Gerador e verificador de assinaturas em um sistema criptográfico híbrido

Gabriel Rocha Fontenele, 15/0126760

[1] Dep. Ciência da Computação – Universidade de Brasília (UnB)
Julho de 2023

CIC0201 - Segurança Computacional
Turma 01 - Prof. João José da Costa Gondim

1. Introdução

O presente documento descreve a implementação de um gerador e verificador de assinaturas em arquivos utilizando os métodos de cifração e decifração AES (Advanced Encryption Standard)[1] e RSA (Rivest–Shamir–Adleman)[2]. O criptosistema desenvolvido foi implementado utilizando a linguagem de programação Python, e sua execução se dá, a partir do diretório raíz com todos os módulos do projeto presentes, através da execução de uma linha de comando em um terminal. Veja o Listing 1.

Por se tratar de algoritmos conhecidos, os detalhes das descrições do AES, RSA e OAEP (Optimal Asymmetric Encryption Padding)[2] são omitidos, mas podem ser consultados a partir das **Referências**. A implementação de cada algoritmo de criptografia, bem como a obtenção do código-fonte do projeto, podem ser encontrados acessando o repositório do projeto no GitHub: https://github.com/ngsylar/syfracao.

```
python3 MAIN.py
```

Listing 1. Comando para início de execução via terminal

1.1. Entrada e Saída

O programa oferece as várias opções no início da execução, cabendo ao usuário inserir a opção desejada e em seguida pressionar a tecla *Enter*. Subsequentemente, o usuário deve inserir corretamente os dados que forem pedidos, os quais em sua maioria são referentes aos nomes dos arquivos a serem lidos ou escritos. Os arquivos de entrada e saída devem se encontrar na pasta raiz do programa e possuem extensão *.txt* ou *.k3y*, contendo o conteúdo das mensagens e as chaves criptográficas respectivamente. Ao inserir os nomes dos arquivos, deve-se deixar de fora as extensões, pois elas serão adicionadas automaticamente pelo programa.

O algoritmo de cifração do AES retorna cifras e chaves como cadeias de caracteres com codificação ISO-8859-1 convertidas para Base64[3] e o algoritmo de cifração do RSA, por sua vez, retorna cifras e chaves como inteiros, que são convertidos para cadeias de caracteres de forma literal para posterior escrita em um arquivo de saída.

2. Cifra simétrica

2.1. AES-CTR

O AES recebe uma mensagem de texto e gera uma chave pseudoaleatória de 16 bytes. Cada caractere do texto é convertido para um octeto de bits correspondente, e

em seguida o texto é divido em vários blocos de 16 bytes cada, com o último bloco podendo ser de tamanho menor. A partir da chave principal são produzidas chaves de bloco derivadas tal que cada uma consiste na cifra da chave principal com um *nonce* (number used once) através do algoritmo de Rijndael[1]. Um *nonce* é a concatenação de um vetor de inicialização *iv* – uma única entrada pseudoaleatória de tamanho fixo – com um valor de contador, começando a contagem no valor 1 e incrementando o valor para cada bloco. Assim, são obtidos *nonce*s, e consequentemente as chaves de bloco na mesma quantidade de blocos da mensagem, cada uma com 16 bytes.

Cada bloco de mensagem e cada chave de bloco correspondentes – mensagem com valor de contador n e chave com valor de contador $n + 1$ – fazem XOR resultando num bloco de cifra. Cada bloco de cifra é concatenado em ordem e o vetor de bytes resultante é a cifra da mensagem no AES. Por fim, o *iv* é concatenado com o tamanho do *authData* (dados de autenticação) em bytes, o *authData* – uma cadeia de caracteres com tamanho máximo de 16 bytes –, a cifra final obtida e o *tag* – um vetor de bytes obtido pelo GCM (Galois Counter Mode)[4], executado após o processo de cifração.

Para decifrar a mensagem cifrada, o AES recebe a cifra e a chave principal, repete toda a operação de cifração da chave para obter as chaves de bloco e faz a operação XOR de cada chave de bloco com cada bloco da mensagem cifrada, obtendo como resultado a mensagem original.

2.2. GCM

O GCM recebe como entradas o *iv*, uma cadeia de caracteres chamada de *authData* (dados de autenticação), a cifra resultante do AES, e a chave principal. Uma *hashKey* é gerada a partir da cifração de um bloco de 16 bytes de zeros com a chave principal pelo algoritmo de Rijndael. Em seguida o *authData* é convertido para um valor numérico, que é multiplicado pelo valor numérico da *hashKey* módulo 128 bits no campo de Galois[5], resultando no primeiro valor numérico do *tag*.

Para cada bloco, o próximo valor do *tag* será o resultado de um XOR do valor atual com um bloco da cifra e em seguida multiplicado pela *hashKey* módulo 128 no campo de Galois. Após o último bloco, o valor do *tag* passa a ser um XOR com a concatenação dos tamanhos do *authData* e da cifra do AES, com o resultado da operação também multiplicada pela *hashKey* módulo 128 bits no campo de Galois. Obtemos o *tag* final fazendo um XOR deste último valor com cifra AES do *iv* concatenado com o contador no valor 0.

Antes do processo de a decifração pelo AES, são retirados da cifra o *iv*, o *authData* e seu tamanho e o *tag*. Repete-se o mesmo processo usado pelo GCM para obter o *tag* e compara o resultado deste novo *computedTag* com o *tag* retirado da cifra. Caso sejam iguais, a mensagem não sofreu nenhuma alteração. Caso contrário, a autenticidade ou a integridade da mensagem foi comprometida.

3. Cifra assimétrica

3.1. Geração de chaves

Moriarty et al. descreve os tamanhos padrões de chaves para o RSA[2], denotando que quanto maior o número de bits presentes na chave, mais segura será a cifração.

O menor tamanho de chave reconhecido como seguro é de 2048 bits e é tomado como tamanho mínimo para um valor númerico único chamado *modulus*, que representa o valor do módulo nas operações do RSA. Para obtê-lo, são gerados dois números primos aleatórios *p* e *q*, cada um com tamanho mínimo de 1024 bits, através do teste de primalidade de Miller-Rabin[6].

Para isso, primeiramente deve-se gerar números ímpares aleatórios com 1024 bits cujo bit mais significativo seja igual a 1. Em seguida, cada ímpar gerado é usado como entrada de uma função que retorna verdadeiro caso o valor passe no teste de primalidade. Assim, com *p* e *q* verdadeiros na saída da função *IsPrime*, obtém-se o *modulus* através de $p \cdot q$.

O Miller-Rabin testa para um *a* aleatório tal que $2 < a < n - 1$, com *n* sendo o número ímpar a ser testado, se a condição (1) é satisfeita – Lema de Euclides. Caso a condição não seja satisfeita, existe uma alta probabilidade de *n* ser composto. O teste é repetido cerca de 40 vezes para diferentes valores de *a* até que se ache dois primos *p* e *q*.

$$a^{n-1} \equiv 1 \mod n \tag{1}$$

Em seguida é calculado o $\lambda(modulus)$ através da função totiente de Charmicael, como mostra a Equação (2) e obtém-se o expoente público *publicExp* tal que a condições (3) e (4) sejam satisfeitas. Na prática, escolhe-se o valor 65537. Caso as condições não sejam satisfeitas, esse valor é recalculado. A chave pública então será uma tupla composta por *modulus* e *publicExp*.

$$\lambda(modulus) = \operatorname{lcm}(\lambda(p), \lambda(q)) = \operatorname{lcm}(\phi(p), \phi(q)) = \operatorname{lcm}(p-1, q-1) \tag{2}$$

$$2 < publicExp < \lambda(modulus) \tag{3}$$

$$\gcd(publicExp, \lambda(modulus)) = 1 \tag{4}$$

A chave privada será uma tupla composta por *modulus* e pelo expoente privado *privateExp*, cujo valor se dá ao resolver a congruência na Equação (5).

$$privateExp \equiv publicExp^{-1} \mod \lambda(modulus) \tag{5}$$

3.2. RSA

Geradas as chaves pública e privada, a mensagem a ser cifrada é convertida para um valor inteiro e a cifra é obtida a partir da Equação (6). De forma semelhante, ao decifrar a mensagem cifrada, obtém-se a mensagem original a partir da Equação (7).

$$cifra = mensagem^{publicExp} \mod modulus \tag{6}$$

$$mensagem = cifra^{privateExp} \mod modulus \tag{7}$$

3.3. OAEP

O OAEP utiliza a duas funções de hash: o SHA3-256 (Secure Hash Algorithm 3)[7] e o MGF1 (Mask Generation Function 1)[2]. O OAEP recebe a mensagem antes de ser cifrada com o RSA, cria um hash de 32 bytes usando o SHA3-256, uma *seed* aleatória de mesmo tamanho e um vetor de bytes zero *PS* cujo tamanho é dado pela Equação (8).

O *seed* e a concatenação dos demais valores obtidos passam pelo MGF1 e por operações XOR, resultando em uma máscara de bits dos dados envolvidos. Essa máscara será passada como mensagem para ser cifrada pelo RSA e também será o resultado da decifração do RSA, fazendo-se necessário que o texto decifrado passe pelo processo inverso do OAEP para obtenção da mensagem original.

$$|PS| = (|modulus| \text{ em bytes}) - |mensagem| - 66 \text{ bytes} \tag{8}$$

4. Assinaturas

4.1. Cifra híbrida

A cifra híbrida combina o AES-GCM com o RSA-OAEP, cuja mensagem é cifrada pelo AES-GCM com uma chave de 16 bytes gerada, que por sua vez é cifrada pelo RSA-OAEP com uma chave pública gerada previamente em conjunto com uma chave privada que permanece em segredo. Desta forma, para obter novamente a mensagem original, faz-se necessária a utilização da chave privada para decifrar a chave que será usada para decifrar a mensagem cifrada. No caso da cifra híbrida com autenticação mútua, o RSA-OAEP ainda é combinado com o ECB (Electronic Codebook).

4.2. Geração de assinatura

Após cifrar uma mensagem com o AES-GCM, a cifra obtida é passada como entrada para a função de hash SHA3-256, que retorna uma hash de 32 bytes baseada na entrada. Essa hash é cifrada pelo RSA-OAEP com uma chave secreta *A*, cujo resultado é passado na saída junto com a cifra AES-GCM da mensagem – codificada em Base64 –, a cifra RSA-OAEP da chave da mensagem – cifrada com uma chave pública *B* –, e a chave pública correspondente a *A*. Veja o Listing 3.

```
def opt6 ():
    secretKey_A = file.read(file_with_secretKey_A)
    publicKey_A = file.read(file_with_publicKey_A)
    publicKey_B = file.read(file_with_publicKey_B)
    message = file.read(file_with_message)

    mainKey = AES_GCM.GenerateKey(16)
    cipher = AES_GCM.Cipher(message, mainKey)
    cipherKey = RSA.Cipher(mainKey, PublicKey_B)

    hashCipher = sha3_256(cipher)
    sign = RSA.Cipher(hashCipher, secretKey_A)
    newFile = file.write(Base64.Enc(cipher), cipherKey, sign,
   publicKey_A)
```

Listing 2. Função de cifração e geração de assinatura – com pequenas alterações

4.3. Verificação de assinatura

Para verificar a assinatura, toma-se da saída obtida na Seção 4.2 a cifra AES-GCM, que é passada já decodificada para a função de hash SHA3-256. Usa-se a chave pública *A* para decifrar a hash e em seguida a hash decifrada é comparada com a hash calculada anteriormente. Se os resultados forem iguais, a cifração está correta e os dados são autênticos. Assim, é possível decifrar a chave principal com a chave privada *B* e a partir da chave principal decifrar a mensagem cifrada decodificada em Base64.

Se tudo ocorrer de maneira correta e nenhum erro for detectado durante a verificação, a mensagem decifrada será igual a mensagem original. Veja o Listing **??**.

```
def opt11 ():
    secretKey_B = file.read(file_with_secretKey_B)
    cipherB64, cipherKey, sign, publicKey_A = file.read(
   file_with_opt6_out)

    cipher = Base64.Dec(cipherB64)
    hashCipher = sha3_256(cipher)
    givenHash = RSA.Decipher(publicKey_A, sign)

    if hashCipher != givenHash:
        print("\nAviso: a assinatura nao corresponde")
        return

    mainKey = RSA.Decipher(cipherKey, secretKey_B)
    message = AES_GCM.Decipher(cipher, mainKey)
    newFile = file.write(message)
```

Listing 3. Função de decifração e verificação de assinatura – com pequenas alterações

Referências

[1] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, Federal Information Processing Standards (FIPS) Publication 197, Novembro de 2001, Atualizado Maio 2023.

[2] K. Moriarty, Ed., B. Kaliski, J. Jonsson, A. Rusch, *PKCS #1: RSA Cryptography Specifications Version 2.2*, Internet Engineering Task Force (IETF). Disponível em https://www.rfc-editor.org/rfc/rfc8017, Novembro de 2016.

[3] *Base64*, Conteúdo Aberto: Wikipédia: a enciclopédia livre. Disponível em https://pt.wikipedia.org/wiki/Base64, Acesso em Junho de 2023.

[4] David A. McGrew, John Viega, *The Galois/Counter Mode of Operation (GCM)*, National Institute of Standards and Technology. Disponível em https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf, Recuperado em Julho de 2013.

[5] fcicq, *Finite Field / Galois Field (GF(2n̂)) in python 2.7+*, Conteúdo Aberto: GitHub. Disponível em https://gist.github.com/fcicq/4352289, Acesso em Junho de 2023.

[6] *Miller–Rabin primality test*, Conteúdo Aberto: Wikipédia: a enciclopédia livre. Disponível em https://en.wikipedia.org/wiki/Miller–Rabin_primality_test, Acesso em Junho de 2023.

[7] *SHA-3*, Conteúdo Aberto: Wikipédia: a enciclopédia livre. Disponível em https://en.wikipedia.org/wiki/SHA-3, Acesso em Junho de 2023.