Assinatura RSA

Matheus Cruz
Raques da Silva

November 2023

1 Introdução

A assinatura RSA, ou algoritmo de assinatura de chave pública RSA, é um método criptográfico amplamente utilizado para garantir a autenticidade e integridade de informações digitais. O RSA é baseado na complexidade de fatores de números primos grandes. A assinatura RSA funciona da seguinte forma: primeiro, um par de chaves é gerado, composto por uma chave privada e uma chave pública. A chave privada é mantida em segredo, enquanto a chave pública é distribuída livremente. Qualquer pessoa pode criptografar uma mensagem usando a chave pública, mas apenas o detentor da chave privada correspondente pode decifrá-la. No contexto de assinatura, o processo é invertido. O detentor da chave privada pode criar uma assinatura digital para uma mensagem específica. Essa assinatura é única para a mensagem e verifica que a mensagem não foi alterada desde a assinatura. A verificação da assinatura é realizada com a chave pública correspondente.

2 Geração de chaves

Antes de implementar o algoritmo, é necessário uma forma de produzir as chaves pública e privada, para isso, primeiro encontramos dois numeros primos de 1024 bits p e q para gerar n tal que $n = p \times q$. Para isso, usamos o teste de primalidade de Miller-Rabin.

```
int Miller_Rabin_test(mpz_t prime_canditate){
    // n-1 = 2^k*r
    if (mpz_cmp_ui(prime_canditate, 2) == 0 | mpz_cmp_ui(prime_canditate, 3) == 0){
        return true;
    }
    mpz_t r, results, p_sub1;
    mpz_inits(results, p_sub1, NULL);
    mpz_sub_ui(p_sub1, prime_canditate, 1); // p-1
    mpz_init_set(r,p_sub1);
    int k = 0; // potencia de 2
    mpz_mod_ui(results, p_sub1, 2);
    while (mpz_cmp_ui(results, 0) == 0) { // while r mod 2 != 0
        k++;
        mpz_div_ui(r, r, 2);
        mpz_mod_ui(results, r, 2);
    }  // enquanto o numero for divisivel por 2
    // aqui temos k e r(p_sub1)
    u8 a_gen[128];
    mpz_t a;
    mpz_init(a);
    for (int i = 0; i <40; ++i) {
        randombytes( buf: a_gen, buf_len: 128);
        mpz_import(a,128,1,1,0,0,a_gen);
        mpz_sub_ui (results,prime_canditate,3); // a entre 2 e p-2 -> (a mod p-3) + 2
        mpz_mod(a,a,results);
        mpz_add_ui(a,a,2);  // a entre 2 e p-2
        mpz_powm(results,a,r,prime_canditate);  // a^r mod p
        if (mpz_cmp_ui(results,1) == 0) return 1; // a^p-1 mod n == 1
    // testar para 0 a n-1
        for (u64 j = 0; j < k; ++j) {
            // a ^ (2^j) * r
            mpz_ui_pow_ui(results, (u64)2, j);          // results = 2^j
            mpz_mul(results,results,r);                 //  results = 2^j * r
            mpz_powm(results,a,results,prime_canditate); // (a^(2^j * r) ) mod p
            if (mpz_cmp(results, p_sub1) == 0){
                break;
            }
            /*
             * se chegou ate o ultimo j sem o break ent
             * nao e congruente para nenhum valor de j
             * logo e um numero composto
             */
            if (j == k - 1) {
                mpz_clear(results);
                mpz_clear(p_sub1);
                mpz_clear(r);
                mpz_clear(a);
                return false;
            }
        }
    }

    //  se o numero nao der como composto em nenhum teste
    //  provavelmente primo
    mpz_clear(results);
    mpz_clear(p_sub1);
    mpz_clear(r);
    mpz_clear(a);
    return true;
}
```

Figure 1: Teste de primalidade

Agora que podemos testar a primalidade, geramos numeros de 1024 bits, ate que passem no teste.

```
void gen_prime(mpz_t prime) {
    u8 gen_buffer[128];
    while(true){
        randombytes( buf: gen_buffer,  buf_len: 128);
        gen_buffer[127] |= 1; // garante que e um valor impar

        mpz_import(prime,128,1,1,0,0,gen_buffer);

        if (Miller_Rabin_test( prime_canditate: prime)) break;
    }
} // gera um numero provavelmente primo
```

Figure 2: Geração dos números de 1024 bits

Agora encontramos dois numeros e e d tal que dada uma mensagem M: $((m^e)^d) \equiv m \pmod{n} \wedge ((m^d)^e) \equiv m \pmod{n}$
Por questao de padronizacao, sempre usaremos $e = 65537$, que sera a chave publica, assim, geramos as chaves publica (e, n) e privada (d, n):

```
void RSA_init_keys(mpz_t e, mpz_t d, mpz_t n){
    mpz_t p,q,phi_n;
    mpz_inits(p,q,phi_n,NULL);
    gen_prime( prime: p);
    gen_prime( prime: q);

    mpz_mul(n,p,q);
    // para achar os fatores coprimos

    mpz_sub_ui(p,p,1);
    mpz_sub_ui(q,q,1);

    mpz_mul(phi_n,p,q); // numero de coprimos de n

    mpz_set_ui(e, 65537); // valor padrao

    mpz_invert(d,e,phi_n);

    if(mpz_cmp(d,e) == 0) mpz_add(d,d,phi_n); // chave publica nao pode ser igual a privada



    mpz_clear(phi_n);
}
```

Figure 3: Geração de chave pública e privada

3 Hasing

Durante o código, usamos o padrao de hash SHA-3, para isso, temos 2 funções auxiliares, uma para gerar o hash de um arquivo, e outra que gera o hash de um array:

```c
void sha3(u8 *message, u8 *digest){
    const EVP_MD *md = EVP_sha3_512();
    unsigned int hash_len;
    EVP_Digest( data: message,  count: sizeof(message),  md: digest,  size: &hash_len,  type: md,  impl: NULL);
}
```

Figure 4: Hash de array

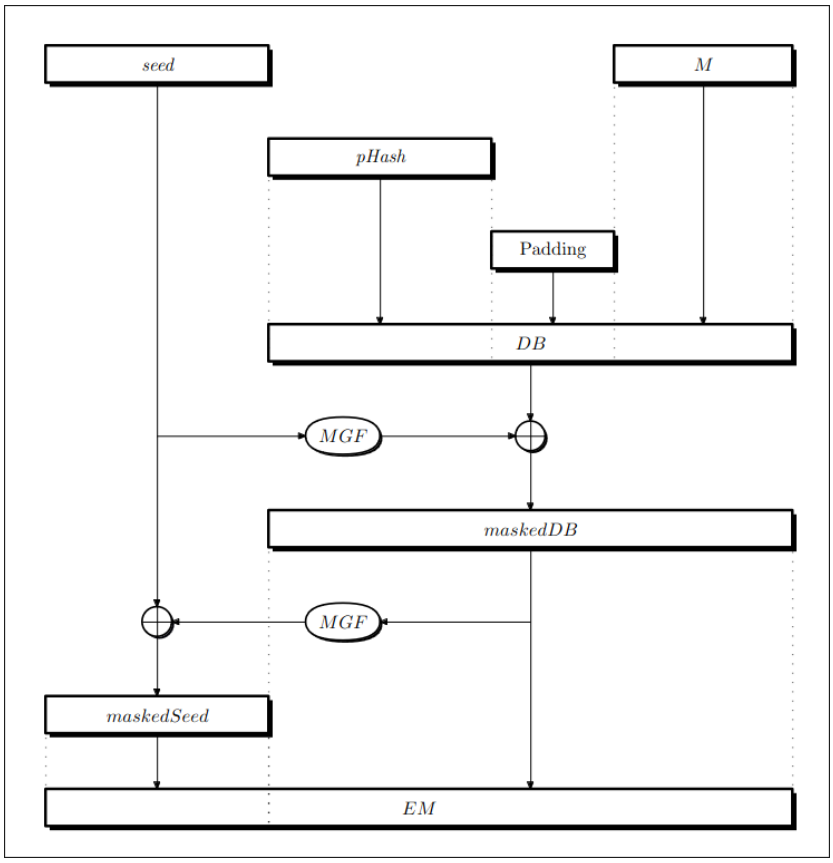
```c
void sha3_file(char *file_path, u8 *digest){
    EVP_MD_CTX *mdctx;
    const EVP_MD *md;
    unsigned char buffer[BUFSIZE];
    size_t bytes_read;
    FILE *file;

    md = EVP_sha3_512();
    file = fopen( filename: file_path,  modes: "rb");

    if(!file) {
        fprintf( stream: stderr,  format: "Error opening file\n");
        return;
    }
    mdctx = EVP_MD_CTX_new();
    EVP_DigestInit_ex( ctx: mdctx,  type: md,  impl: NULL);
    while((bytes_read = fread( ptr: buffer,  size: 1,  n: BUFSIZE,  stream: file)) != 0)
        EVP_DigestUpdate( ctx: mdctx,  d: buffer,  cnt: bytes_read);

    EVP_DigestFinal_ex( ctx: mdctx,  md: digest,  s: NULL);
    EVP_MD_CTX_free( ctx: mdctx);

    fclose( stream: file);
}
```

Figure 5: Hash de arquivo

4 OAEP

4.1 Encoding

O OAEP(Optimal asymmetric encryption padding) é um sistema de padding tipicamente usado junto do RSA. O OAEP adiciona uma certa aleatoriodade ao RSA, além de fornecer uma estrutura verificável durante a operação de decode. O encode OAEP segue o seguinte esquema:



E implementamos da seguinte forma:

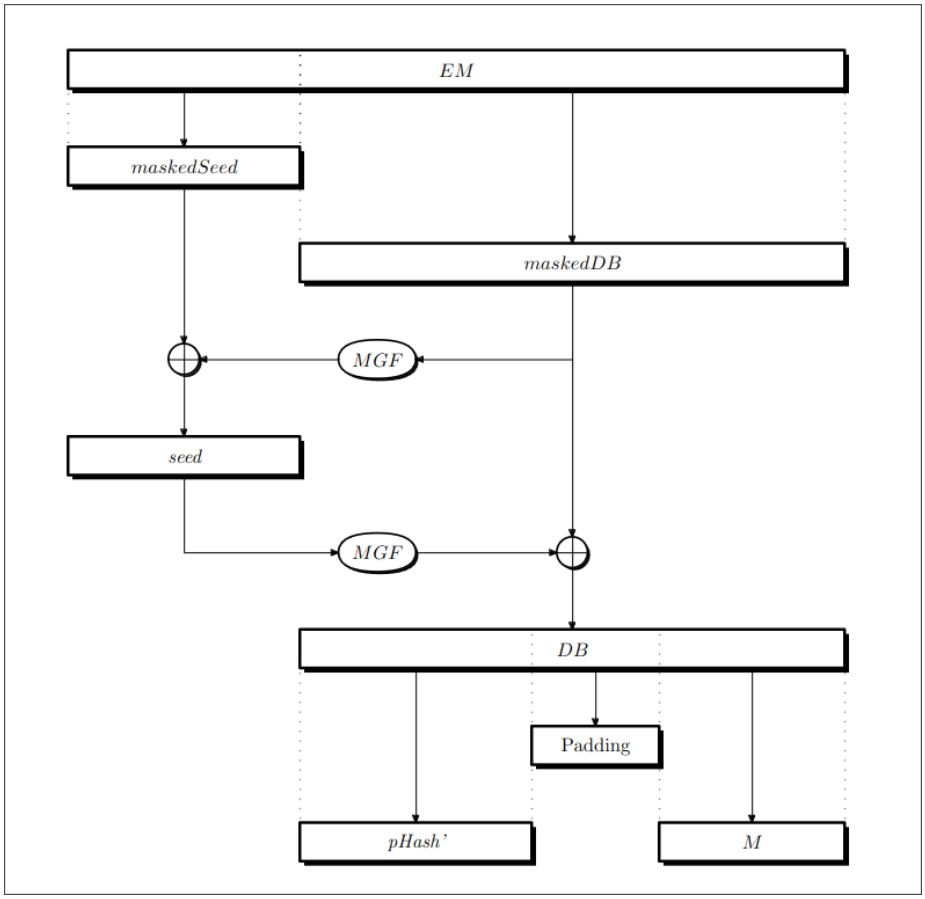
```
void OAEP_encode(const char* M, char* EM, u8* Parameter, int emLen){
    size_t mLen = strlen( s: M);
    const size_t hash_len = 64; //sha-3 512
    size_t PS_len = emLen - (mLen + 2*hash_len + 1);
    // P pode ser de qualquer tamanho ja q sha-3 n tem problema com tamanho de input
    if (mLen > (emLen - 2*hash_len - 1) || PS_len <= 0 ) {
        printf( format: "mensagem muito longa\n");
        return;
    }
    u8 PS[PS_len]; memset( s: PS, c: 0, n: PS_len); //padding
    u8 Phash[hash_len]; //hash do parametro de seguranca
    sha3( message: Parameter, digest: Phash);
    u8 DB[hash_len + PS_len + 1 + mLen]; // phash || PS ||01 || M
    memcpy( dest: DB, src: Phash, n: hash_len);
    memcpy( dest: DB + hash_len, src: PS, n: PS_len);
    DB[hash_len+PS_len] = 0x01; // separador do padding
    memcpy( dest: DB+hash_len+PS_len+1, src: M, n: mLen);

    u8 seed[hash_len]; randombytes( buf: seed, buf_len: hash_len); //gera seed
    //gera DB = DB masked
    u8 dbMask[emLen - hash_len];
    MGF(seed, mask: dbMask, len: emLen - hash_len, seedLen: hash_len); // Db mask
    for (int i = 0; i < (emLen - hash_len); ++i) DB[i]^=dbMask[i]; // DB = masked DB
    u8 seedMask[hash_len]; MGF( seed: DB, mask: seedMask, len: hash_len, seedLen: hash_len); // seed mask
    for (int i = 0; i < hash_len; ++i) seed[i] ^= seedMask[i]; // seed = masked seed
    // EM = masked seed || masked DB
    memcpy( dest: EM, src: seed, n: hash_len);
    memcpy( dest: EM+hash_len, src: DB, n: emLen - hash_len);

}
```

Agora que temos o OAEP, usamos o RSA para criptografar essa mensagem:

```
u8* RSA_OAEP_encrypt(char *message ,mpz_t n, mpz_t e, u8* parameter, size_t *count){
    mpz_t c;
    mpz_init(c);
    u8 enc_message[224];
    OAEP_encode( M: message, EM: enc_message, parameter, emLen: 224);
    mpz_import(c,224,1,1,0,0,enc_message);
    mpz_powm(c,c,e,n);
    return (u8 *) mpz_export(NULL,count, 1,1,0,0,c);

}
```

4.2 Decoding

O decode do OAEP segue o seguinte esquema:



E implementamos da seguinte forma:

```
u8* OAEP_decode(const char *EM,  u8 *Parameter){
    size_t emLen = 224;
    const size_t hash_len = 64; //sha-3 512
    size_t db_len = emLen-hash_len;
    if (emLen <= hash_len + 1) printf( format: "decoding error: size\n");
    u8 maskedSeed[hash_len], maskedDB[db_len];
    memcpy( dest: maskedSeed, src: EM,  n: hash_len);
    memcpy( dest: maskedDB, src: EM+hash_len,  n: emLen-hash_len);
    u8 seedMask[hash_len], dbMask[db_len];
    MGF( seed: maskedDB, mask: seedMask, len: hash_len,  seedLen: db_len);
    for (int i = 0; i < hash_len; ++i) maskedSeed[i] ^= seedMask[i]; // restaura a seed
    // restaura DB
    MGF( seed: maskedSeed,  mask: dbMask, len: db_len,  seedLen: hash_len);
    for (int i = 0; i < db_len; ++i) maskedDB[i] ^= dbMask[i];     // restaura DB
    u8 Phash[hash_len]; sha3( message: Parameter,  digest: Phash);
    // DB PARTS phash || PS ||01 || M
    // checa de se o Phash ta igual
    for (int i = 0; i < hash_len; ++i) {
        if (maskedDB[i] != Phash[i] ){
            printf( format: "decoding error: Phash\n ");
            return NULL;
        }
    }  //mesmo phash, agr e pra achar o comeco da msg
    int m_start = 0;
    for (int i = hash_len; i < db_len; ++i) {
        if (maskedDB[i] == 0x01) {
            m_start = i+1;
            break;
        }
        else if(maskedDB[i] != 0x00){
            printf( format: "decoding error: PS\n");
            return NULL;
        }
    }
    size_t mLen = db_len - m_start;
    u8 *message = malloc( size: mLen);
    memcpy( dest: message,  src: maskedDB+m_start,  n: mLen);
    return message;
}
```

Como estamos usando RSA para criptografar a mensagem, primeiro precisamos decifrar usando RSA para podermos aplicar o decode do OAEP.

```
u8* RSA_OAEP_decrypt(u8 *ciphertext, mpz_t n, mpz_t d, u8* Parameter, size_t count_cipher){
    mpz_t c;
    mpz_init(c);

    mpz_import(c,count_cipher, 1, 1, 0, 0, ciphertext);
    mpz_powm(c,c,d,n);

    size_t count;
    u8 *message = (u8 *) mpz_export(NULL, &count, 1,1,0,0,c);

// export do M para message
    return OAEP_decode( EM: message, Parameter);

}
```

4.3 Geração de mascara

A funcao MGF usada para geracao de mascaras, utiliza do hash SHA-3 e gera uma mascara do tamanho desejado:

```
void MGF(u8* seed, u8* mask,int len, int seedLen){

    u8 seed_concat[seedLen + 4];
    u8 digest[64];
    memcpy( dest: seed_concat, src: seed,  n: seedLen);

    int i,j;
    for ( i = 0; i < ceil( x: len/64); ++i) {
        for (j = 0; j < 4; j++) {
            seed_concat[seedLen + j] = (i >> ((3 - j) * 8)) & 0xFF;
        }
        sha3( message: seed_concat,digest);
        memcpy( dest: mask + (i*64), src: digest,  n: 64);
    }
    if (i*64 != len) {
        for (j = 0; j < 4; j++) {
            seed_concat[seedLen + j] = (i >> ((3 - j) * 8)) & 0xFF;
        }
        sha3( message: seed_concat,digest);
        memcpy( dest: mask + (i*64), src: digest,  n: len-i*64);
    }

}
```

5 BASEv64

5.1 Encode

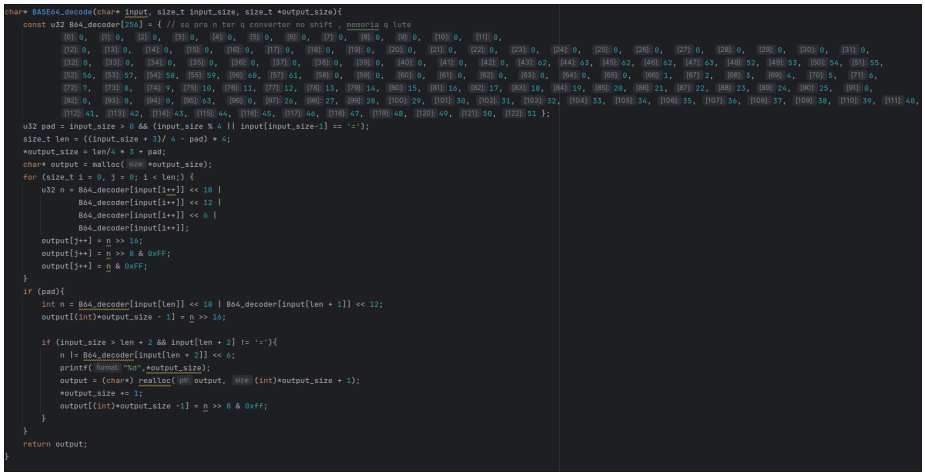
A codificação Base64 é um método comum utilizado para representar dados binários de forma textual, especialmente em ambientes que suportam apenas caracteres ASCII. No processo de codificação Base64, os dados binários são convertidos em uma sequência de caracteres ASCII utilizando um conjunto específico de 64 caracteres, incluindo letras maiúsculas e minúsculas, números e caracteres especiais. Cada conjunto de três bytes de dados binários é dividido em quatro conjuntos de seis bits e, em seguida, mapeado para o caractere correspondente na tabela Base64:

Nr	Character	Nr	Character	Nr	Character	Nr	Character
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

```c
char* BASE64_encode(char* input, size_t input_size, size_t *output_size){
    static char encoding_table[] = {
            [0]: 'A', [1]: 'B', [2]: 'C', [3]: 'D', [4]: 'E', [5]: 'F', [6]: 'G', [7]: 'H',
            [8]: 'I', [9]: 'J', [10]: 'K', [11]: 'L', [12]: 'M', [13]: 'N', [14]: 'O', [15]: 'P',
            [16]: 'Q', [17]: 'R', [18]: 'S', [19]: 'T', [20]: 'U', [21]: 'V', [22]: 'W', [23]: 'X',
            [24]: 'Y', [25]: 'Z', [26]: 'a', [27]: 'b', [28]: 'c', [29]: 'd', [30]: 'e', [31]: 'f',
            [32]: 'g', [33]: 'h', [34]: 'i', [35]: 'j', [36]: 'k', [37]: 'l', [38]: 'm', [39]: 'n',
            [40]: 'o', [41]: 'p', [42]: 'q', [43]: 'r', [44]: 's', [45]: 't', [46]: 'u', [47]: 'v',
            [48]: 'w', [49]: 'x', [50]: 'y', [51]: 'z', [52]: '0', [53]: '1', [54]: '2', [55]: '3',
            [56]: '4', [57]: '5', [58]: '6', [59]: '7', [60]: '8', [61]: '9', [62]: '+', [63]: '/'};

    static int reboco[] = { [0]: 0, [1]: 2, [2]: 1};

    *output_size = (input_size + 2)/3 *4;
    char *output = malloc( size: *output_size);
    u32 a,b,c;
    for (size_t i = 0, j = 0; i < input_size;) {
        a = i < input_size ? (u8)input[i++] : 0;
        b = i < input_size ? (u8)input[i++] : 0;
        c = i < input_size ? (u8)input[i++] : 0;

        u32 tripla = a << 16 | b << 8 | c;

        output[j++] = encoding_table[(tripla >> 3 * 6) & 0x3F];
        output[j++] = encoding_table[(tripla >> 2 * 6) & 0x3F];
        output[j++] = encoding_table[(tripla >> 1 * 6) & 0x3F];
        output[j++] = encoding_table[(tripla >> 0 * 6) & 0x3F];
    }
    for (int i = 0; i < reboco[input_size % 3]; i++)
        output[*output_size - 1 - i] = '=';
    return output;
}
```

5.2 Decode

A decodificação Base 64, por sua vez, reverte o processo, convertendo a sequência de caracteres Base 64 de volta para os dados binários originais.



```
char* BASE64_decode(char* input, size_t input_size, size_t *output_size){
    const u32 B64_decoder[256] = { // so pra n ter q converter no shift , memoria q lute
            [0] 0,   [1] 0,   [2] 0,   [3] 0,   [4] 0,   [5] 0,   [6] 0,   [7] 0,   [8] 0,   [9] 0,   [10] 0,   [11] 0,
            [12] 0,   [13] 0,   [14] 0,   [15] 0,   [16] 0,   [17] 0,   [18] 0,   [19] 0,   [20] 0,   [21] 0,   [22] 0,   [23] 0,   [24] 0,   [25] 0,   [26] 0,   [27] 0,   [28] 0,   [29] 0,   [30] 0,   [31] 0,
            [32] 0,   [33] 0,   [34] 0,   [35] 0,   [36] 0,   [37] 0,   [38] 0,   [39] 0,   [40] 0,   [41] 0,   [42] 0,   [43] 62,   [44] 63,   [45] 62,   [46] 62,   [47] 63,   [48] 52,   [49] 53,   [50] 54,   [51] 55,
            [52] 56,   [53] 57,   [54] 58,   [55] 59,   [56] 60,   [57] 61,   [58] 0,   [59] 0,   [60] 0,   [61] 0,   [62] 0,   [63] 0,   [64] 0,   [65] 0,   [66] 1,   [67] 2,   [68] 3,   [69] 4,   [70] 5,   [71] 6,
            [72] 7,   [73] 8,   [74] 9,   [75] 10,   [76] 11,   [77] 12,   [78] 13,   [79] 14,   [80] 15,   [81] 16,   [82] 17,   [83] 18,   [84] 19,   [85] 20,   [86] 21,   [87] 22,   [88] 23,   [89] 24,   [90] 25,   [91] 0,
            [92] 0,   [93] 0,   [94] 0,   [95] 63,   [96] 0,   [97] 26,   [98] 27,   [99] 28,   [100] 29,   [101] 30,   [102] 31,   [103] 32,   [104] 33,   [105] 34,   [106] 35,   [107] 36,   [108] 37,   [109] 38,   [110] 39,   [111] 40,
            [112] 41,   [113] 42,   [114] 43,   [115] 44,   [116] 45,   [117] 46,   [118] 47,   [119] 48,   [120] 49,   [121] 50,   [122] 51 };
    u32 pad = input_size > 0 && (input_size % 4 || input[input_size-1] == '=');
    size_t len = ((input_size + 3)/ 4 - pad) * 4;
    *output_size = len/4 * 3 + pad;
    char* output = malloc( size *output_size);
    for (size_t i = 0, j = 0; i < len;) {
        u32 n = B64_decoder[input[i++]] << 18 |
                B64_decoder[input[i++]] << 12 |
                B64_decoder[input[i++]] << 6 |
                B64_decoder[input[i++]];
        output[j++] = n >> 16;
        output[j++] = n >> 8 & 0xFF;
        output[j++] = n & 0xFF;
    }
    if (pad){
        int n = B64_decoder[input[len]] << 18 | B64_decoder[input[len + 1]] << 12;
        output[(int)*output_size - 1] = n >> 16;

        if (input_size > len + 2 && input[len + 2] != '='){
            n |= B64_decoder[input[len + 2]] << 6;
            printf( format: "%d",*output_size);
            output = (char*) realloc( ptr: output,  size: (int)*output_size + 1);
            *output_size += 1;
            output[(int)*output_size -1] = n >> 8 & 0xff;
        }
    }
    return output;
}
```

6 Assinatura e verificação RSA

6.1 Assinatura

Como dito na introdução, a assinatura é realizada usando a chave privada, a ideia é criptografar o hash do arquivo com OAEP-RSA, que então é codificado no padrão Base 64:

```
u8* RSA_sign(char *file_path, mpz_t d, mpz_t n){
    mpz_t t;
    mpz_init(t);
    u8 digest[64];
    hash_file_sha3(file_path, digest);
    mpz_import(t,64,1,1,0,0,digest);
    // n sei oq seria o parametro ent na duvida vai uns 0
    u8 parameter[64];
    memset( s: parameter,  c: 0,  n: 64);
    size_t ciphertext_len;
    u8* ciphertext = RSA_OAEP_encrypt( message: digest,n, e: d,parameter,  count: &ciphertext_len);
    u8 *full_msg = malloc( size: sizeof(ciphertext_len) + ciphertext_len);
    memcpy( dest: full_msg,  src: &ciphertext_len,  n: sizeof(ciphertext_len));
    memcpy( dest: full_msg+sizeof(ciphertext_len),  src: ciphertext,  n: ciphertext_len);
    size_t message_size;
    char *encoded_message = BASE64_encode( input: full_msg,  input_size: ciphertext_len+ sizeof(ciphertext_len),  output_size: &message_size);
    u8* full_encoded = malloc( size: message_size + sizeof(message_size));
    memcpy( dest: full_encoded,  src: &message_size,  n: sizeof(message_size));
    memcpy( dest: full_encoded+ sizeof(message_size),  src: encoded_message,  n: message_size);
    return full_encoded;

}
```

6.2 Verificação

Para verficar a assinatura, a mesma é decodificada usando a chave pública, e após recalcular o hash do arquivo, os resultados são comparados, e caso bata com o hash a assinatura é verificada.

```
int RSA_verify(char* file_path,u8* full_msg, mpz_t n, mpz_t e){
    size_t message_len;
    memcpy( dest: &message_len,  src: full_msg,  n: sizeof(message_len));
    char encoded_ciphertext[message_len];
    memcpy( dest: encoded_ciphertext,  src: full_msg + sizeof(message_len),  n: message_len);
    char* decoded_ciphertext = BASE64_decode( input: encoded_ciphertext,  input_size: message_len,  output_size: & message_len);
    size_t ciphertext_len;
    memcpy( dest: &ciphertext_len,  src: decoded_ciphertext,  n: sizeof(ciphertext_len));
    u8 ciphertext[ciphertext_len];
    memcpy( dest: ciphertext,  src: decoded_ciphertext + sizeof(ciphertext_len),  n: ciphertext_len);
    u8 Parameter[64];
    memset( s: Parameter,  c: 0,  n: 64);
    u8 *received_hash = RSA_OAEP_decrypt(ciphertext, n,  d: e, Parameter,  count_cipher: ciphertext_len);
    u8 hash[64];
    hash_file_sha3(file_path,  digest: hash);
    for (int i = 0; i < 64; ++i) {
        if (hash[i] != received_hash[i]){
            printf( format: "tem uma cobra na minha bota!\n");
            free( ptr: received_hash);
            return false;
        }
    }
    free( ptr: received_hash);
    return true;
}
```