.2: Separador de compressão e descompressão de arquivos

Começando pela compressão, como se pode ver em Alg. 4, o método cria o arquivo ZIP e itera os elementos adicionados, criando entradas para cada um, com a ajuda da extensão definida em CryptoLib.Extensions.

```
public void CompressArchive(string path)
{
    using var fileStream = File.Create(path);
    using var archive = new ZipArchive(fileStream,
        ZipArchiveMode.Create);
    foreach (var entry in ArchiveEntries)
        if (entry.IsDirectory)
            archive.CreateEntryFromDirectory(entry.Path,
            entry.Name, CompressionLevel);
        else
            archive.CreateEntryFromFile(entry.Path, entry.Name,
                CompressionLevel);
}
```

Excerto de código 4: Compressão de ficheiros

Já a descompressão, como seria de esperar, é mais simples, consistindo

apenas em abrir o arquivo ZIP e extrair todos os ficheiros para o diretório desejado, como se pode ver em Alg. 5.

```
public void DecompressArchive(string path)
{
    using var fileStream = File.OpenRead(path);
    using var archive = new ZipArchive(fileStream,
        ZipArchiveMode.Read);
    var directory = Path.GetDirectoryName(path);
    if (CreateNewFolder)
    {
        var name = Path.GetFileNameWithoutExtension(path);
        directory = Path.Combine(directory, name);
    }
    archive.ExtractToDirectory(directory);
}
```

Excerto de código 5: Extração de ficheiros

4.2 Síntese de informação

Tal como foi referido no capítulo 2, as funções de síntese permitem gerar uma hash única a partir de um conjunto de dados, quer se trate de uma simples mensagem de texto, ou de um ficheiro.

O segundo separador da aplicação, visível na Fig. 4.3, serve este propósito, permitindo escolher entre várias funções hash, como MD5 e SHA256.

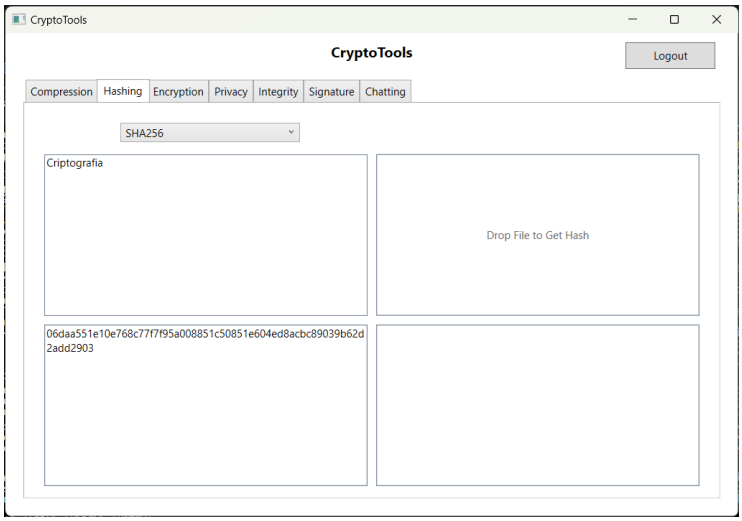


Figura 4.3: Separador de síntese de mensagens e checksum de ficheiros

No caso do utilizador escrever uma mensagem, é apresentado abaixo, em formato hexadecimal, o *digest* da mesma, calculado a partir da função hash escolhida (Alg.6).

```
public static byte[] Hash(string text, string algorithm)
{
    using var hashAlgorithm = GetHashAlgorithm(algorithm);
    var textBytes = Encoding.UTF8.GetBytes(text);
    return hashAlgorithm.ComputeHash(textBytes);
}
```

Excerto de código 6: Síntese de mensagem

Caso opte por fornecer um ficheiro, é apresentado o checksum do mesmo, obtido a partir da função em Alg.7.

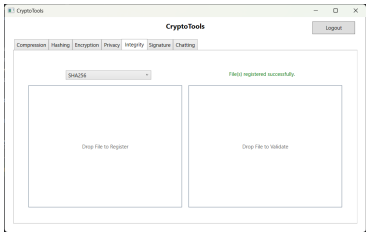
```
public static byte[] HashFile(string file, string algorithm)
{
    using var hashAlgorithm = GetHashAlgorithm(algorithm);
    using var stream = File.OpenRead(file);
    return hashAlgorithm.ComputeHash(stream);
}

public static string ToHexString(byte[] bytes)
{
    return Convert.ToHexString(bytes).ToLower();
}
```

Excerto de código 7: Checksum de ficheiro

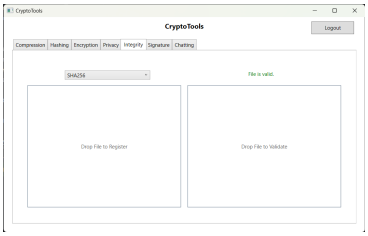
4.3 Integridade de dados

O separador "Integrity" disponibiliza ao utilizador uma forma de validar a integridade de um ficheiro local, permitindo registar um ou mais ficheiros na base de dados remota (Fig. 4.4a), bem como a possibilidade de atualizar um registo.

Este processo rápido e seguro permite ao cliente determinar se um ficheiro foi ou não alterado desde o último registo (Fig. 4.4b).



(a) Registo de ficheiro(s)



(b) Validação de ficheiro

Figura 4.4: Separador de registo e validação dos ficheiros

4.4 Encriptação de imagens

O separador "Encryption" serve um propósito meramente ilustrativo, permitindo ao utilizador cifrar uma imagem PNG, JPEG, ou BMP à sua escolha, e visualizar o conteúdo cifrado da mesma, como é possível observar na Fig. 4.5.

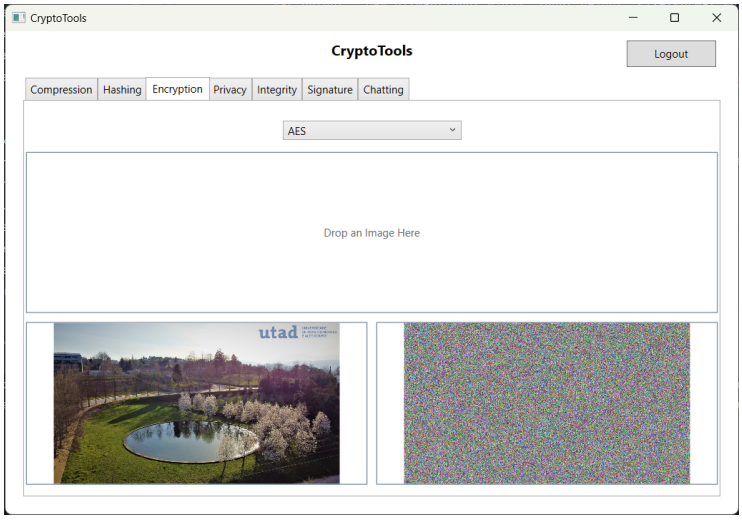


Figura 4.5: Separador de cifragem de imagens

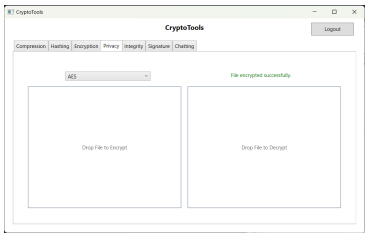
Em Alg. 8, encontra-se resumido o processo executado, começando pela conversão da imagem em escala de cinzentos, para diminuir a alocação de memória, e posterior cifragem da mesma.

```
public Bitmap EncryptImage(string imagePath)
{
    using var aes = Aes.Create();
    aes.GenerateKey();
    aes.GenerateIV();
    var grayscale = BitmapUtils.ToGrayscale(imagePath);
    var pixelData = new byte[grayscale.PixelWidth *
↪   grayscale.PixelHeight];
    grayscale.CopyPixels(pixelData, grayscale.PixelWidth, 0);
    using var encryptor = aes.CreateEncryptor();
    var encryptedPixelData = encryptor
        .TransformFinalBlock(pixelData, 0, pixelData.Length);
    var encryptedBitmap = new Bitmap(grayscale.PixelWidth,
↪   grayscale.PixelHeight, PixelFormat.Format8bppIndexed);
    // Copy the encrypted data to the new bitmap
    return encryptedBitmap;
}
```

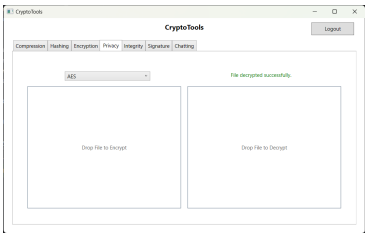
Excerto de código 8: Cifragem de imagem

4.5 Cifragem de ficheiros

O separador "Privacy" fornece ao utilizador um mecanismo simples e seguro para poder cifrar e decifrar ficheiros locais (Fig. 4.6), assegurando assim a sua confidencialidade.



(a) Cifragem de ficheiro



(b) Validação de ficheiro

Figura 4.6: Separador de cifragem e decifragem de ficheiros

Quando o utilizador opta por cifrar um determinado ficheiro (Fig. 4.6a), é chamado o seguinte método, descrito em Alg. 9. Este recebe como parâmetros o caminho do ficheiro, a cifra simétrica escolhida e os parâmetros RSA para poder cifrar a chave simétrica.

```
public void EncryptFile(string fileName, string algorithmName,
↪  RSAParameters parameters)
{
    using rsa = RSA.Create(parameters);
    using var symAlg = GetAlgorithm(algorithmName);
    symAlg.GenerateKey();
    symAlg.GenerateIV();
    var encryptedKey = rsa.Encrypt(symAlg.Key,
↪  RSAEncryptionPadding.Pkcs1);
    var outFile = Path.ChangeExtension(fileName, ".enc");
    using var outFs = new FileStream(outFile, FileMode.Create);
    // Write encrypted key, IV and extension
    using var cs = new CryptoStream(outFs,
↪  symAlg.CreateEncryptor(), CryptoStreamMode.Write);
    var blockSizeBytes = symAlg.BlockSize / 8;
    var data = new byte[blockSizeBytes];
    using var inFs = new FileStream(fileName, FileMode.Open);
    int count;
    do {
        count = inFs.Read(data, 0, blockSizeBytes);
        cs.Write(data, 0, count);
    } while (count > 0);
    cs.FlushFinalBlock();
}
```

Excerto de código 9: Cifragem de ficheiro

Inicialmente, é criada uma nova instância do algoritmo RSA e da cifra

escolhida, e gerados a chave e vetor de inicialização (IV). A chave simétrica é cifrada com a instância RSA, e é criado um novo ficheiro ".enc" onde são guardados a chave cifrada e o IV, bem como a extensão original. Por fim, é criada uma *stream* criptográfica, onde é processada a cifragem do ficheiro original, bloco a bloco, com a cifra simétrica.

Já no caso da decifragem (Fig. 4.6b), um método semelhante é chamado, responsável por ler e processar o ficheiro decifrado (Alg. 10).

```
public static void DecryptFile(string fileName, string
↪  algorithmName, RSAParameters parameters)
{
    using var rsa = RSA.Create(parameters);
    using var symAlg = GetAlgorithm(algorithmName);
    using var inFs = new FileStream(fileName, FileMode.Open);
    // Read encrypted key, IV and extension
    // Assuming 'encryptedKey', 'iv' and 'extension' exist
    var key = rsa.Decrypt(encryptedKey,
↪  RSAEncryptionPadding.Pkcs1);
    var outFile = Path.ChangeExtension(fileName, extension);
    using var outFs = new FileStream(outFile, FileMode.Create);
    using var cs = new CryptoStream(outFs,
↪  symAlg.CreateDecryptor(key, iv), CryptoStreamMode.Write);
    var blockSizeBytes = symAlg.BlockSize / 8;
    var data = new byte[blockSizeBytes];
    int count;
    do
    {
        count = inFs.Read(data, 0, blockSizeBytes);
        cs.Write(data, 0, count);
    } while (count > 0);
    cs.FlushFinalBlock();
}
```

Excerto de código 10: Decifragem de ficheiro

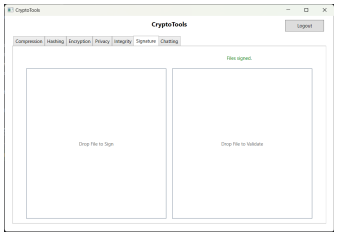
Este método, tal como o Alg. 9, começa por criar as instâncias do RSA e da cifra simétrica com base nos parâmetros de entrada, abrir o ficheiro decifrado e ler os dados escritos no cabeçalho: a chave simétrica cifrada, o vetor de inicialização, e a extensão original.

De seguida, usando a instância RSA, decifra a chave simétrica usada para cifrar o conteúdo do ficheiro, e cria o novo ficheiro decifrado, com a mesma extensão. Por fim, cria uma *stream* criptográfica, fornecendo a cifra simétrica com a chave decifrada e o IV, e decifra o ficheiro, novamente, bloco a bloco.

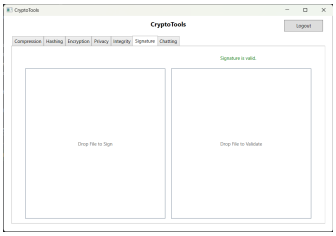
Para permitir que o utilizador cifre e decifre ficheiros em sessões diferentes, os parâmetros da instância RSA são guardados num ficheiro binário "rsa.bin", na pasta "%APPDATA%\CryptoTools".

4.6 Assinatura digital

O separador "Signature", tal como o nome indica, permite ao utilizador assinar digitalmente um ficheiro à sua escolha, tal como mostra a Fig. 4.7.



(a) Assinatura de ficheiro



(b) Verificação de assinatura

Figura 4.7: Separador de criação e validação de assinaturas digitais

Para garantir a integridade da informação, o cliente computa a hash SHA-256 do ficheiro escolhido (Alg. 11), e envia esta ao servidor, que a assina com a sua chave privada RSA (Alg. 12). Após assinar, o servidor devolve a assinatura ao cliente, que a guarda localmente num ficheiro ".sign", e informa o utilizador com a mensagem presente em Fig. 4.7a.

```
public async Task SignFile(string fileName)
{
    await using var fs = new FileStream(fileName, FileMode.Open);
    var hash = await _sha256.ComputeHashAsync(inputStream);
    using var cl = new HttpClient();
    var res = await cl.PostAsJsonAsync($"{Model.ServerUrl}/sign",
↪   new SignatureRequest {Data = Convert.ToBase64String(hash)});
    var signature = await res.Content.ReadAsStringAsync();
    var signatureBytes = Convert.FromBase64String(signature);
    await using var outFs = new FileStream(fileName + ".sig",
        FileMode.Create);
    await outFs.WriteAsync(signatureBytes);
}
```

Excerto de código 11: Assinatura digital de ficheiro (Cliente)

```
public IActionResult Sign([FromBody] SignatureRequest request)
{
    var data = Convert.FromBase64String(request.Data);
    var signature = _rsa.SignData(data, HashAlgorithmName.SHA256,
↪   RSASignaturePadding.Pkcs1);
    return Ok(Convert.ToBase64String(signature));
}
```

Excerto de código 12: Assinatura digital de ficheiro (Servidor)

Para verificar a assinatura digital do ficheiro, o cliente envia ao servidor a hash SHA-256 do ficheiro e a assinatura digital armazenada no ficheiro

".sign"(Alg. 13). O servidor verifica a assinatura digital usando a sua chave pública RSA e a hash SHA-256 do ficheiro recebida do cliente.

Se a verificação for bem sucedida, o servidor retorna uma resposta HTTP `Ok` ao cliente, indicando que a assinatura digital é válida (Fig.4.7b). Caso contrário, o servidor retorna uma resposta HTTP `BadRequest` com a mensagem "Signature is invalid." (Alg. 14).

```
public async Task VerifySignature(string fileName)
{
    await using var fs = new FileStream(fileName, FileMode.Open);
    var hash = await _sha256.ComputeHashAsync(fs);
    var signature = await File.ReadAllBytesAsync(fileName +
↪   ".sign");
    using var client = new HttpClient();
    var response = await
↪   client.PostAsJsonAsync($"{Model.ServerUrl}/verify",
        new VerifyRequest
        {
            Hash = Convert.ToBase64String(hash),
            Signature = Convert.ToBase64String(signature)
        });
    if (!response.IsSuccessStatusCode)
    {
        var message = await response.Content.ReadAsStringAsync();
        DisplayMessage?.Invoke(message, Colors.Coral);
        return;
    }

    DisplayMessage?.Invoke("Signature is valid.", Colors.Green);
}
```

Excerto de código 13: Verificação de assinatura digital (Cliente)

```
public IActionResult Verify([FromBody] VerifyRequest request)
{
    var hash = Convert.FromBase64String(request.Hash);
    var signature = Convert.FromBase64String(request.Signature);
    var verified = _rsa.VerifyHash(hash, signature,
↪   HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    if (verified) return Ok();
    return BadRequest("Signature is invalid.");
}
```

Excerto de código 14: Verificação de assinatura digital (Servidor)

De forma a simplificar o processo de assinatura no servidor, os parâmetros RSA são armazenados localmente num ficheiro XML. No caso de uma falha do sistema, é possível recuperar as chaves privada e pública a partir deste arquivo e, assim, restaurar o funcionamento normal do sistema de assinatura digital.

4.7 Conversação segura

O último separador, "Chatting", consiste numa "chatroom", ou sala de conversação, que assegura a confidencialidade e integridade das mensagens enviadas entre vários utilizadores, como é possível ver na Fig. 4.8.

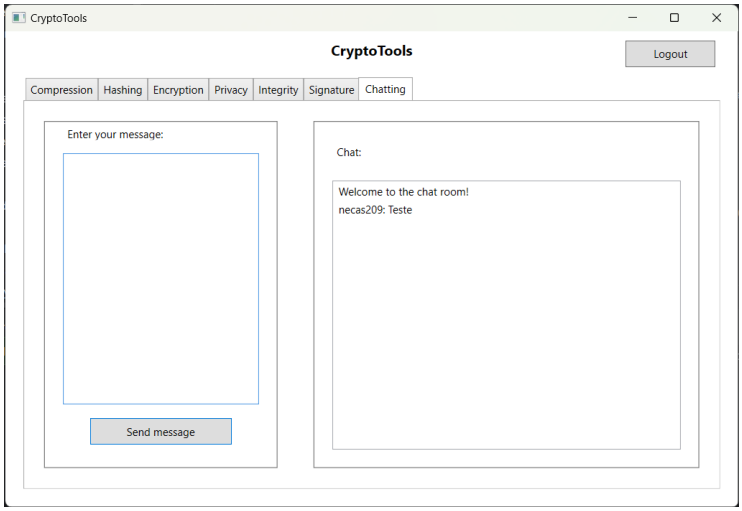


Figura 4.8: Separador de chat

Após efetuar login, o cliente envia um pedido HTTP ao servidor contendo a sua chave pública, a requisitar a chave pública RSA do servidor (Alg 15). O servidor, ao receber o pedido, armazena a chave do cliente associada ao *username*, e responde com a sua chave RSA, que é importada pelo cliente (Alg 16).

```
public async Task OpenConnection()
{
    using var client = new HttpClient();
    var response = await client.PostAsJsonAsync(
        $"{ServerUrl}/exchange",
        new ExchangeRequest
        {
            PublicKey = _clientRsa.ExportRSAPublicKey()
        });
    var content = await response.Content.ReadAsStringAsync();
    var serverKey = Convert.FromBase64String(content);
    _serverRsa.ImportRSAPublicKey(serverKey, out _);
    _socket.Options.SetRequestHeader("X-Access-Token",
↪   AccessToken);
    await _socket.ConnectAsync(ChatUri, CancellationToken.None);
}
```

Excerto de código 15: Abertura de conexão WebSocket

O método SendMessage (Alg. 17) envia uma mensagem encriptada e autenticada por HMAC para o servidor, utilizando criptografia simétrica e as-

```
public IActionResult Exchange([FromBody] ExchangeRequest request)
{
    var token = HttpContext.Request.Headers["X-Access-Token"]
        .ToString();
    var userName = TokenUtils.ValidateAccessToken(token);
    _chatHandler.AddUser(userName, request.PublicKey);
    var serverKey = _chatHandler.GetPublicKey();
    var base64Key = Convert.ToBase64String(serverKey);
    return Ok(base64Key);
}
```

Excerto de código 16: Troca de chaves públicas RSA

simétrica. Primeiro, é gerada uma chave aleatória para o HMAC. Em seguida, uma chave simétrica é gerada e utilizada para cifrar a mensagem. Esta é então cifrada com a chave pública do servidor utilizando RSA e preenchimento OAEP SHA-256. Da mesma forma, a chave HMAC é cifrada com a chave do servidor. Um objeto é criado contendo a mensagem cifrada, o HMAC, as chaves simétrica e HMAC cifradas, serializado e enviado para o servidor através do WebSocket.

```
public async Task SendMessage(string message) {
    var messageBytes = Encoding.UTF8.GetBytes(message);
    var hmacKey = RandomNumberGenerator.GetBytes(32);
    var hmac = HMACSHA256.HashData(hmacKey, messageBytes);
    using var aes = Aes.Create();
    aes.GenerateKey();
    var encMessage = AesUtils.Encrypt(messageBytes, aes.Key);
    var encryptedSymmetricKey = _serverRsa.Encrypt(aes.Key,
↪   RSAEncryptionPadding.OaepSHA256);
    var encryptedHmacKey = _serverRsa.Encrypt(hmacKey,
↪   RSAEncryptionPadding.OaepSHA256);
    var chatMessage = new ChatMessage
    {
        Message = encMessage,
        Hmac = hmac,
        SymmetricKey = encryptedSymmetricKey,
        HmacKey = encryptedHmacKey
    };
    var json = JsonSerializer.Serialize(chatMessage);
    var buffer = Encoding.UTF8.GetBytes(json);
    await _socket.SendAsync(new ArraySegment<byte>(buffer),
↪   WebSocketMessageType.Text, true, CancellationToken.None);
}
```

Excerto de código 17: Envio de mensagens pelo cliente

O método ReceiveMessage (Alg. 18) começa por receber uma mensagem do servidor cifrada e protegida por uma chave simétrica, a qual também se encontra cifrada com a chave pública do cliente. Para decifrar a mensagem, o cliente começa por decifrar a chave simétrica com a sua chave privada RSA.

Em seguida, decifra a mensagem propriamente dita utilizando a chave simétrica obtida anteriormente. Posteriormente, é verificada a integridade da mensagem com recurso a HMAC, o qual utiliza uma chave específica para o efeito, que também é decifrada com a chave privada RSA do cliente.

Por fim, caso o HMAC da mensagem decifrada corresponda ao HMAC enviado pelo servidor, a mensagem é convertida para string e devolvida com a indicação do nome do utilizador que a enviou. Caso contrário, é lançada uma exceção com a mensagem "HMAC verification failed".

```
public async Task<string?> ReceiveMessage()
{
    var buffer = new byte[4096];
    var result = await _socket.ReceiveAsync(buffer,
↪   CancellationToken.None);
    var msg = Encoding.UTF8.GetString(buffer, 0, result.Count);
    var message = JsonSerializer.Deserialize<ChatMessage>(msg);
    var decryptedKey = _clientRsa.Decrypt(message.SymmetricKey,
↪   RSAEncryptionPadding.OaepSHA256);
    var hmacKey = _clientRsa.Decrypt(message.HmacKey,
↪   RSAEncryptionPadding.OaepSHA256);
    var decryptedMessage = AesUtils.Decrypt(message.Message,
↪   decryptedKey);
    var hmac = HMACSHA256.HashData(hmacKey, decryptedMessage);
    if (!hmac.SequenceEqual(message.Hmac)) throw new
↪   Exception("HMAC verification failed");
    var decMsg = Encoding.UTF8.GetString(decryptedMessage);
    return $"{message.UserName}: {decMsg}";
}
```

Excerto de código 18: Receção de mensagens no cliente

O método Handle da classe ChatHandler (Alg. 19) recebe uma conexão WebSocket do cliente, juntamente com o seu nome de utilizador e token de acesso. Em primeiro lugar, é enviado uma mensagem de boas-vindas para o novo utilizador e informados os restantes utilizadores da sua chegada.

Em seguida, o método aguarda a chegada de mensagens do utilizador, decifra-as com a chave privada do servidor e transmite-as em broadcast. Se a conexão do WebSocket for fechada, o utilizador será removido da lista de utilizadores conectados e a sua chave pública será removida do servidor. Durante todo este processo, a validade do token de acesso é verificada.

O método SendMessage (Alg. 20) é responsável por cifrar e enviar uma mensagem para uma conexão específica. Primeiro, ele importa a chave pública do cliente que está associada à conexão, a fim de usar essa chave para cifrar a mensagem. Em seguida, ela cifra a chave simétrica e a chave HMAC da mensagem usando a chave pública do cliente e cria uma nova mensagem com as informações cifradas. Finalmente, a mensagem é serializada em formato JSON e enviada como um array de bytes através do WebSocket.

```
public async Task Handle(WebSocket socket, string token) {
    // Send welcome and joined messages
    var buf = new byte[4096];
    var result = await socket.ReceiveAsync(new
↪   ArraySegment<byte>(buf), CancellationToken.None);
    while (!result.CloseStatus.HasValue)
    {
        // Close connection if token is no longer valid
        var json = Encoding.UTF8.GetString(buf, 0, result.Count);
        var msg = JsonSerializer.Deserialize<ChatMessage>(json);
        msg.SymmetricKey = _serverRsa.Decrypt(msg.SymmetricKey,
↪   RSAEncryptionPadding.OaepSHA256);
        msg.HmacKey = _serverRsa.Decrypt(msg.HmacKey,
↪   RSAEncryptionPadding.OaepSHA256);
        await BroadcastMessage(msg);
        buf = new byte[4096];
        result = await socket.ReceiveAsync(new
↪   ArraySegment<byte>(buf), CancellationToken.None);
    }
    await socket.CloseAsync(result.CloseStatus.Value,
↪   result.CloseStatusDescription, CancellationToken.None);
    BroadcastMessage(sender, $"{sender.UserName} left the chat");
    _clientKeys.Remove(sender.UserName);
}
```

Excerto de código 19: Gestão de mensagens no servidor

```
 private async Task SendMessage(WebSocketConnection connection,
↪   ChatMessage message) {
    var userPublicKey = _clientKeys[connection.UserName];
    _clientRsa.ImportRSAPublicKey(userPublicKey, out _);
    var encSymKey = _clientRsa.Encrypt(message.SymmetricKey,
↪   RSAEncryptionPadding.OaepSHA256);
    var encryptedHmacKey = _clientRsa.Encrypt(message.HmacKey,
↪   RSAEncryptionPadding.OaepSHA256);
    var messageToSend = new ChatMessage
    {
        UserName = message.UserName, Message = message.Message,
        Hmac = message.Hmac, HmacKey = encryptedHmacKey,
        SymmetricKey = encSymKey,
    };
    var json = JsonSerializer.Serialize(messageToSend);
    var buffer = Encoding.UTF8.GetBytes(json);
    await connection.WebSocket.SendAsync(new
↪   ArraySegment<byte>(buffer), WebSocketMessageType.Text, true,
↪   CancellationToken.None);
}
```

Excerto de código 20: Envio de mensagens pelo servidor

Capítulo 5

Conclusões

No que diz respeito aos requisitos preenchidos, a aplicação desenvolvida cumpre com os objetivos propostos no protocolo do Trabalho Prático N.º 1. Foram implementados algoritmos de síntese de mensagens (hashing), checksum de ficheiros, cifra simétrica (AES) e assimétrica (RSA), assinaturas digitais, comunicações seguras e integridade de dados.

Destaca-se a implementação do chat seguro, que utiliza WebSockets para comunicação entre os utilizadores e o servidor. A comunicação é encriptada com um algoritmo de cifra simétrica (AES), utilizando uma chave gerada aleatoriamente para cada mensagem enviada. A chave é cifrada assimetricamente (RSA) com a chave pública do destinatário e enviada juntamente com a mensagem cifrada. Além disso, é utilizada uma soma de verificação (HMAC) para garantir a integridade das mensagens transmitidas.

A escolha do algoritmo RSA para o chat seguro foi baseada em vários fatores. Embora o ECDH permitisse que as mensagens fossem transmitidas cifradas sem o servidor ter acesso a elas, o processo seria pesado e implicaria que cada cliente cifrasse a chave simétrica com a chave pública de cada um dos restantes utilizadores e enviasse todos estes dados. Por outro lado, o uso do RSA para a troca de chaves permitiu que a chave simétrica fosse cifrada apenas uma vez, com a chave pública do servidor, e depois cifrada com a chave pública de cada utilizador. Além disso, o RSA é mais maduro e amplamente utilizado, o que o torna uma escolha mais segura e confiável.

A integração com o servidor também é um ponto importante a destacar, uma vez que foram implementados serviços que permitem o registo e login dos utilizadores, a integridade e assinatura digital de ficheiros, e o chat seguro. Os utilizadores são autenticados com um token de acesso, gerado pelo servidor no momento do login, e as chaves públicas dos utilizadores são armazenadas no servidor para permitir a cifragem assimétrica das mensagens do chat.

Em suma, a ferramenta desenvolvida cumpre com os objetivos propostos e implementa uma variedade de funcionalidades e algoritmos relevantes para a segurança de sistemas informáticos. A integração com o servidor e a implementação do chat seguro são pontos importantes que demonstram a aplicação prática dos conceitos aprendidos na Unidade Curricular.

Bibliografia

[1] Matt Bishop. *Introduction to Computer Security.* Addison-Wesley Professional, 2004.

[2] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[3] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced Encryption Standard (AES), 2001-11-26 2001.

[4] Barbara Guttman and E Roback. An Introduction to Computer Security: the NIST Handbook, 1995-10-02 1995.

[5] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2014.

[6] U.S. Department of Commerce, National Institute of Standards, and Technology. *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4.* CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2012.

[7] Shirley Radack. Federal Information Processing Standard (FIPS) 199, Standards for Security, 2004-03-01 2004.

[8] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[9] William Stallings. *Cryptography and Network Security: Principles and Practice.* Prentice Hall Press, USA, 6th edition, 2013.