Universidade de Brasília - Semestre 2021/2
Segurança Computacional - CIC0201
Trabalho 2 - Gerador/Verificador de Assinatura

Professor:
João José Costa Gondim
Autores:
Gabriel Matheus da Rocha de Oliveira - 170103498
Pedro Victor Rodrigues de Carvalho - 170113043

- **Visão Geral**

O trabalho foi implementado utilizando a linguagem Python 3.8.10. O código consiste de 8 módulos: *"AES.py"*, *"RSA.py"*, *"eratosthenes_sieve.py"*, *"execute_AES.py, execute_RSA.py, execute_miller_rabin.py, miller_rabin.py e run_project.py".* O módulo chamado "run_project.py" integra os demais módulos para a realização do processo de geração e verificação de assinatura digital. Os módulos com nome *"execute_"* são módulos de teste isolado das partes do projeto.

Foram utilizadas as seguintes bibliotecas: *random*, *numpy e hashlib*. A biblioteca random foi utilizada para ter acesso ao método getrandbits() para gerar inteiros com uma quantidade específica de bits. Da biblioteca numpy, foi importado as funções array() e matmul() para a realização da multiplicação entre matrizes. E, por último, a biblioteca hashlib foi utilizada para a utilização dos métodos sha3_256 e sha3_512, implementações do hash SHA3.

O método de desenvolvimento utilizado foi a programação em pares, utilizando o github como plataforma para o controle de versão.

- **Módulo miller_rabin**

Não houve problemas nessa parte da implementação. O algoritmo de miller_rabin consiste em um teste probabilístico da primalidade de um dado número inteiro. Se o número passar no teste há uma probabilidade de 75% do número em questão ser primo, se não passar, o número com certeza é composto, ou seja, não primo. O módulo execute_miller_rabin.py gera os primos para a geração das chaves pública e privada que seriam utilizadas durante o processo criptográfico assimétrico RSA. Durante a implementação do projeto, entretanto, verificou-se que a natureza probabilística deste algoritmo não seria benéfica para os testes aplicados durante o desenvolvimento do projeto. Portanto, optou-se por outro algoritmo relativamente simples, que retorna números primos com total certeza e em tempo menor, para limites superiores pequenos. Esse algoritmo escolhido foi o Crivo de Eratóstenes.

- **Módulo eratosthenes_sieve**

Contém a implementação de um algoritmo simples para encontrar todos os números primos até um determinado limite especificado. Seu desenvolvimento foi relativamente mais simples e o resultado final mais satisfatório uma vez que retorna números garantidamente primos. O módulo RSA.py permite que as chaves sejam geradas utilizando tanto os números primos obtidos pelo algoritmo de Eratóstenes quanto pelo de Miller-Rabin.

- **Funções e variáveis auxiliares**

Para maior clareza e modularização do código criou-se diversas funções auxiliares, dentre as mais relevantes pode-se citar :

  - ***listXOR:*** Realiza a operação lógica ou exclusivo (XOR) entre os operadores de uma lista
  - ***listRotate:*** Rotaciona os elementos de uma lista.
  - ***padZeros:*** Retorna uma string com um número específico de '0'.
  - ***areCoprime:*** Verifica se dois números inteiros são coprimos.
  - ***bitsFromHash:*** Transforma uma string com caracteres '0' e '1' para um tipo bytes.

E utilizou-se as seguintes variáveis notáveis:

  - ***RCONTABLE:*** Armazena os dados da tabela Rcon, utilizada no passo generateRoundKeys do algoritmo AES.
  - ***BYTESTABLE:*** Armazena os dados da tabela de substituição de bytes, utilizada no passo subBytes do algoritmo AES.
  - ***GALOISMATRIX:*** Armazena a matriz utilizada nos produtos matriciais feitos no passo mixColumns do algoritmo AES.

- **Módulo AES**

Este módulo contém a implementação do algoritmo AES. Esse algoritmo é uma cifra de bloco simétrica. O algoritmo AES implementado utiliza chaves criptográficas de 128 bits e 10 rodadas (rounds) para criptografar e descriptografar dados em blocos de 128 bits. Cada round é composto por 4 operações sendo elas: AddRoundKey, SubBytes, ShiftRows e MixColumns. A criptografia usa o Rijndael Key Schedule, que deriva as subchaves da chave principal num processo denominado expansão de chave, expansão que define a lista de chave de rodada que é usada em cada rodada mais uma rodada inicial adicional. A mensagem, nesse algoritmo, é dividida em blocos de 128 bits que são interpretados como uma matriz 4x4.

32	88	31	e0
43	5a	31	37
f6	30	98	07
a8	8d	a2	34

Interpretação de bloco de 128 bits como matriz 4x4.

  - **Descrição de funcionamento**

O modo de operação do AES utilizado nesta implementação foi o CTR (counter).
Nesse arquivo se encontram as seguintes funções:

  - ***CTRmode:*** Garante o modo de operação CTR utilizando a cifra de bloco AES
  - ***block_encryption:*** Realiza as 4 operações de acordo com a cifra de bloco AES com 128 bits.
  - ***generateRoundKeys:*** Gera uma lista que contém todas as chaves utilizadas na cifra de bloco AES.
  - ***addRoundKey:*** Realiza a operação XOR das colunas da mensagem com as colunas da chave da rodada.

- ***subBytes:*** Checa a tabela BYTESTABLE e substitui o byte presente na mensagem pelo correspondente indicado pela tabela.
- ***shiftRows:*** Realiza rotações nas linhas da mensagem.
- ***mixColumns:*** Calcula uma multiplicação matricial de cada coluna da mensagem com uma matriz armazenada na variável GALOISMATRIX.

- **Módulo RSA**

Esse módulo concentra as implementações das funcionalidades do algoritmo RSA usando OAEP e hashing SHA3. A partir de dois primos, geram-se duas chaves: uma pública e uma privada. Essas chaves são utilizadas, juntamente com a mensagem, para realizar a assinatura da mensagem por meio da equação:

$$enc.msg = hash.msg^{key[0]} (mod\ key[1])$$

onde key representa uma das duas chaves geradas anteriormente.

  - **Descrição de funcionamento**

Nesse arquivo se encontram as seguintes funções:

- ***generateKeys:*** Gera primos dado um tamanho máximo definido globalmente para gerar chaves para o algoritmo RSA
- ***oaep:*** Realiza padding de acordo com o padrão OAEP.
- ***encrypt:*** Gera nonce, utiliza OAEP e retorna encriptação RSA.
- ***reverse_oap:*** Realiza o processo inverso do padrão OAEP.
- ***decrypt:*** Obtém a mensagem do RSA e reverte o processo OAEP.
- ***decrypt_hash:*** Retorna a mensagem obtida pelo RSA sem reverter o OAEP e retorna também o nonce encontrado.

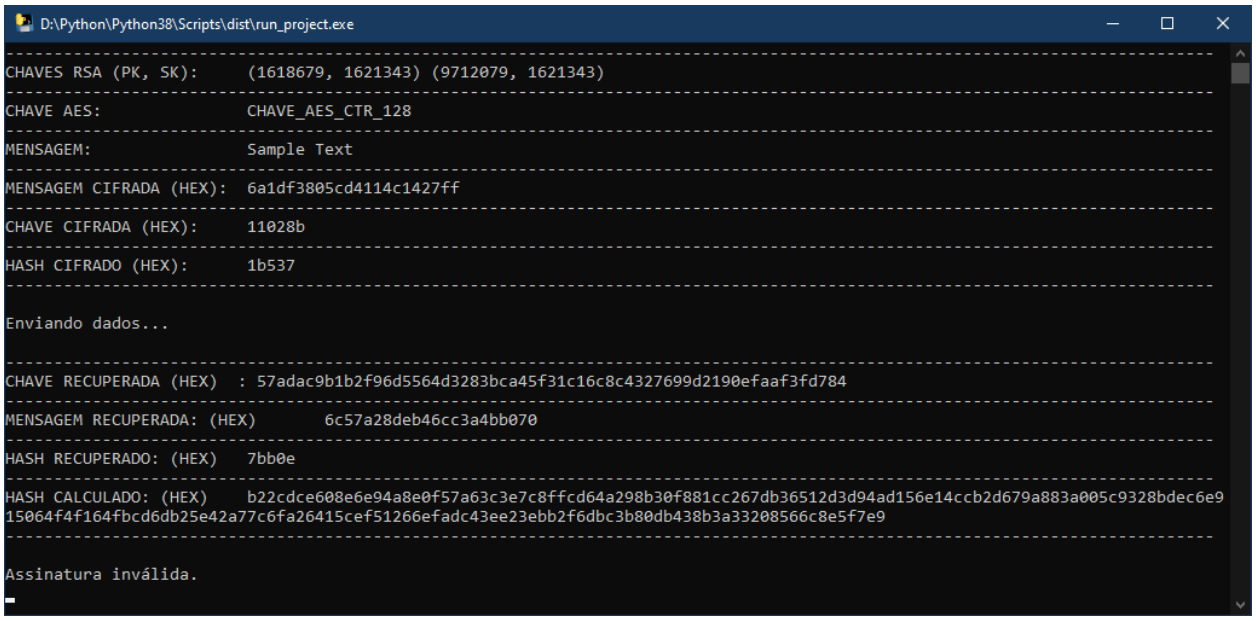
- **Módulo run_Project**

Esse módulo contém o escopo geral do trabalho, englobando os algoritmos AES e RSA, e simulando uma transmissão. Simula-se a seguinte situação:

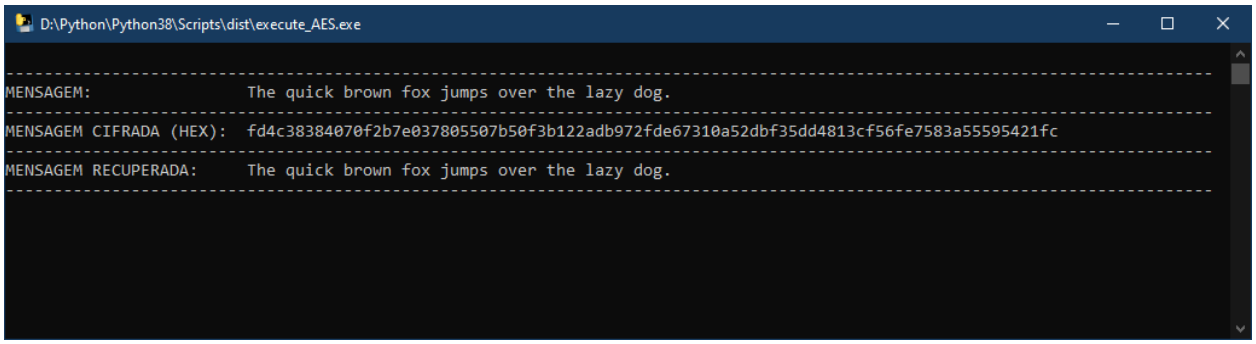
1. O remetente gera a mensagem e as chaves para os algoritmos AES e RSA;
2. A mensagem é cifrada com o AES;
3. A chave utilizada na cifração AES e o hash da mensagem são assinados com RSA;
4. O remetente envia para o destinatário:
    a. A chave pública;
    b. A mensagem cifrada com AES;
    c. A chave da cifração AES assinada com RSA;
    d. O hash da mensagem assinado com RSA.
5. O destinatário recebe todos os dados enviados;
6. É recuperada a chave de cifração AES por meio do RSA;
7. É decifrada a mensagem por meio do AES com a chave recuperada;
8. É recuperado o hash da mensagem por meio do RSA;
9. É calculado um novo hash para a mensagem decifrada;
10. Checa-se identidade entre hash calculado e recuperado.

A seguir estão prints dos resultados dos executáveis fornecidos.

run_project.exe:



execute_AES.exe



- **Considerações finais**

O grupo encontrou diversos desafios durante a realização do projeto, sendo o principal deles a dificuldade e incapacidade de testar e verificar o funcionamento dos algoritmos devido a presença de números primos de grande magnitude, uma vez que possuem pelo menos 1024 bits, ou seja, 308 dígitos em decimal. A presença de tais números tornou a execução do programa utilizando os parâmetros solicitados inviável, possibilitando apenas testes com valores extremamente reduzidos. Não foi possível implementar a formatação do resultado para verificação em BASE64 e devido a incapacidade de testar eficientemente a implementação do algoritmo RSA o resultado obtido pelo mesmo foi insatisfatório o que comprometeu o resultado final do processo de assinatura digital.