UnB - Universidade de Brasília
CIC0201 - Segurança Computacional – 2023/1
Professor: João Gondim
Alunos: Lorenzo Martins Lazzarin (200022610); Ayrton Jorge Nassif (200048805)

Trabalho de Implementação 2 - Gerador/Verificador de Assinaturas

Estrutura do projeto e Recursos

O projeto foi implementado utilizando a linguagem de programação Python, e está estruturado em classes. Há testes de execução dentro de cada classe, entretanto, o fluxo de execução inicia a partir do arquivo run.py ou o run_failure_case.py, sendo que a execução do run_failure_case.py é um caso com em que a mensagem é comprometida, não sendo possível validar a assinatura. Os dois fluxos podem ser executados com os comandos *python3 src/run.py* ou *python3 src/run_failure_case.py* a partir da raiz do projeto.

Teste de primalidade por Miller-Rabin (Classe MillerRabin)

Os testes de primalidade buscam determinar se um número é primo ou composto. Sendo que, a sua utilização é necessária por conta da criptografia RSA, necessita de números primos muito grandes para garantir sua segurança. O Teste de Primalidade de Miller-Rabin usa como base o Teste de Primalidade de Fermat. Para implementação do método **__primality** da classe **MillerRabin** foi utilizado o algoritmo que encontra-se nas referências. Em que por meio do método **primeGenerate** são gerados um número primo a partir de um tamanho (em bits) previamente definido pelo atributo da classe **MillerRabin**, e esse número é verificado pelo método **_primality**.

Descrição da classe:

- **keyLen**: atributo de classe que define o tamanho do primo gerado em bits.
- **primeGenerae(self, **args)**: método responsável por gerar um número primo de tamanho **keyLen** usando o método **getrandbits** da classe **random**, e também garantir que o número é primo através do método **__primality.**
- **setBit(self, value, bit)**: método privado responsável por fazer os "sets" nos bits do número gerado para garantir que ele é ímpar.
- **__primality(self, n: int, t=2000)**: método privado responsável por realizar o teste de primalidade no número gerado. Recebe um inteiro **"n"** e um valor **"t"**

que indica a quantidade de testes, sendo que há um valor default para **“t”** de 2000.

AES

O **Advanced Encryption Standard (AES**) especifica um algoritmo criptográfico aprovado pelo FIPS que pode ser usado para proteger dados eletrônicos. O algoritmo AES é uma cifra de bloco simétrico que pode criptografar (codificar) e descriptografar (decifrar) informações.
A criptografia converte os dados em uma forma ininteligível chamada texto cifrado; descriptografar o texto cifrado converte os dados de volta em sua forma original, chamada de texto simples. O algoritmo AES é capaz de usar chaves criptográficas de 128, 192 e 256 bits para criptografar e descriptografar dados em blocos de 128 bits. – NIST

O AES é uma variante da família Rijndael de algoritmos de criptografia de bloco simétrico, que é uma combinação dos nomes de dois criptógrafos belgas, Joan Daemen e Vincent Rijmen.

Para ver como o padrão de criptografia avançado realmente funciona, no entanto, primeiro precisamos ver como isso é configurado e as “regras” relativas ao processo com base na seleção do usuário da força da criptografia. Normalmente, quando discutimos o uso de níveis de bits mais altos de segurança, observamos coisas que são mais seguras e mais difíceis de quebrar ou hackear. Embora os blocos de dados sejam divididos em 128 bits, o tamanho da chave tem alguns comprimentos variados: 128 bits, 196 bits e 256 bits.

Sabemos que a criptografia normalmente lida com a codificação de informações em algo ilegível e uma chave associada para descriptografar a codificação. Os procedimentos de embaralhamento AES usam quatro operações de embaralhamento em rodadas, o que significa que ele executará as operações e, em seguida, repetirá o processo com base nos resultados da rodada anterior X número de vezes. De forma simplista, se colocarmos X e obtivermos Y, isso seria uma rodada. Em seguida, colocaríamos Y à prova e obteríamos Z para a rodada 2. Enxágue e repita até termos concluído o número especificado de rodadas.

O tamanho da chave AES, especificado acima, determinará o número de rodadas que o procedimento executará. Por exemplo:

- Uma chave de criptografia AES de 128 bits terá 10 rodadas.
- Uma chave de criptografia AES de 192 bits terá 12 rodadas.
- Uma chave de criptografia AES de 256 bits terá 14 rodadas.

Os dados sobre os quais as operações são realizadas não são sequenciais da esquerda para a direita como normalmente pensamos. Ele é empilhado em uma matriz 4 × 4 de 128 bits (16 bytes) por bloco em uma matriz que é conhecida como “estado”. Um estado seria uma matriz, exemplificada a seguir:

$$\begin{bmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{bmatrix}$$

Portanto, se sua mensagem fosse “Pill blue or red”, seria algo assim:

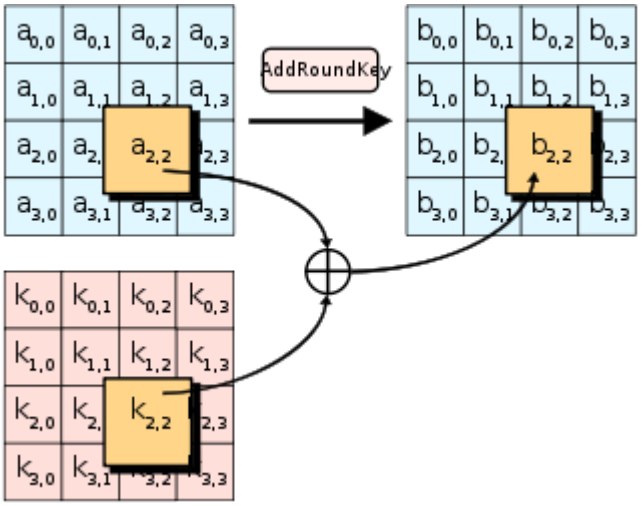
$$\begin{bmatrix} B & \square & L & \square \\ L & P & \square & R \\ U & I & O & E \\ E & L & R & D \end{bmatrix}$$

Este é apenas um bloco de 16 bytes – então, isso significa que cada grupo de 16 bytes em um arquivo é organizado dessa maneira. Nesse ponto, o embaralhamento sistemático começa com a aplicação de cada operação de criptografia AES.

Conforme mencionado anteriormente, uma vez que tenhamos nosso arranjo de dados, existem certas operações vinculadas que realizarão o embaralhamento em cada estado. O objetivo aqui é converter os dados de texto simples em texto cifrado por meio do uso de uma chave secreta.

Os quatro tipos de operações AES são os seguintes:

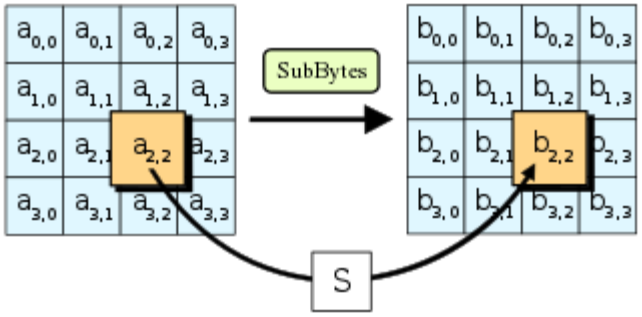
- Expansão de chave e AddRoundKey



Conforme mencionado anteriormente, o tamanho da chave determina o número de rodadas de embaralhamento que serão realizadas. A criptografia AES usa o Rjindael Key Schedule, que deriva as subchaves da chave principal para realizar a expansão da chave.

A operação AddRoundKey obtém o estado atual dos dados e executa a operação booleana XOR na subchave round atual. XOR significa “Exclusivamente Ou,” que produzirá um resultado verdadeiro se as entradas forem diferentes (por exemplo, uma entrada deve ser 1 e a outra entrada deve ser 0 para ser verdadeira). Haverá uma subchave exclusiva por rodada, mais uma (que será executada no final).

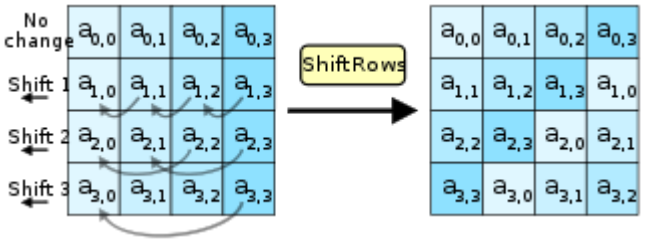
- SubBytes



A operação SubBytes, que significa bytes substitutos, pegará o bloco de 16 bytes e o executará por meio de uma S-Box (caixa de substituição) para produzir um valor alternativo. Simplificando, a operação pegará um valor e então o substituirá cuspindo outro valor.

A operação real do S-Box é um processo complicado, mas saiba que é quase impossível decifrar com a computação convencional. Juntamente com o resto das operações AES, ele fará seu trabalho para embaralhar e ofuscar com eficácia os dados de origem. O “S” na caixa branca na imagem acima representa a tabela de pesquisa complexa para a S-Box.

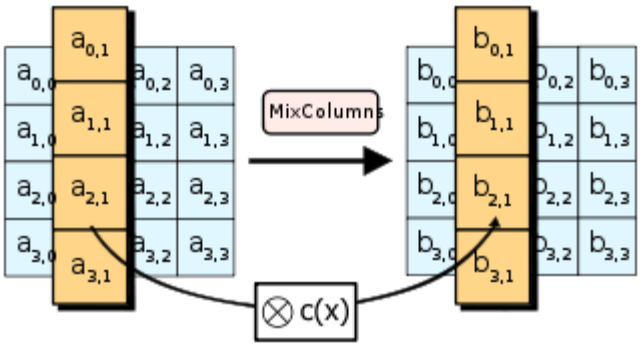
- ShiftRows



A operação ShiftRows é um pouco mais direta e mais fácil de entender. Com base na organização dos dados, a ideia do ShiftRows é mover as posições dos dados em suas respectivas linhas com quebra automática. Lembre-se de que os dados são organizados empilhados e não da esquerda para a direita, como a maioria de nós está acostumada a ler. A imagem fornecida ajuda a visualizar esta operação.

A primeira linha permanece inalterada. A segunda linha desloca os bytes para a esquerda em uma posição com a quebra de linha. A terceira linha desloca os bytes uma posição além disso, movendo o byte para a esquerda por um total de duas posições com quebra de linha. Da mesma forma, isso significa que a quarta linha desloca os bytes para a esquerda em um total de três posições com a quebra da linha.

- MixColumns



A operação MixColumns, em poucas palavras, é uma transformação linear das colunas do conjunto de dados. Ele usa multiplicação de matrizes e adição XOR bit a bit para gerar os resultados. Os dados da coluna, que podem ser representados como uma matriz 4 × 1, serão multiplicados por uma matriz 4 × 4 em um formato denominado campo Gallois e definido como um inverso de entrada e saída. Isso será parecido com o seguinte:

$$\begin{bmatrix} M_0 & M_1 & M_2 & M_3 \\ M_3 & M_0 & M_1 & M_2 \\ M_2 & M_3 & M_0 & M_1 \\ M_1 & M_2 & M_3 & M_0 \end{bmatrix} \times \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

Acima vemos que há quatro bytes em que são executados em uma matriz 4 × 4. Nesse caso, a multiplicação da matriz tem cada byte de entrada afetando cada byte de saída e, produz o mesmo tamanho.

A ordem em que essas operações que são executadas é:

- Expansão de chave define a lista de chave de rodada que é usada em cada rodada mais uma rodada adicional (e inicial, como você verá).
- AddRound é a primeira etapa para ofuscar os dados. Imediatamente, temos dados embaralhados.
- Agora entramos nas rodadas de intensa mistura de dados. Novamente, dependendo do bit de cifra selecionado, o número de rodadas será diferente.
- Para 9, 11 ou 13 rodadas, dependendo da seleção do bit de cifra, o seguinte será executado nesta ordem:
- SubBytes
- ShiftRows
- MixColumns
- AddRound
- No 10° , 12° , ou 14° rodada, respectivamente, que executam o conjunto final das operações, que são os seguintes:
- SubBytes
- ShiftRows
- AddRound

OEAP

Na criptografia, o Optimal Asymmetric Encryption Padding (OAEP) é um esquema de preenchimento frequentemente usado junto com a criptografia RSA. OAEP foi introduzido por Bellare e Rogaway,[1] e posteriormente padronizado em PKCS#1 v2 e RFC 2437.

O algoritmo OAEP é uma forma de rede Feistel que usa um par de oráculos aleatórios G e H para processar o texto simples antes da criptografia assimétrica. Quando combinado com qualquer permutação unidirecional segura,

provou-se que esse processamento no modelo de oráculo aleatório resulta em um esquema combinado que é semanticamente seguro sob ataque de texto simples escolhido (IND-CPA). Quando implementado com certas permutações de trapdoor (por exemplo, RSA), o OAEP também provou ser seguro contra o ataque de texto cifrado escolhido. O OAEP pode ser usado para criar uma transformação tudo ou nada.

OAEP satisfaz os dois objetivos a seguir:

- Adicione um elemento de aleatoriedade que pode ser usado para converter um esquema de criptografia determinístico (por exemplo, RSA tradicional) em um esquema probabilístico.
- Evite a descriptografia parcial de textos cifrados (ou outro vazamento de informações), garantindo que um adversário não possa recuperar qualquer parte do texto simples sem ser capaz de inverter a permutação unidirecional do alçapão

Cifragem/Decifragem usando o RSA (Classe RSA)

O RSA é basicamente o resultado de dois cálculos matemáticos. Um para cifrar e outro para decifrar. O RSA usa duas chaves criptográficas, uma chave pública e uma privada. No caso da criptografia assimétrica tradicional, a chave pública é usada para criptografar a mensagem e a chave privada é usada para descriptografar a mensagem. A classe **RSA** é responsável por realizar todas as implementações das funcionalidades do algoritmo RSA usando OAEP, hashing SHA3 e o Base64.

A classe **RSA** herda a classe **MillerRabin,** e por meio dela são gerados os primos, e em seguida são gerados as duas chaves pelo método **generateKeys**, após a geração das chaves é realizado o preenchimento a clara da mensagem usando o método **oaep**, realizando a assinatura da mensagem por meio da equação:

$$M_{Cifrado} = oaep(M_{Claro})^{kU[0]} \bmod kU[1]$$

Já a verificação e a cifragem da mensagem é realizada por meio das chaves privadas por meio da seguinte equação:

$$M_{claro} = reverseOaep(M_{Cifrado}^{kR[0]} \bmod kR[1])$$

Descrição da classe:

- **kU**: atributo da classe que é uma tupla com a chave pública
- **kR**: atributo da classe que é uma tupla com a chave privada
- **DEFAULT_SIZE_PADDING**: atributo constante da classe que representa um tamanho padrão do preenchimento
- **oaep_send** e **oaep_receive**: atributos da classe que representam o hash de envio e recebimento da mensagem, que é utilizado para verificar assinatura e autenticidade da mensagem.
- **__coprime**: método privado responsável por verificar se dois números são coprimos.
- generateKeys: método responsável por gerar as chaves: privada e pública.
- **__XOR**: método privado responsável por realizar a operação XOR (OR exclusivo) para cada posição da string de bits.
- **__strToByte**: método privado responsável por realizar a conversão de binário (string) para bytes.
- **__bitStrToBytes**: método privado responsável por realizar a conversão de bit em forma de string para byte.
- **toBase64**: método responsável por realizar o base64 de string.
- oaep: método responsável por realizar o preenchimento da mensagem. Nesse método é obtido os valores do dict que são as duas partes do hash do envio da mensagem.
- **reverseOaep**: método responsável por realizar o processo inverso do preenchimento da mensagem. Nesse método é obtido os valores do dict que são as duas partes do hash de recebimento da mensagem.
- **__encoderFunction**: método privado auxiliar que é responsável por efetuar o cálculo da cifração dos caracteres da mensagem.
- **__decoderFunction**: método privado auxiliar que é responsável por efetuar o cálculo da decifração dos caracteres da mensagem.
- **encoder**: método responsável por realizar a chamada do do método que realiza o preenchimento e cifração da mensagem. Nesse método foi implementado uma flag para indicar o uso do preenchimento oaep.
- **decoder**: método responsável por realizar a chamada do método que realiza a decifração da mensagem e desfazer o preenchimento

Arquivos de fluxo de execução

A execução do programa pode começar pelo arquivo run.py em que será solicitado o tamanho da chave e a mensagem a ser enviada. Entretanto, o programa pode ser executado pelo arquivo run_failure_case.py que simula um caso que a mensagem é corrompida, não sendo possível autenticar a confiabilidade da mensagem.

Considerações

Durante o desenvolvimento do trabalho foram encontrados vários desafios, entretanto foi possível implementar todas as etapas propostas, o algoritmo de cifragem usando o preenchimento está bastante ineficiente para mensagens e chaves longas.

Referências Bibliográficas

1 DE LIMA, Daniel Chaves. Trabalho de Conclusão de Curso: **Algoritmo para o teste de Primalidade de Miller-Rabin**. Universidade Federal do Pará Campus de Castanhal: Faculdade de Matemática, 2019.

2 ANDERSON, Ross J.. **Security Engineering**: A Guide to Building Dependable Distributed Systems Second Edition. Indianapolis, Indiana: Wiley Publishing, Inc., 2008.

3 BARBOSA, Luis Alberto De Moraes; BRAGHETTO, Luis Fernando B; BRISQUI, Marcelo Lotierso; DA SILVA, Sirlei Cristina. **RSA Criptografia Assimétrica e Assinatura Digital**. Campinas: UNICAMP – Universidade Estadual de Campinas, 2023.

4 SHOUP, Victor. **OAEP Reconsidered**. Switzerland: IBM Zurich Research Lab, 2001. KATZ, Jonathan; LINDELL, Yehuda. **Introduction to Modern Cryptography**.Boca Raton London New York Washington, D.C: CRC PRESS, 2007.