CRIPTOGRAFÍA

PRÁCTICA 3: OpenSSL y Criptografía Pública

AUTORES

CARLOS GARCÍA SANTA

INGENIERÍA INFORMÁTICA

ESCUELA POLITÉCNICA SUPERIOR

UNIVERSIDAD AUTÓNOMA DE MADRID

20/04/2025

Índice

1. OpenSSL

a. Cifrados simétricos

Tras ejecutar el comando openssl ciphers -v, se obtiene el listado de cifrados disponibles en OpenSSL. El resultado obtenido se compone de líneas donde se especifica la familia criptográfica, la versión TLS, el método de intercambio de claves (*Kx*), autenticación (*Au*), cifrado (*Enc*) y el código de autenticación del mensaje (*Mac*). En total se han detectado únicamente tres algoritmos de cifrado simétrico distintos:

- AES-GCM
  - Aparece codificado como AESGCM(128) y AESGCM(256), donde el número indica la longitud de la clave en bits y GMC indica el modo de operación, en este caso Galois Counter Mode, que proporciona confidencialidad y autenticidad (AEAD).
- AES-CBC
  - Identificado como AES(128) y AES(256), este modo corresponde al Cipher Block Chaining, este cifrado no proporciona AEAD por ende, se usa un SHA asociado.
- CHACHA20-POLY1305
  - Cifrado que a diferencia de AES, no requiere instrucciones específicas de hardware para su ejecución eficiente. El cifrado se basa en el generador de flujo CHACHA20, mientras que POLY1305 proporciona autenticación. Su uso también es AEAD, garantizando integridad además de confidencialidad. La información la he obtenido de: https://en.wikipedia.org/wiki/ChaCha20-Poly1305

b. Cifrados asimétricos

Dentro de los algoritmos asimétricos que pueden utilizarse con OpenSSL se encuentran principalmente RSA, DHE (Diffie–Hellman Ephemeral) y los de curva elíptica (ECC).

El RSA es uno de los algoritmos asimétricos más extendidos. Su seguridad se fundamenta en la dificultad del problema de factorización de grandes enteros. Se basa en un par de claves generadas a partir de dos números primos, una clave pública, utilizada para cifrar o verificar firmas; y una clave privada, utilizada para descifrar o firmar digitalmente.
El DHE es una variante del protocolo de intercambio de claves Diffie–Hellman que incorpora el uso de claves temporales. Este enfoque implica que, aunque una clave privada sea comprometida, posteriormente, no podrá recuperarse el contenido de cifrados pasados. El protocolo permite a dos usuarios acordar una clave secreta compartida a través de un canal inseguro, sin necesidad de cifrar datos directamente. La información la he extraído de:
https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchangeeli

EC (Elliptic Curve Cryptography) se refiere a un conjunto de algoritmos asimétricos basados en la aritmética de curvas elípticas sobre cuerpos finitos. La principal ventaja de los algoritmos ECC frente a RSA es que ofrecen niveles similares de seguridad con claves mucho más pequeñas.
Información obtenida de:
https://en.wikipedia.org/wiki/Elliptic-curve_cryptography

c. Generación de claves privadas y públicas

El esquema de cifrado público RSA se basa en la teoría de números y, concretamente, en la dificultad computacional de factorizar grandes números enteros compuestos. Este sistema utiliza un par de claves: una clave pública, que puede compartirse libremente, y una clave privada, que debe mantenerse en secreto. El fundamento de seguridad de RSA radica en que, aunque es sencillo multiplicar dos números primos grandes para obtener un número compuesto, resulta computacionalmente inviable realizar la operación inversa, es decir, obtener los factores primos a partir de ese producto.

La generación de claves RSA comienza con la elección de dos números primos grandes p y q, a partir de ellos se calcula el módulo $n = p * q$, que será común tanto para la clave pública como para la privada, con esto se calcula $\varphi(n) = (p - 1)(q - 1)$. Se escoge un exponente público *e que cumpla con*

$mcd(e, \varphi(n)) = 1$, y se calcula el exponente privado d como $ed \equiv 1\, mod\, \varphi(n)$. La clave pública queda formada por el par (n,e), y la clave privada por (n,d).

En un proceso de cifrado, un mensaje M (convertido previamente a un número menor que n) se cifra aplicando la operación $C = M^{e}\, mod\, n$. Para descifrarlo, se realiza $M = C^{d}\, mod\, m$, recuperando así el mensaje original.

Para generar un par de claves RSA utilizando OpenSSL, se ejecutan los siguientes comandos:

```
[santacg@archlinux ~]$ openssl genpkey -algorithm RSA -out clave_privada.txt
..+.....+.+.........+......+..++++++++++++++++++++++++++++++++++++++++*....+...+
........+....+......+.........+.....................+..+.......+...+..+.+.....+
...+..+.......+...............+...........+..........+.........+.....+.........
...........+.+..................+......+.+...+.....+...+....+..................
........+...+......+....+...........+......+...+....+......+.....+..........+..
.+......+....+............+.........+.........+.+......+...+...+..+.+..+...+.+..
....+.+..+..............++++++
.....+..+...+.+.....+....+......+.....+.+.........+.......+......+.............
+++*......+...+.........+.+........+....+........+.............+..+....+.......
[santacg@archlinux ~]$ cat clave_privada.txt
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDG75VDU87SkeiQ
pERNg9tQ+RnjIIaqL0O4g17U/cOWoBRuXTmZ5unY8HKUb/yo5oS2+u7F98q5+jL4
Tl96mvvCQZRGnXQsbcA+vFU9iNP8F/Bmt4DvwDQkth/zP+ASeeibvPXBUqwnWLgy
/1jsZKFVnMtLMciElXp3eQmfadD4za6/4tavl+nXLf07vNld4C1fpKpoH5aYtsp2
rmOArmtOVTwF8uFFYeSyh1FwfM7zt45O+wjao+cKn4Rw8be5VZtAiesBlyKMr9eF
JSc5oNYEgJxxQA0vx4+gWrR7d0nf/zyByE67qDvONgJt2TlDXngWFI9I1InaP+mN
KC0CebMRAgMBAAECggEADY7VguWq0lIMYhM37np+j3myqHaWhvG6sVAOvgBffJwD
WHQ1EUaaCGV2waQSOaagQ+I9NLPLFrq1D5Qz0KTQZLBYfSmM+dwu/neSbaOuIt+0
zCNUf8J/K0NYURk42bmCP1709sx+HB5hS2R4uH+xOSb6+fB2JIcVdiADJmCtU89+
AsFSNBcjoQAaSnITu5cH6zJL0nzqFiLDfC/Cog3mVoEDHJptsgXpWxS0ubYPRBbf
RH8achegrWnmdW+vnBhbcwhN80CLa0J9xiSO03g4YN3h88hnmaaUGxqP4Y0B4qbl
5sNiLV2SD/zhajGQQCnr2QUxowFKfpwlpuH8TJyzwwKBgQDpZH8lALbKwMY7R1XR
As2jFoxmFvgGYyWSUiGvtR5j84Xv2eMvg1sIUKF7qh3freOv6/hOKt+vM532PjQf
ClPqFYDQADYyF8T3PffRzfR+soln7uDfCwLZBiPVmDb9K7ZeMg+NFJDrpHv2cIjH
aee9kX6FA+kefq/Oo52NoYeT6wKBgQDaNKdbg6TQGUH/pvKzUYr9BK0kmu9NxxJJ
e8xrPokoM7qoRYkoWLtiGZ6Lq6yR5aw+QSxX+lvGEOtEW0EiDYzwDLN6HglomfLP
9s03nWuj+YqaPCPHhFT2Pasxh8DVXTS/YWMhYlOrqfk+KkA6RN0B77w0CnY82gzd
ueeFS2Uh8wKBgEtreFLCyk72wJhDUw35s+3xyWjsHzYhL/D6rXwUMT/nX+7EIFAo
SHCFbcmCjajMjnk47cKExEffjDoYm8s0ES3hyQBuMRU67SFgWwbnYQzYMOLvJKeu
7Tqoa0gIqixhxcrmrnPOo44iVxSqBOk75fEyZQoi22BJXBKQeMdd3WwFAoGBAL31
SqCLiQY0htUlpHFhEotT79tHGf7ux8e7A2l1ZnfeGKf26QF7xZ0QSoqy6CeW9wy3
NgBdDlnuTrISczBO9ZvgorozveMW6mALLfo1jAiS7At7QXamiERZyVf0I3Zbjtjw
0hi0/bwa28U3iPRxEKln58KyPN6awMqh1aCf0ynrAoGAHtIEleIDJKijRKpiVqji
2tJh7ITiYzUBl5WiVB/nlkKhweKhJF2XMCcSYTZ1lAJkNBhYUUWmg42edOsiekw8
u8KNGkG0A6nyHICyGpunGdPQ/xc8nAh2SrVibs3WS/7IXvYo3UgPWGocg8B3ULZD
n4jomVw1SGm9etBd2zN3Bcw=
-----END PRIVATE KEY-----
```

```
-----END PRIVATE KEY-----
[santacg@archlinux ~]$ openssl rsa -pubout -in clave_privada.txt -out clave_publica.txt
writing RSA key
[santacg@archlinux ~]$ cat clave_publica.txt
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAxu+VQ1PO0pHokKRETYPb
UPkZ4yCGqi9DuINe1P3DlqAUbl05mebp2PBylG/8qOaEtvruxffKufoy+E5fepr7
wkGURp10LG3APrxVPYjT/BfwZreA78A0JLYf8z/gEnnom7z1wVKsJ1i4Mv9Y7GSh
VZzLSzHIhJV6d3kJn2nQ+M2uv+LWr5fp1y39O7zZXeAtX6SqaB+WmLbKdq5jgK5r
TlU8BfLhRWHksodRcHzO87eOTvsI2qPnCp+EcPG3uVWbQInrAZcijK/XhSUnOaDW
BICccUANL8ePoFq0e3dJ3/88gchOu6g7zjYCbdk5Q154FhSPSNSJ2j/pjSgtAnmz
EQIDAQAB
-----END PUBLIC KEY-----
```

d. Diferencia de velocidades entre cifrados simétricos y asimétricos.

```
[santacg@archlinux ~]$ openssl speed aes-256-cbc
Doing aes-256-cbc ops for 3s on 16 size blocks: 211469736 aes-256-cbc ops in 2.99s
Doing aes-256-cbc ops for 3s on 64 size blocks: 58993564 aes-256-cbc ops in 2.99s
Doing aes-256-cbc ops for 3s on 256 size blocks: 15180043 aes-256-cbc ops in 2.99s
Doing aes-256-cbc ops for 3s on 1024 size blocks: 3817957 aes-256-cbc ops in 2.99s
Doing aes-256-cbc ops for 3s on 8192 size blocks: 478679 aes-256-cbc ops in 3.00s
Doing aes-256-cbc ops for 3s on 16384 size blocks: 239556 aes-256-cbc ops in 2.99s
version: 3.5.0
built on: Thu Apr 10 10:45:53 2025 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -march=x86-64 -mtune=generic -O2 -pipe -fno-plt
omit-frame-pointer -mno-omit-leaf-frame-pointer -g -ffile-prefix-map=/build/openssl/src=/usr/src/deb
CPUINFO: OPENSSL_ia32cap=0x7ef8320b078bffff:0x0040069c219c97a9:0x0000000000000010:0x000000000000000
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
aes-256-cbc    1131610.63k  1262738.49k  1299695.99k  1307554.50k  1307112.79k  1312670.74k
```

```
[santacg@archlinux ~]$ openssl speed rsa2048
Doing 2048 bits private rsa sign ops for 10s: 22812 2048 bits private RSA sign ops in 9.97s
Doing 2048 bits public rsa verify ops for 10s: 796355 2048 bits public RSA verify ops in 9.98s
Doing 2048 bits public rsa encrypt ops for 10s: 746092 2048 bits public RSA encrypt ops in 9.85s
Doing 2048 bits private rsa decrypt ops for 10s: 22550 2048 bits private RSA decrypt ops in 9.97s
Doing rsa2048 keygen ops for 10s: 306 rsa2048 KEM keygen ops in 9.99s
Doing rsa2048 encaps ops for 10s: 714849 rsa2048 KEM encaps ops in 9.89s
Doing rsa2048 decaps ops for 10s: 22825 rsa2048 KEM decaps ops in 9.97s
Doing rsa2048 keygen ops for 10s: 312 rsa2048 signature keygen ops in 9.99s
Doing rsa2048 signs ops for 10s: 22802 rsa2048 signature sign ops in 9.97s
Doing rsa2048 verify ops for 10s: 798356 rsa2048 signature verify ops in 9.98s
version: 3.5.0
built on: Thu Apr 10 10:45:53 2025 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -march=x86-64 -mtune=generic -O2 -pipe -fno-plt -
omit-frame-pointer -mno-omit-leaf-frame-pointer -g -ffile-prefix-map=/build/openssl/src=/usr/src/debu
CPUINFO: OPENSSL_ia32cap=0x7ef8320b078bffff:0x0040069c219c97a9:0x0000000000000010:0x0000000000000000:
                   sign    verify    encrypt   decrypt   sign/s verify/s  encr./s  decr./s
rsa  2048 bits 0.000437s 0.000013s 0.000013s 0.000442s   2288.1  79795.1  75745.4   2261.8
                               keygen    encaps    decaps keygens/s  encaps/s  decaps/s
                    rsa2048 0.032647s 0.000014s 0.000437s      30.6   72280.0    2289.4
                               keygen     signs    verify keygens/s    sign/s  verify/s
                    rsa2048 0.032019s 0.000437s 0.000013s      31.2    2287.1   79995.6
```

La comparación entre cifrados simétricos y asimétricos revela una diferencia significativa en cuanto a velocidad de procesamiento. En las pruebas realizadas, el algoritmo simétrico AES-256-CBC alcanza velocidades de hasta 1.312.670 kilobytes por segundo en bloques de 16384 bytes, lo que muestra su alta eficiencia en el tratamiento de grandes volúmenes de datos. En contraste, el algoritmo asimétrico RSA-2048 muestra una velocidad media de unas 2287 operaciones por segundo en el caso de firmado y descifrado con la clave privada, y cerca de 80.000 operaciones por segundo en verificaciones con clave pública, donde cada operación procesa únicamente un bloque pequeño. Esta diferencia cuantitativa confirma que los cifrados simétricos están optimizados para ofrecer un rendimiento alto, mientras que los asimétricos, mucho más costosos computacionalmente, se reservan para tareas puntuales como el intercambio inicial de claves o la autenticación.

e. Certificados X.509

Un certificado digital X.509 permite vincular de forma verificable una clave pública con un usuario determinado. Su utilidad radica en posibilitar la autenticación y el cifrado seguro en protocolos como HTTPS o correo electrónico. Cada certificado contiene un número de versión, número de serie, identidad del sujeto, emisor, fechas de validez y los algoritmos empleados para firma y cifrado.

Para generar la solicitud de certificado (CSR), se ha utilizado la clave privada previamente generada. Durante el proceso, se introducen los campos que conforman el Distinguished Name (DN), como el país, ciudad, organización o el nombre común (CN) asociado al certificado.

```
[santacg@archlinux ~]$ openssl req -new -key clave_privada.txt -out solicitud.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:Madrid
Locality Name (eg, city) []:Madrid
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Carlos.ltd
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:Santa
Email Address []:santa@gmail.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:holaadiosholaadios
An optional company name []:.
```

Una vez obtenida la CSR, se genera un certificado X.509 auto-firmado mediante la clave privada. Esto significa actuar como autoridad de certificación (CA), firmando el certificado con la misma clave usada para generar el par de claves. Este certificado vincula la clave pública del usuario con su identidad y la firma digital asegura su integridad. El resultado es un certificado válido durante un día.

```
[santacg@archlinux ~]$ openssl x509 -req -in solicitud.csr -signkey clave_privada.txt -days 1 -out certificado.crt
Certificate request self-signature ok
subject=C=ES, ST=Madrid, L=Madrid, O=Carlos.ltd, OU=IT, CN=Santa, emailAddress=santa@gmail.com
```

Finalmente, se analiza el contenido del certificado X.509 generado, openSSL permite verificar internamente la firma del certificado, comprobando criptográficamente que la clave pública contenida coincide con la utilizada para firmarlo, garantizando su validez y autenticidad.

```
[santacg@archlinux ~]$ openssl x509 -in certificado.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            01:ed:c4:21:83:7b:58:97:64:8f:6c:59:43:6f:92:f4:81:28:dc:af
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=ES, ST=Madrid, L=Madrid, O=Carlos.ltd, OU=IT, CN=Santa, emailAddress=
        Validity
            Not Before: Apr 17 17:25:21 2025 GMT
            Not After : Apr 18 17:25:21 2025 GMT
        Subject: C=ES, ST=Madrid, L=Madrid, O=Carlos.ltd, OU=IT, CN=Santa, emailAddress
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:c6:ef:95:43:53:ce:d2:91:e8:90:a4:44:4d:83:
                    db:50:f9:19:e3:20:86:aa:2f:43:b8:83:5e:d4:fd:
                    c3:96:a0:14:6e:5d:39:99:e6:e9:d8:f0:72:94:6f:
                    fc:a8:e6:84:b6:fa:ee:c5:f7:ca:b9:fa:32:f8:4e:
                    5f:7a:9a:fb:c2:41:94:46:9d:74:2c:6d:c0:3e:bc:
                    55:3d:88:d3:fc:17:f0:66:b7:80:ef:c0:34:24:b6:
                    1f:f3:3f:e0:12:79:e8:9b:bc:f5:c1:52:ac:27:58:
                    b8:32:ff:58:ec:64:a1:55:9c:cb:4b:31:c8:84:95:
                    7a:77:79:09:9f:69:d0:f8:cd:ae:bf:e2:d6:af:97:
                    e9:d7:2d:fd:3b:bc:d9:5d:e0:2d:5f:a4:aa:68:1f:
                    96:98:b6:ca:76:ae:63:80:ae:6b:4e:55:3c:05:f2:
                    e1:45:61:e4:b2:87:51:70:7c:ce:f3:b7:8e:4e:fb:
                    08:da:a3:e7:0a:9f:84:70:f1:b7:b9:55:9b:40:89:
                    eb:01:97:22:8c:af:d7:85:25:27:39:a0:d6:04:80:
                    9c:71:40:0d:2f:c7:8f:a0:5a:b4:7b:77:49:df:ff:
                    3c:81:c8:4e:bb:a8:3b:ce:36:02:6d:d9:39:43:5e:
                    78:16:14:8f:48:d4:89:da:3f:e9:8d:28:2d:02:79:
                    b3:11
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                7B:65:12:32:4B:A4:5C:A7:F4:1D:97:11:DD:76:56:3B:13:A6:A4:9B
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        57:9a:fc:3f:10:f0:2e:90:22:8e:36:89:bd:57:fc:42:76:ad:
        3b:59:a1:87:c8:12:5d:8a:b1:f7:ed:28:30:a5:e0:f6:83:4a:
        d6:03:ab:18:9e:ee:34:a1:71:38:cc:5c:d2:a4:07:51:a8:94:
        ba:32:ad:71:79:f0:e5:41:a7:9c:c8:db:eb:62:22:f8:dc:81:
```

2. RSA

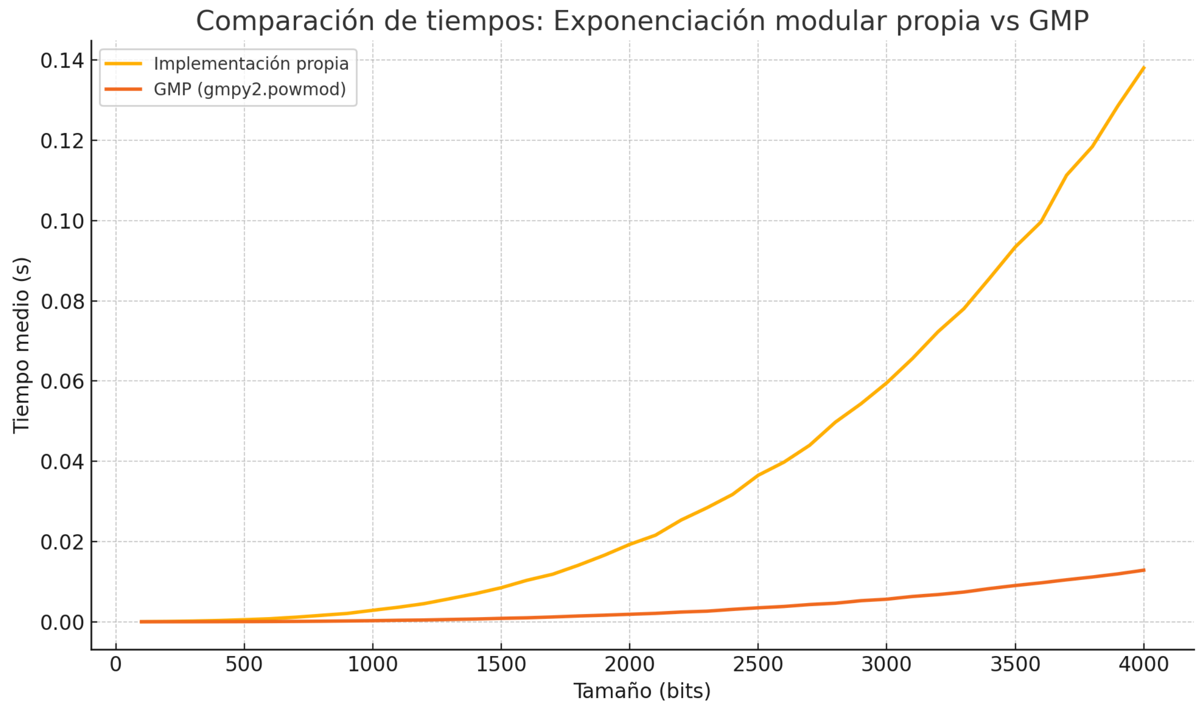
a. Potenciación de grandes números

Para resolver el problema de la potenciación modular de grandes números de forma eficiente, se ha implementado el algoritmo de exponenciación binaria modular siguiendo la siguiente referencia (Right-to-left binary method): https://en.wikipedia.org/wiki/Modular_exponentiation. Este algoritmo permite calcular eficientemente potencias modulares incluso cuando el exponente *e* y el módulo *m* son de gran tamaño. La eficiencia se logra al reducir el número de multiplicaciones necesarias mediante la descomposición binaria del exponente.

El script desarrollado en *exp_mod.py* implementa este algoritmo en la función *binary_exponentiation*(b, e, m) y proporciona dos modos de uso:

- Modo calc: ejecuta la operación modular con los tres operandos introducidos (base, exponente, módulo), mostrando el resultado tanto de la implementación que se ha realizado como el obtenido mediante la función powmod() de la biblioteca *GMPY2*, para verificar la corrección del algoritmo.
- Modo test: realiza un análisis comparativo de rendimiento entre la implementación desarrollada y la función *powmod* de *GMPY2*, usando pruebas aleatorias por tamaño, con tamaños de entrada que varían entre 100 y 4000 bits en pasos de 100.

Los resultados obtenidos muestran que ambos algoritmos proporcionan resultados idénticos para cada operación, garantizando la corrección del funcionamiento. No obstante, en cuanto al rendimiento, como se observa en la imagen siguiente, la implementación de GMP presenta tiempos notablemente inferiores debido a su implementación en bajo nivel, altamente optimizada, esto se nota especialmente a medida que aumentamos el número de bits.



b. Generación de números primos: Miller-Rabin

El script *miller_rabin.py* permite generar un número primo de un número arbitrario de bits, especificando además una probabilidad máxima de error aceptable, la cual determina el nivel de seguridad del test. El algoritmo comienza generando un número impar aleatorio de la longitud indicada, y lo somete al test de Miller-Rabin con un número de iteraciones *k* calculado en función del tamaño del número y de la probabilidad de error requerida, siguiendo la siguiente fórmula:

$$p = \frac{1}{1 + \frac{4^m}{n * \ln 2}} \Rightarrow m = \frac{\ln[(\frac{1}{p} - 1) n \ln 2]}{\ln 4}$$

Luego, para cada iteración, se selecciona una base aleatoria $a \in [2, n - 2]$, y se comprueba si el número falla las condiciones que caracterizan a los primos en el test. Si el número pasa todas las iteraciones, se considera como probablemente primo.

El programa incluye, además, una comparación con la función de *GMPY2 is_prime()*, contrastando tanto los resultados obtenidos como los tiempos de ejecución. En el ejemplo ejecutado, se ha generado un número primo de 8096 bits con una probabilidad de error inferior a 10^{-7}. El resultado del test propio y del test de GMP ha sido coincidente, en ambos casos, el número se clasifica como probablemente primo.

```
(Cripto_venv) [santacg@archlinux P3]$ python3 miller_rabin.py -b 8096 -p 1e-7
Número candidato: 129001705208173744642326592985159374575840073818047987202764534
8006701153738022947381084588340688948384738857991636201849990855320963538732249
3579079145463865965008300057965684961477408828879449896675066681478009935764655
1293008200017065186083995972707369403535853470191106008780055581088453660386835154
1598715097695362246476284079625246885281625048991258433376972427679310352399403
7266035621864181690834309151116500476110337528416009445161124701641583261333443
7803684749470842646797176989279811933721689300176632711536014844452796616870348700
6203709633003179099886728918259464516156849871810671142149628664063838199596232
9034588678878847910188182100031103778229157138332504694516523743461772143068890225
3463440670994580544400555829742138196781904385448342164161287400024315648531761
086271000008856301557386331557880422134836824081796630047
Resultado del test: Probablemente primo
Resultado de GMP: Probablemente primo
Probabilidad de equivocación: 1e-07
Número de iteraciones del test: 17
Tiempo de ejecución del test de GMP: 0.292804 segundos
Tiempo de ejecución del test:        388.533777 segundos
```

c. Factorización del módulo del RSA mediante el conocimiento de d (A. las Vegas)

El ataque probabilístico de tipo Las Vegas, implementada en *vegas.py* explota la relación fundamental del sistema RSA:

$$ed \equiv 1\, mod\, \varphi(n)$$

El ataque se basa en generar múltiples valores aleatorios ww y aplicar un procedimiento similar al test de Miller-Rabin, buscando descubrir una no trivial raíz cuadrada de la unidad módulo *n*, es decir, un valor *x* tal que:

$$x^2 \equiv mod\, n \Leftrightarrow x \neq \pm 1\, mod\, n$$

El script desarrollado acepta como entrada el módulo RSA *n*, el exponente público *e* y el exponente privado *d*, e intenta recuperar los factores *p* y *q* tales que $n = p * q$. Si el ataque tiene éxito, muestra los factores encontrados junto con la verificación de que su producto coincide con *n*. En caso contrario, informa que no se ha podido factorizar dentro del número de intentos establecido.

```
(Cripto_venv) [santacg@archlinux P3]$ python3 vegas.py -n 187 -e 7 -d 23
Factores encontrados:
p = 11
q = 17
Verificación (p*q = n): 187 = 187
```