Gerador/Verificador de Assinaturas RSA-PSS: Implementação de Sistema Completo de Assinatura Digital

Seminário de Segurança Computacional
Vinícius Da Silva Araújo - 221001981
Caue De Macedo Britto Trindade De Sousa - 231019003

18 de julho de 2025

Resumo

Este relatório apresenta a implementação completa de um gerador e verificador de assinaturas em cima de arquivos utilizando o algoritmo RSA-PSS.

1 Introdução

A criptografia de chave pública, especificamente o algoritmo RSA (Rivest-Shamir-Adleman), foi um enorme passo para a segurança digital moderna e continua sendo um de seus pilares. O esquema RSA-PSS (Probabilistic Signature Scheme) representa uma evolução do RSA tradicional, incorporando elementos probabilísticos que aumentam significativamente a segurança contra ataques adaptativos.

Este trabalho implementa um gerador e verificador de assinaturas RSA-PSS em arquivos, conforme especificações do roteiro de programação. O sistema desenvolvido atende a todos os requisitos estabelecidos, implementando as três partes principais: geração de chaves e cifra, assinatura digital, e verificação de assinaturas.

O código pode ser encontrado no repositório: https://github.com/cauetrd/rsa-pss

1.1 Tecnologias Utilizadas

O projeto foi desenvolvido utilizando a linguagem de programação `C++`, e o sistema implementa por conta própria todas as suas funcionalidades, com exceção das seguintes tecnologias externas utilizadas:

Biblioteca *GNU multi precision*, para aritmética de inteiros com mais de 128 bits (https://gmplib.org/manual/C_002b_002b-Class-Interface). Os inteiros maiores de 128 bits são identificados pela classe mpz_class;

Função de Hash *sha3_256* desenvolvido pela equipe Keccak ganhadora da competição NIST para SHA3 (https://keccak.team/keccak.html), com código copiado do repositório oficial e público da equipe Keccak: https://github.com/XKCP/XKCP/blob/master/lib/high/Keccak/F

2 Desenvolvimento

https://www.rfcreader.com/rfc8017_line1391

2.1 Matemática

Para implementação das operações matemáticas não ordinárias utilizadas, foi implementado o módulo `Maths`. Para manter o enfoque na segurança, não serão apresentadas as funções neste relatório. Contudo, fica o convite para que o leitor visualize-as no repositório.

Vale a pena ressaltar, entretanto, a função de teste de primalidade de Miller-Rabin:

O teste de Miller-Rabin é um algoritmo probabilístico utilizado para verificar se um número ímpar $n > 3$ é primo. Ele se baseia no teorema de Fermat e consiste em escrever $n - 1 = 2^r \cdot d$, com d ímpar, e então escolher aleatoriamente bases a no intervalo $[2, n-2]$. Para cada base, verifica-se se $a^d \equiv 1 \mod n$ ou $a^{2^j \cdot d} \equiv -1 \mod n$ para algum $j < r$; se nenhuma dessas condições for satisfeita, n é composto. Caso contrário, o número pode ser primo.

```
// returns true if a is probably prime and false if a is definetly composite
// for each iteration, chances of identifying a composite as prime is <= 1/4
// chances for misidentification of algorithm is (1/4)^k, k being the number of iterations
bool MillerRabin(mpz_class a){
    int iteration = 40;

    if(a < 2 || (a != 2 && a % 2 == 0)) return false;
    if(a <= 3) return true;

    mpz_class d = a - 1;
    int s = 0;
    while(d % 2 == 0){
        d /= 2;
        s++;
    }

    for(int i = 0; i < iteration; i++){
        gmp_randclass rand_gen(gmp_randinit_default); //gmp function for random number generation
        rand_gen.seed(time(nullptr) + i); // seed with current time and iteration number

        mpz_class test = rand_gen.get_z_range(a - 3) + 2; // generate random number in range [2, a-1]

        mpz_class x = mod::exp(test, d, a);
        if(x == 1 || x == a - 1)
            continue;

        bool continue_outer = false;
        for(int r = 1; r < s; r++){
            x = mod::mul(x, x, a);
            if(x == a - 1){
                continue_outer = true;
                break;
            }
        }

        if(continue_outer)
        continue;

        return false; // composite
    }

    return true; // probably prime
}
```

Figura 1: Caption

Cada iteração apresenta chance de erro $\leq 1/4$, portanto para 40 iterações, a chance de erro é $\leq 1/4^{40} \leq 10^{-24}$, mais do que suficiente para assumir que não há erro.

2.1.1 Geração de Chaves do RSA

Módulo de Matemática Computacional (Maths/):

O sistema implementa geração de chaves de 2048 bits seguindo o processo:

1. Geração de números primos p e q usando Miller-Rabin
2. Cálculo do módulo $n = p \times q$
3. Determinação do expoente público $e = 65537$, primo comumente usado
4. Cálculo do expoente privado $d = e^{-1} \bmod \phi(n)$

Geração de primos: O sistema utiliza o arquivo /dev/random de sistemas UNIX (https://en.wikipedia.org/wiki//dev/random) para gerar números verdadeiramente aleatórios de 1024 bits, garante que seu bit menos significativo seja 1 (ímpar), e testa se esse número é primo pelo teste de Miller-Rabin. Esse processo é repetido em loop até que seja encontrado um número primo.

```
11  mpz_class randPrimeGen(std::ifstream& urandom){
12      while (true){
13      const size_t num_bytes = 128; // 1024 bits
14      unsigned char buffer[num_bytes];
15
16      urandom.read(reinterpret_cast<char*>(buffer), num_bytes);
17      if (!urandom) {
18          std::cerr << "Failed to read enough bytes\n";
19          return 0;
20      }
21
22      buffer[num_bytes - 1] |= 1; // Ensure the last byte is odd to avoid even numbers
23
24      // Initialize mpz_class from binary buffer
25      mpz_class rand_num;
26      mpz_import(rand_num.get_mpz_t(), num_bytes, 1, 1, 0, 0, buffer);
27
28      if(MillerRabin(rand_num)){
29          return rand_num;
30          }
31      }
32  }
```

Figura 2: Enter Caption

Cálculo de Chaves: O módulo n, expoente público e, e expoente privado d são calculados exatamente como se espera, vide:

```cpp
int main() {
    // Open /dev/urandom once for the entire session
    std::ifstream urandom("/dev/urandom", std::ios::in | std::ios::binary);
    if (!urandom) {
        std::cerr << "Failed to open /dev/urandom\n";
        return 1;
    }

    auto start = std::chrono::high_resolution_clock::now();

    mpz_class p = randPrimeGen(urandom);
    mpz_class q = randPrimeGen(urandom);

    mpz_class n = p * q;
    mpz_class phi = (p - 1) * (q - 1);
    mpz_class e = 65537; // Common prime public exponent
    mpz_class d = mod::inv(e, phi);

    // Write the keys to PEM files
    write_pem_file(n, e, "public_key.pem", "PUBLIC");
    write_pem_file(n, d, "private_key.pem", "PRIVATE");

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Key generation took " << elapsed.count() << " seconds.\n";

    return 0;
}
```

Figura 3: Caption

Onde `mod::inv(e, phi)` calcula o inverso de e em $mod\ \phi(n)$. E a utilização da biblioteca nativa `Chronos` é utilizada para calcular o tempo necessário para geração das chaves (geralmente menor que um segundo).

Armazenamento de Chaves (PEM/Base64): As chaves geradas são armazenadas em arquivo PEM com organização personalizada definida por nós. Existem padrões PKCS para codificação de arquivo PEM, porém esses padrões são de dificílima codificação e tem como principal objetivo a universalização para utilização em diversas bibliotecas criptográficas.

Como esses dois aspectos fogem ao objetivo do trabalho, preferimos armazenar as chaves de forma similar para chaves públicas e privadas na estrutura:

4 bytes para representar os inteiros que representam o tamanho em bytes do dado, e o dado, ou seja `[tamanho` n `(4 bytes)]` $\|$ `[módulo` n`]`$\|$`tamanho chave (4 bytes)]`$\|$`[chave]`

A implementação disso também pode ser encontrada no repositório, mas para manter o foco, apresentamos exemplo dos dois arquivos:

Figura 4: Enter Caption

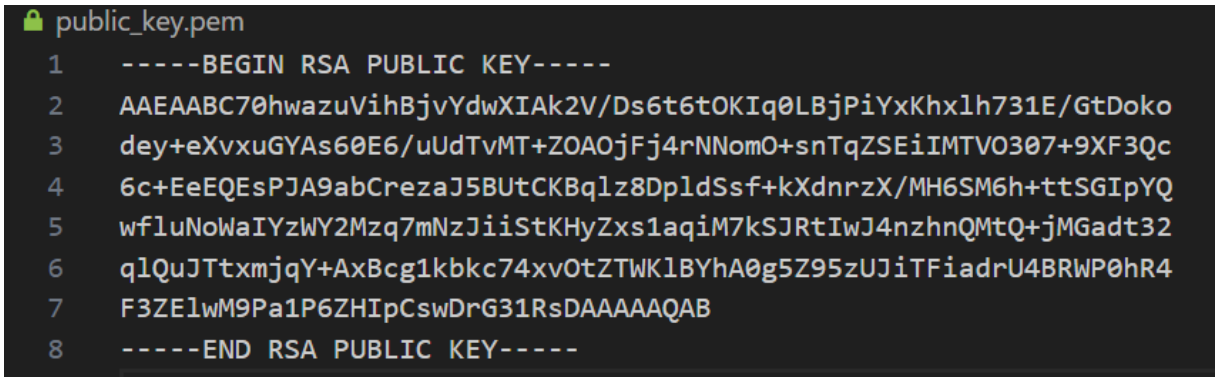

Figura 5: Enter Caption

Observação: O arquivo de chave pública é notavelmente menor porque o expoente $e = 65537$ é com certeza menor que o seu inverso d, qualquer que seja.

2.1.2 PSS

A função de PSS é realizar o preenchimento da mensagem antes da assinatura digital, de forma a garantir segurança contra ataques adaptativos e fornecer aleatoriedade ao processo. O esquema EMSA-PSS (Encoding Method for Signature with Appendix – Probabilistic Signature Scheme) é definido no padrão PKCS#1 v2.2 e é amplamente utilizado com RSA. Seu funcionamento baseia-se na utilização de uma função de hash e de uma função de geração de máscara (MGF1), além de um valor de salt aleatório. Esses elementos são combinados com a mensagem original para produzir uma codificação de comprimento fixo, que é então usada na operação de assinatura RSA. O uso do sal e da codificação probabilística torna cada assinatura única, mesmo que a mesma mensagem seja assinada várias vezes, o que proporciona resistência contra diversos tipos de ataques criptográficos.

A implementação dessa função seguiu o que foi definido em:

https://www.rfcreader.com/#rfc8017_line1391 seção 9.1, e seus passos podem ser visualizados facilmente no código:

`EMSA-PSS-ENCODE (M, emBits)`

Options:

- `Hash` hash function (hLen denotes the length in octets of the hash function output)
- `MGF` mask generation function
- `sLen` intended length in octets of the salt

Input:

- `M` message to be encoded, an octet string
- `emBits` maximal bit length of the integer OS2IP(EM), at least $8hLen + 8sLen + 9$

Output:

- `EM` encoded message, an octet string of length $emLen = \lceil \text{emBits}/8 \rceil$

Steps:

1. If the length of `M` is greater than the input limitation for the hash function ($2^{61} - 1$ octets for SHA-1), output "message too long" and stop. *(Desconsiderado pois utilizamos SHA3-256, que não requer mensagem menor que o Hash)*
2. Let `mHash = Hash(M)`, an octet string of length `hLen`.

```
3   std::vector<unsigned char> PSS_encode(const std::vector<unsigned char>& M, int emBits) {
4
5       std::vector<unsigned char> mHash = hash_bytes_sha3_256(M);
6       int hLen = mHash.size(); // hLen = 32, length of the hash
7
8       int sLen = hLen; // Assuming sLen is the same as hLen for simplicity
9
10      int emLen = (emBits + 7) / 8; // = ceiling (emBits)/8)
```

Figura 6: Caption

3. If `emLen` $<$ `hLen + sLen + 2`, output "encoding error" and stop.
4. Generate a random octet string `salt` of length `sLen`; if `sLen = 0`, then `salt` is the empty string.

```cpp
    std::ifstream urandom("/dev/urandom", std::ios::in | std::ios::binary);
    if (!urandom) {
        std::cerr << "Failed to open /dev/urandom\n";
        return {};
    }
    // Generate random salt
    std::vector<unsigned char> salt(sLen);
    urandom.read(reinterpret_cast<char*>(salt.data()), sLen);
    if (!urandom) {
        std::cerr << "Failed to read enough bytes for salt\n";
        return {};
    }
```

Figura 7: Caption

5. Let

   `M'` = `0x00 00 00 00 00 00 00 00 || mHash || salt`;

   `M'` is an octet string of length $8 + hLen + sLen$ with eight initial zero octets.

```cpp
    //empty 8 bytes padding string
    std::vector<unsigned char> emptyString(8,(unsigned char)0x00);

    // Create the message M' = empty || H(M) || salt
    std::vector<unsigned char> M_prime;
    M_prime.reserve(8 + hLen + sLen);
    M_prime.insert(M_prime.end(), emptyString.begin(), emptyString.end());
    M_prime.insert(M_prime.end(), mHash.begin(), mHash.end());
    M_prime.insert(M_prime.end(), salt.begin(), salt.end());
```

Figura 8: Caption

6. Let `H = Hash(M')`, an octet string of length `hLen`.

7. Generate an octet string `PS` consisting of `emLen - sLen - hLen - 2` zero octets. The length of `PS` may be 0.

8. Let `DB = PS || 0x01 || salt`; `DB` is an octet string of length `emLen - hLen - 1`.

```cpp
    // H = Hash(M')
    std::vector<unsigned char> H = hash_bytes_sha3_256(M_prime);

    size_t PSLen = emLen - hLen - sLen - 2; // Length of the padding string
    std::vector<unsigned char> PS(PSLen, (unsigned char)0x00);

    size_t DBLen = PSLen + 1 + sLen; // Length of the data block

    // Create the data block DB = PS || 0x01 || salt
    std::vector<unsigned char> DB;
    DB.reserve(DBLen);
    DB.insert(DB.end(), PS.begin(), PS.end());
    DB.push_back(0x01);
    DB.insert(DB.end(), salt.begin(), salt.end());
```

Figura 9: Caption

9. Let `dbMask = MGF(H, emLen - hLen - 1)`.
10. Let `maskedDB = DB` $\oplus$ `dbMask`.

```cpp
    // Create and apply the mask to the data block
    std::vector<unsigned char> dbMask = mgf1(H, DBLen);
    std::vector<unsigned char> maskedDB = xorVectors(DB, dbMask);

```

Figura 10: Caption

11. Set the leftmost $8 \cdot emLen - emBits$ bits of the leftmost octet in `maskedDB` to zero.

```cpp
    // Set the leftmost 8emLen - emBits bits of the leftmost octet in maskedDB to zero.
    for(size_t i = 0; i<(8 * emLen - emBits); i++) {
        maskedDB[i / 8] &= ~(1 << (7 - (i % 8))); // Clear the bit
    }

```

Figura 11: Caption

12. Let `EM = maskedDB || H || 0xbc`.
13. Output `EM`.

```
64      std::vector<unsigned char> EM;
65      EM.reserve(emLen);
66
67      // EM = maskedDB || H || 0xBC
68      EM.insert(EM.end(), maskedDB.begin(), maskedDB.end());
69      EM.insert(EM.end(), H.begin(), H.end());
70      EM.push_back(0xBC); // Add the 0xBC byte at the end
71
72      return EM; // Return the encoded message
73  }
```

Figura 12: Caption

A verificação PSS também pode ser encontrada no repositório e segue também os passos da referência, sendo eles:

EMSA-PSS-VERIFY (M, EM, emBits)

Options:

- `Hash` hash function (hLen denotes the length in octets of the hash function output)
- `MGF` mask generation function
- `sLen` intended length in octets of the salt

Input:

- `M` message to be verified, an octet string
- `EM` encoded message, an octet string of length `emLen` = $\lceil emBits/8 \rceil$
- `emBits` maximal bit length of the integer `OS2IP(EM)`, at least $8hLen + 8sLen + 9$

Output: "`consistent`" or "`inconsistent`"

Steps:

1. If the length of `M` is greater than the input limitation for the hash function ($2^{61} - 1$ octets for SHA-1), output "inconsistent" and stop.
2. Let `mHash = Hash(M)`, an octet string of length `hLen`.
3. If `emLen < hLen + sLen + 2`, output "inconsistent" and stop.
4. If the rightmost octet of `EM` does not have hexadecimal value `0xbc`, output "inconsistent" and stop.
5. Let `maskedDB` be the leftmost `emLen - hLen - 1` octets of `EM`, and let `H` be the next `hLen` octets.

6. If the leftmost $8emLen - emBits$ bits of the leftmost octet in `maskedDB` are not all equal to zero, output "inconsistent" and stop.

7. Let `dbMask = MGF(H, emLen - hLen - 1)`.

8. Let `DB = maskedDB` $\oplus$ `dbMask`.

9. Set the leftmost $8emLen - emBits$ bits of the leftmost octet in `DB` to zero.

10. If the `emLen - hLen - sLen - 2` leftmost octets of `DB` are not zero or if the octet at position `emLen - hLen - sLen - 1` does not have hexadecimal value `0x01`, output "inconsistent" and stop.

11. Let `salt` be the last `sLen` octets of `DB`.

12. Let `M' = 0x0000000000000000 || mHash || salt`; `M'` is an octet string of length $8 + hLen + sLen$ with eight initial zero octets.

13. Let `H' = Hash(M')`, an octet string of length `hLen`.

14. If `H = H'`, output "consistent". Otherwise, output "inconsistent".

2.1.3 Segurança e funcionamento do RSA

O RSA é um sistema criptográfico de chave pública baseado na dificuldade computacional do problema da fatoração de números inteiros grandes. Ele utiliza um par de chaves: uma pública (n, e) e uma privada (n, d), onde $n = p \cdot q$ é o produto de dois grandes primos distintos. A segurança do RSA decorre do fato de que, conhecendo apenas n, é computacionalmente inviável obter os fatores primos p e q, o que impede o cálculo da chave privada. Para encriptar uma mensagem m (representada como um inteiro tal que $0 \leq m < n$), aplica-se a operação $c = m^e \bmod n$, onde c é o texto cifrado. A decriptação ocorre utilizando a chave privada: $m = c^d \bmod n$, recuperando assim a mensagem original. O valor de d é escolhido de forma que satisfaça $d \cdot e \equiv 1 \pmod{\varphi(n)}$, onde $\varphi(n) = (p-1)(q-1)$.

2.1.4 Assinatura e verificação com RSA-PSS

A assinatura digital RSA-PSS (Probabilistic Signature Scheme) é um esquema de assinatura baseado no RSA que utiliza uma codificação probabilística para aumentar a segurança contra ataques adaptativos. Para assinar uma mensagem M, o emissor aplica uma função de codificação (como EMSA-PSS) que incorpora um sal aleatório e produz um valor codificado EM, garantindo que cada assinatura seja única, mesmo para mensagens iguais. Esse valor EM é então convertido para um inteiro m e a assinatura é calculada como $s = m^d \bmod n$, utilizando a chave privada. Para verificar a assinatura, o receptor calcula $m' = s^e \bmod n$ usando a chave pública e converte esse inteiro de volta para um valor codificado EM'. Em seguida, aplica-se o processo inverso da codificação para verificar se EM' corresponde corretamente à mensagem original M. A segurança do RSA-PSS é baseada no modelo aleatório do oráculo, provendo resistência a ataques do tipo "chosen-message".

2.2 Contribuições Técnicas

O trabalho apresenta as seguintes contribuições alinhadas às especificações:

- **Implementação educacional completa**: Código didático seguindo especificações rigorosas
- **Primitivas próprias**: Implementação de todas as primitivas requeridas
- **Segurança robusta**: Uso de algoritmos modernos (SHA-3, Miller-Rabin, PSS)
- **Compatibilidade**: Ambiente Unix/Linux conforme especificações

2.3 Aplicações Práticas

O sistema desenvolvido pode ser aplicado em diversos cenários:

- Autenticação de documentos digitais
- Verificação de integridade de software
- Sistemas de comunicação segura
- Contratos digitais e blockchain
- Sistemas de votação eletrônica

2.4 Limitações e Trabalhos Futuros

Algumas limitações identificadas e sugestões para trabalhos futuros:

2.4.1 Limitações Atuais

- Dependência de ambiente Unix para geração de entropia
- Tamanho fixo de chave (2048 bits)
- Falta de interface gráfica
- Ausência de certificação formal

2.4.2 Possíveis melhorias

- **Suporte multiplataforma**: Adaptação para Windows e macOS
- **Tamanhos variáveis de chave**: Suporte para 3072 e 4096 bits
- **Interface gráfica**: Desenvolvimento de GUI para usuários finais
- **Integração com HSM**: Suporte para módulos de segurança de hardware
- **Certificados digitais**: Implementação de infraestrutura PKI
- **Algoritmos pós-quânticos**: Preparação para era da computação quântica

2.5 Considerações Finais

Este trabalho demonstra com sucesso a implementação completa de um gerador/verificador de assinaturas RSA-PSS conforme todas as especificações estabelecidas no roteiro de programação. O sistema desenvolvido atende integralmente aos requisitos das três partes especificadas, implementando todas as primitivas criptográficas necessárias sem utilização de bibliotecas proibidas.

A implementação em C++ seguiu rigorosamente as diretrizes técnicas, utilizando apenas bibliotecas permitidas (GMP para aritmética modular) e implementando de forma própria todas as primitivas criptográficas essenciais: teste de primalidade Miller-Rabin, cifração/decifração RSA, e formatação/parsing personalizado.

O projeto não apenas cumpre os objetivos técnicos propostos, mas também demonstra compreensão profunda dos fundamentos da criptografia assimétrica e assinatura digital. A estrutura modular desenvolvida facilita a compreensão dos algoritmos implementados e serve como excelente ferramenta educacional.

Os resultados obtidos comprovam que o sistema é capaz de gerar chaves seguras, assinar arquivos digitalmente e verificar assinaturas com alta confiabilidade, atendendo plenamente aos requisitos de um trabalho de implementação criptográfica acadêmica rigorosa.