DRSA
Relatório

CA-DRSA

Telmo Sauce (104428) Ricardo Covelo (102668)

DRSA

CA-DRSA

DETI

Universidade de Aveiro

Telmo Sauce (104428) Ricardo Covelo (102668)

telmobelasauce@ua.pt, ricardocovelo11@ua.pt

21/12/2023

Index

Chapter 1

Introduction

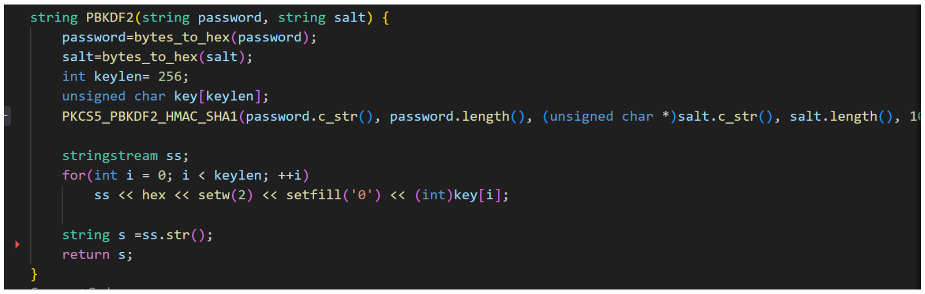
In this report, we will show how the DRSA project was built, and the reasons for certain choices along the making of the project. We chose to build the projects in C++ and Python.

1.1 Pseudo-Random Generator

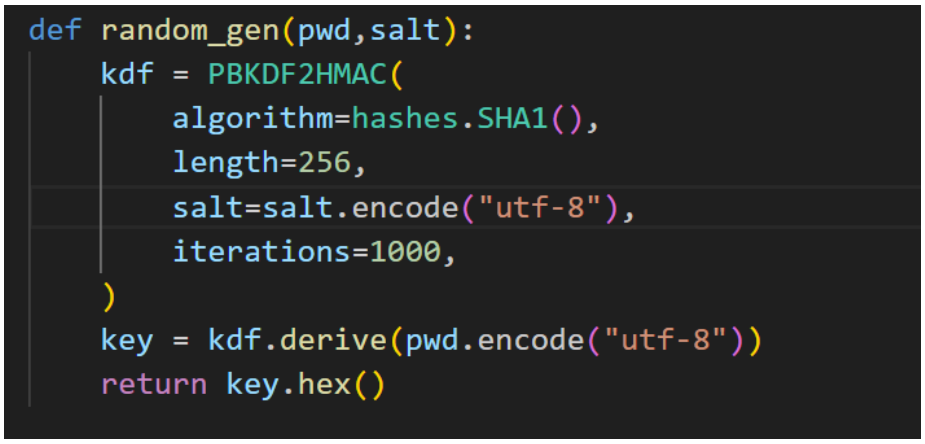
In this function we had a lot of trouble with making the C++ and Python codes generate the same output, many libraries have different implementations of the same algorithms, so we needed to test if the results of the two Hash/cryptographic functions were the same. Also, as many functions with the same purpose in the different languages required different input types, so, the addition of some functions and alterations to variables along the code were needed. this is the case of the ".Encode()", ".Hex()" and "decode()" functions in Python and the "bytes_to_Hex()" function in C++

1.1.1 PBKDF2

For the generation of the seed, we used the PBKDF2 function, we chose a length of 256 and 1000 iterations since that was recommended by the Cryptodome Website.



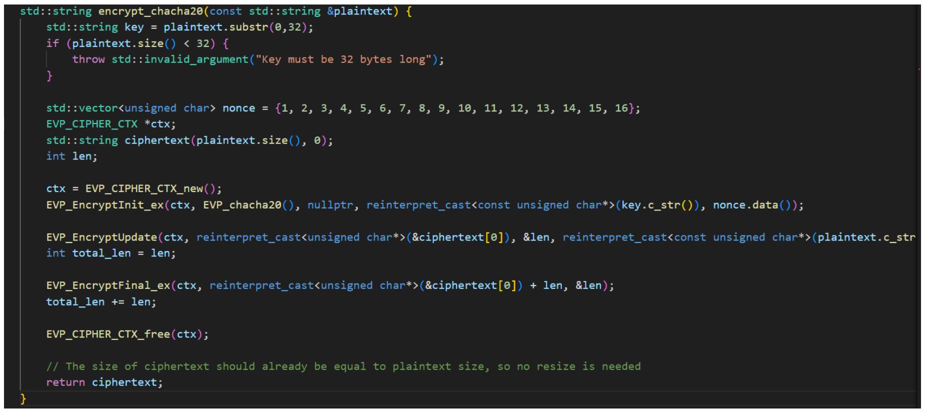
```
string PBKDF2(string password, string salt) {
    password=bytes_to_hex(password);
    salt=bytes_to_hex(salt);
    int keylen= 256;
    unsigned char key[keylen];
    PKCS5_PBKDF2_HMAC_SHA1(password.c_str(), password.length(), (unsigned char *)salt.c_str(), salt.length(), 1

    stringstream ss;
    for(int i = 0; i < keylen; ++i)
        ss << hex << setw(2) << setfill('0') << (int)key[i];

    string s =ss.str();
    return s;
}
```



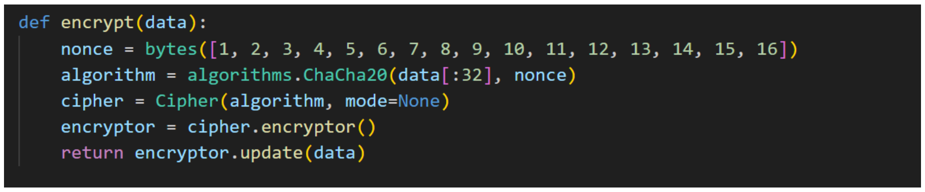
```python
def random_gen(pwd,salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA1(),
        length=256,
        salt=salt.encode("utf-8"),
        iterations=1000,
    )
    key = kdf.derive(pwd.encode("utf-8"))
    return key.hex()
```

1.1.2 Cryptographic algorithm

We used the Stream Cipher "ChaCha20" as it was the first we could implement in both programming languages and had the same result. To achieve this we needed to use a static hard-coded nonce, or the output would differ between languages. As a key, we tried to have a static one but encrypting an already encrypted text would return the original text resulting in an infinite loop, we instead chose to use the first 32 bytes of the data to be encrypted as a key.

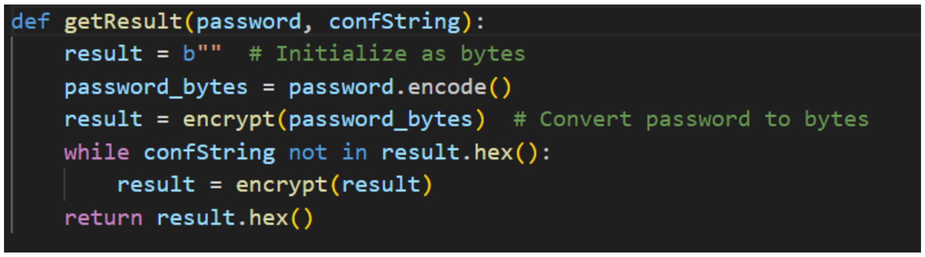


```cpp
std::string encrypt_chacha20(const std::string &plaintext) {
    std::string key = plaintext.substr(0,32);
    if (plaintext.size() < 32) {
        throw std::invalid_argument("Key must be 32 bytes long");
    }

    std::vector<unsigned char> nonce = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
    EVP_CIPHER_CTX *ctx;
    std::string ciphertext(plaintext.size(), 0);
    int len;

    ctx = EVP_CIPHER_CTX_new();
    EVP_EncryptInit_ex(ctx, EVP_chacha20(), nullptr, reinterpret_cast<const unsigned char*>(key.c_str()), nonce.data());

    EVP_EncryptUpdate(ctx, reinterpret_cast<unsigned char*>(&ciphertext[0]), &len, reinterpret_cast<const unsigned char*>(plaintext.c_str
    int total_len = len;

    EVP_EncryptFinal_ex(ctx, reinterpret_cast<unsigned char*>(&ciphertext[0]) + len, &len);
    total_len += len;

    EVP_CIPHER_CTX_free(ctx);

    // The size of ciphertext should already be equal to plaintext size, so no resize is needed
    return ciphertext;
}
```

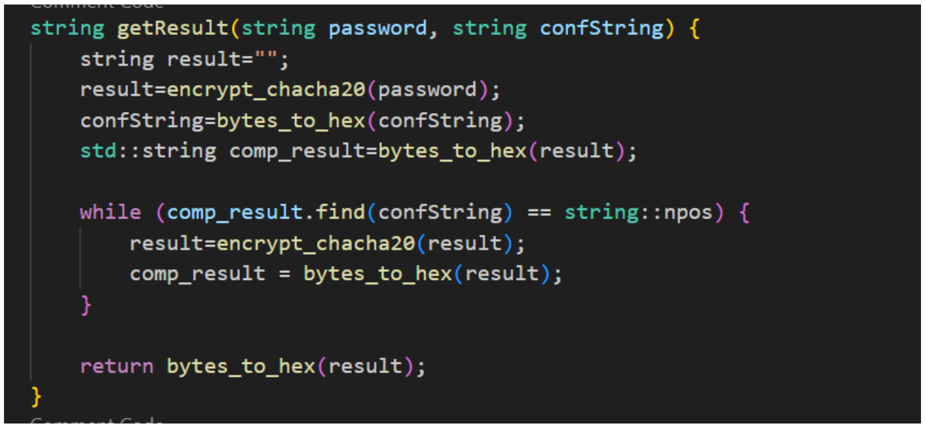


```python
def encrypt(data):
    nonce = bytes([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
    algorithm = algorithms.ChaCha20(data[:32], nonce)
    cipher = Cipher(algorithm, mode=None)
    encryptor = cipher.encryptor()
    return encryptor.update(data)
```

We used this algorithm until the result produced would have the confusion String in it.



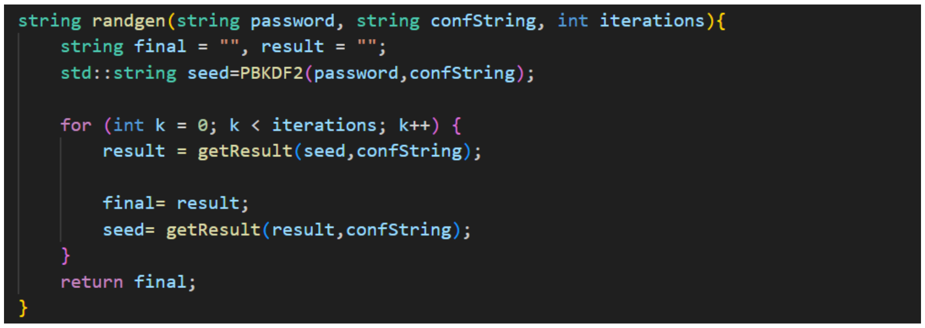
```python
def getResult(password, confString):
    result = b""  # Initialize as bytes
    password_bytes = password.encode()
    result = encrypt(password_bytes)  # Convert password to bytes
    while confString not in result.hex():
        result = encrypt(result)
    return result.hex()
```



```cpp
string getResult(string password, string confString) {
    string result="";
    result=encrypt_chacha20(password);
    confString=bytes_to_hex(confString);
    std::string comp_result=bytes_to_hex(result);

    while (comp_result.find(confString) == string::npos) {
        result=encrypt_chacha20(result);
        comp_result = bytes_to_hex(result);
    }

    return bytes_to_hex(result);
}
```

1.1.3 Random Generation Algorithm

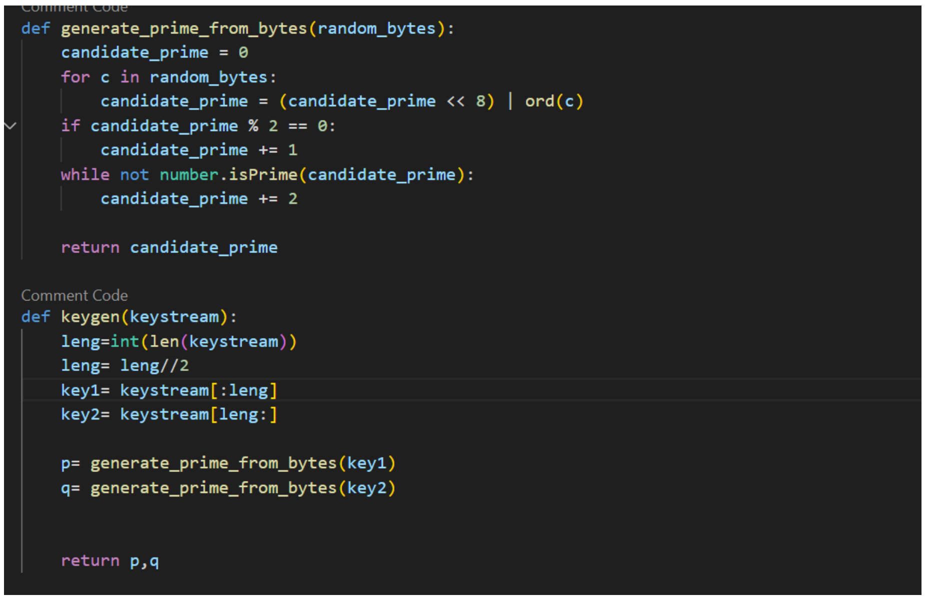
We start by generating a seed using PBKDF2 and, for each iteration, we calculate the ciphertext of that seed then calculate a new seed, and so on.



```cpp
string randgen(string password, string confString, int iterations){
    string final = "", result = "";
    std::string seed=PBKDF2(password,confString);

    for (int k = 0; k < iterations; k++) {
        result = getResult(seed,confString);

        final= result;
        seed= getResult(result,confString);
    }
    return final;
}
```

1.2 RSA Generation

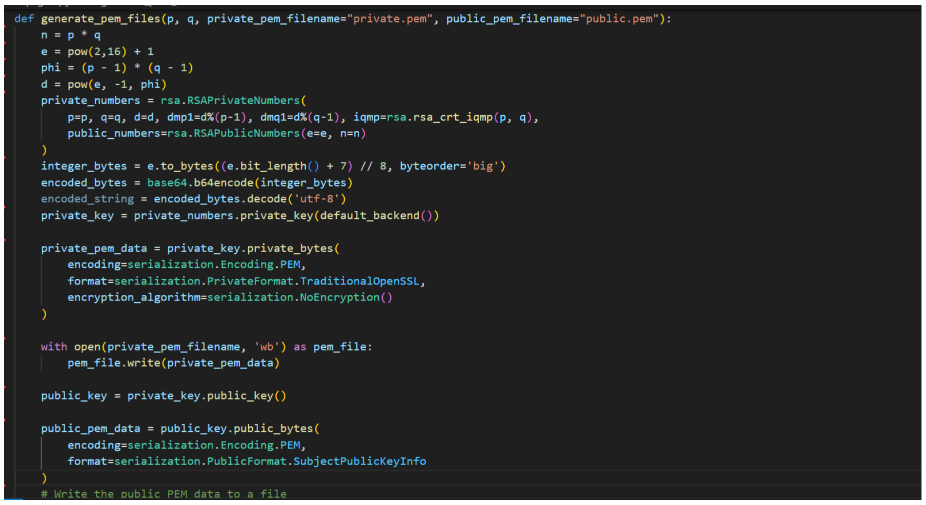
As this part was only required to be made in one language we chose Python for its ease of use and we were already familiar with it. As a strategy, we used the key stream to generate two prime numbers to be used in our RSA key. To achieve this we check if the number is divisible by two (if it is not a prime), and if is not, check if it is a prime using Miller Rabin test. We keep adding two until we get two prime numbers.



```
def generate_prime_from_bytes(random_bytes):
    candidate_prime = 0
    for c in random_bytes:
        candidate_prime = (candidate_prime << 8) | ord(c)
    if candidate_prime % 2 == 0:
        candidate_prime += 1
    while not number.isPrime(candidate_prime):
        candidate_prime += 2

    return candidate_prime

Comment Code
def keygen(keystream):
    leng=int(len(keystream))
    leng= leng//2
    key1= keystream[:leng]
    key2= keystream[leng:]

    p= generate_prime_from_bytes(key1)
    q= generate_prime_from_bytes(key2)


    return p,q
```

1.2.1 PEM

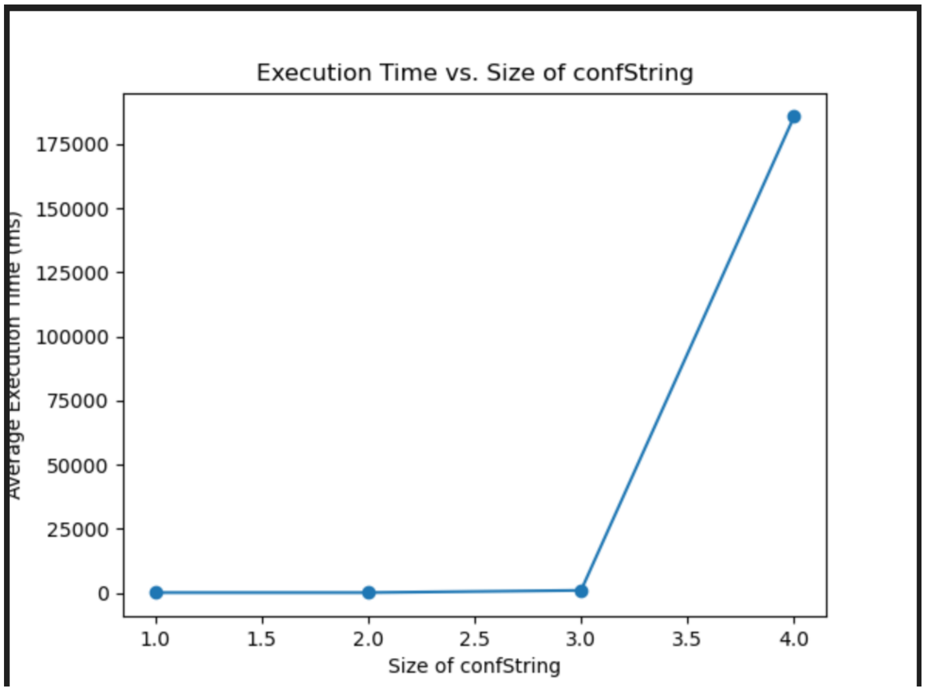
The first step in this stage is to calculate the modulus, define the exponent, and calculate the private exponent (d), we just need to pass our values to a function make an object from the cryptodome library, and input the format, serialization and name of the files were the key will be saved.

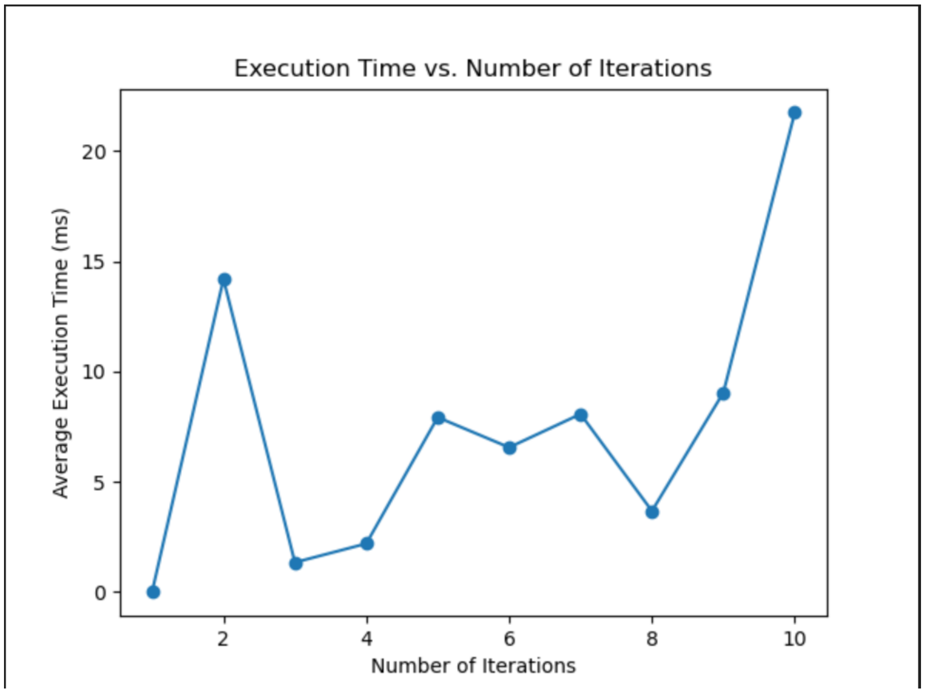


```python
def generate_pem_files(p, q, private_pem_filename="private.pem", public_pem_filename="public.pem"):
    n = p * q
    e = pow(2,16) + 1
    phi = (p - 1) * (q - 1)
    d = pow(e, -1, phi)
    private_numbers = rsa.RSAPrivateNumbers(
        p=p, q=q, d=d, dmp1=d%(p-1), dmq1=d%(q-1), iqmp=rsa.rsa_crt_iqmp(p, q),
        public_numbers=rsa.RSAPublicNumbers(e=e, n=n)
    )
    integer_bytes = e.to_bytes((e.bit_length() + 7) // 8, byteorder='big')
    encoded_bytes = base64.b64encode(integer_bytes)
    encoded_string = encoded_bytes.decode('utf-8')
    private_key = private_numbers.private_key(default_backend())

    private_pem_data = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    )

    with open(private_pem_filename, 'wb') as pem_file:
        pem_file.write(private_pem_data)

    public_key = private_key.public_key()

    public_pem_data = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    # Write the public PEM data to a file
```

1.3 Results

As requested we made applications for making graphs relating the size of the confusion string and the number of iterations. These were the results running the randgen app in python:




Execution Time vs. Number of Iterations
Average Execution Time (ms)
0
5
10
15
20
2
4
6
8
10
Number of Iterations