# Segurança computacional - Implementação de um assinador

João Tito do Nascimento Silva (18/0123301)

Setembro de 2022

## 1 Objetivo

Este trabalho tem o objetivo de estudar a construção e implementação de assinaturas digitais. Também almejamos estudar mais afundo a estrutura e a implementação dos algoritmos AES e RSA, que são amplamente utilizados em diversas aplicações atualmente.

O presente documento visa detalhar o processo de implementação realizado, as dificuldades que surgiram e a estrutura do código produzido, que consiste de um simples assinador e verificador de arquivos.

## 2 Funcionalidades da aplicação e código

### 2.1 Tecnologias utilizadas e estrutura do código

A implementação dos algoritmos AES e RSA foi feita na linguagem C# para o framework *.NET 5.0*. Para demonstrar o funcionamento das implementações dos algoritmos criptográficos, foi feito um projeto de API em C# e uma aplicação simples em Angular (Typescript) integradas como um assinador de arquivos.

O código é dividido em 4 diretórios:

- ClientApp: aplicação em Angular (Typescript) para demonstração das funcionalidades.
- BobAndAlice.App: código de API em C# que é integrada com o ClientApp para a demonstração
- BobAndAlice.Core: implementação das funções matemáticas e criptográficas utilizadas, além da implementação do esquema de assinatura
- BobAndAlice.Core.Tests: projeto de testes automatizados para as funções matemáticas e criptográficas no Core

### 2.2 Aplicação para demonstração

Para demonstração, foi implementada uma aplicação simples que assina e verifica assinaturas em arquivos, além de gerar chaves. O usuário se registra e realizar login com senha. A partir do momento em que está autenticado, poderá realizar as ações descritas abaixo.

### 2.2.1 Geração de chaves de assinatura (RSA)

A primeira funcionalidade que pode ser utilizada é a geração de chaves RSA. O usuário pode escolher e fornecer dois números primos no formato base64 para serem utilizados na geração de chaves ou solicitar ao software que gere os dois números primos, o que será feito por meio do código implementado em C#. Após a submissão dos dois números primos, é realizado o processo de derivação de chaves RSA a partir desses números. As chaves geradas são exibidas na tela de chaves.

### 2.2.2 Assinatura

Depois de gerar suas chaves, o usuário pode utilizar a funcionalidade de assinar arquivos. Ao clicar no botão, o usuário será fornecido uma tela para selecionar que arquivo deseja assinar. O arquivo é, então, fornecido por API ao código C#, que lê os conteúdos do arquivo e assina, retornando os dados da assinatura no formato base64. O usuário pode, então, baixar o arquivo original a partir da assinatura, para testar a decriptação simétrica; e também pode baixar o arquivo assinado no formato JSON (escolheu-se o formato JSON para facilitar a demonstração dos campos gerados pela assinatura).

### 2.2.3 Verificação da assinatura

Para a verificação da assinatura, o usuário seleciona um arquivo de assinatura JSON gerado pelo assinador, que é fornecido para o código C# via API. A assinatura é então lida e verificada utilizando os algoritmos RSA. O usuário pode, também, baixar o arquivo original decriptado.

# 3 Algoritmos criptográficos

## 3.1 RSA-OAEP

O algoritmo de criptografia assimétrica implementado foi o RSA-OAEP. A geração de chaves e a função de trapdoor RSA dependem da realização de exponenciações modulares com números muito grandes. Para isso, foi utilizada a função *BigInteger.ModPow* built-in no C#.

### 3.1.1 Geração de números primos

Para a geração de chaves RSA é necessário gerar dois números primos grandes e depois realizar um processo de derivação de cahves a partir desses números. Para isso, foi utilizado um gerados de números pseudo-aleatórios built-in nos pacotes padrão do .NET 5.0 em combinação com uma implementação do teste de Miller-Rabin.

Primeiramente, é gerado um número de 1024 bits aleatório utilizando o gerador. Esse número é testado para divisibilidade com alguns números primos pequenos (de 2 a 29), para eliminar alguns números com maior velocidade e evitar passar números demais para o teste de Miller-Rabin, melhorando a performance do código. Caso o número testado não seja divisível, o processo é realizado novamente.

Quando um número não divisível por esses primos pequenos é finalmente encontrado, é então testado com o teste de Miller-Rabin.

O teste usa, por padrão, 20 rounds com bases aleatórias. As bases são limitadas até 1000000, para evitar testes com números grandes demais, o que é desnecessário. Com 20 rounds, a probabilidade de um falso positivo acontecer (o teste marcar um número como primo e este, na verdade, não ser) é de $2^{-40}$, que é um número muito pequeno. A quantidade de rounds utilizada pode ainda ser parametrizada.

Alguns testes automatizados foram implementados, e um dos testes particularmente interessante é testar os números primos de Mersenne com o teste de Miller-Rabin implementado.

### 3.1.2 Derivação de chaves RSA

Para a geração de chaves RSA, são inicialmente gerados números primos de 1024 bits (128 bytes). Depois, é realizado o processo normal de derivação de chaves RSA descrito por Katz e Lindell [1].

### 3.1.3 Cifração e decifração

Com a geração de chaves finalizada, foi implementada a cifração e decifração com RSA-OAEP. Primeramente, foi implementada a função de trapdoor RSA e seus testes automatizados. Quando esta função estava funcionando corretamente, foi combinada com duas funções que adicionam ou retiram o padding e os campos gerados pelo padrão OAEP para a construção das funções de cifração e decifração.

Como função de geração de máscaras para o OAEP foi utilizado o algoritmo SHA-3. O usuário define no construtor da classe qual tamanhos de mensagem e de número aleatório deem ser utilizados, e esses valores devem ser tamanhos suportados pela implementação do SHA-3 utilizada, da biblioteca *BouncyCastle*.

## 3.2 Implementação do AES

A implementação realizada para o algoritmo AES buscou seguir as especificações na estrutura do código, mantendo os nomes padronizados para os métodos e constantes. Os tamanhos de chave padrão (128, 192 e 256) são todos suportados e devem ser especificados no construtor da classe AES.

### 3.2.1 Geração da chave

A geração da chave de encriptação simétrica é realizada por meio de um simples gerador de números pseudo-aleatórios.

### 3.2.2 Expansão de chave

O algoritmo da expansão de chave é a primeira etapa utilizada no AES, e foi implementado. A expansão consulta apenas uma tabela de *round constants* e realiza outras operações em métodos implementados na classe AES, de acordo com o parâmetro do tamanho de chave.

### 3.2.3 Implementação dos métodos de encriptação e decriptação

Os métodos de mais alto nível para encriptação e decriptação recebem mensagens no formato Binary que são divididas em blocos de 16 bytes (128 bits). Em seguida, cada bloco é cifrado independentemente e os resultados são reunidos ao final.

Para a cifração de cada bloco, é utilizada a estrutura normal de rounds do AES, sendo que cada round possui um método e sua inversa correspondente. Cada round possui uma estrutura específica de acordo com a documentação do AES, e cada função utilizada pelos rounds também é implementada individualmente.

As funções de *S-Box* consultam uma tabela constante para facilitar a implementação e entendimento, além de melhorar a performance.

# 4 Esquema de assinatura

O esquema de assinatura utilizado consiste das seguintes etapas:

1. Geração de uma chave aleatória para AES com o tamanho especificado no construtor da classe Signer.
2. Preparação da mensagem para cifração AES com a adição de um padding, que faz com que a mensagem tenha tamanho divisível por 16 bytes, que é o tamanho do bloco no AES.
3. Cifração da mensagem com padding com a chave aleatória gerada para o AES.
4. Cálculo do hash SHA-3 da mensagem original com o tamanho de hash especificado no construtor da classe Signer.
5. Geração de um campo pela concatenação do hash, da chave AES e do tamanho do padding utilizado para o AES. Note que todos esses campos têm um tamanho especificado pelo usuário, o que é importante para a verificação.
6. Cifração do campo gerado no item 4 com a chave privada RSA fornecida.
7. Adição da chave pública fornecida para permitir a verificação da assinatura.

O esquema de verificação segue a mesma ideia, porém decifrando os campos ao invés de cifrar e removendo o padding gerado.

Se não é possível decifrar, a assinatura é apontada como inválida. Isso pode acontecer por tamanhos inesperados para o tamanho da mensagem AES.

A assinatura também é apontada como inválida se o tamanho dos campos da assinatura decifrada com a chave pública RSA não é o esperado ou se o hash calculado para a mensagem decifrada não é igual ao hash especificado.

# Referências

[1] Jonathan Katz e Yehuda Lindell. *Introduction to Modern Cryptography. Second edition.* CRC Press, 2015.