МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Лабораторна робота №1

З дисципліни «МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ МЕХАНІЗМІВ»

Роботу виконав

студент групи ФІ-21мн, ФТІ

Татенко Вадим

Хмелевський Святослав

Кірсенко Єгор

2023

Мета роботи

Хід роботи

OpenSSL

The OpenSSL crypto library (libcrypto) implements a wide range of cryptographic algorithms used in various Internet standards. The services provided by this library are used by the OpenSSL implementations of TLS and CMS, and they have also been used to implement many other third-party products and protocols.

The functionality includes symmetric encryption, public key cryptography, key agreement, certificate handling, cryptographic hash functions, cryptographic pseudo-random number generators, message authentication codes (MACs), key derivation functions (KDFs), and various utilities.

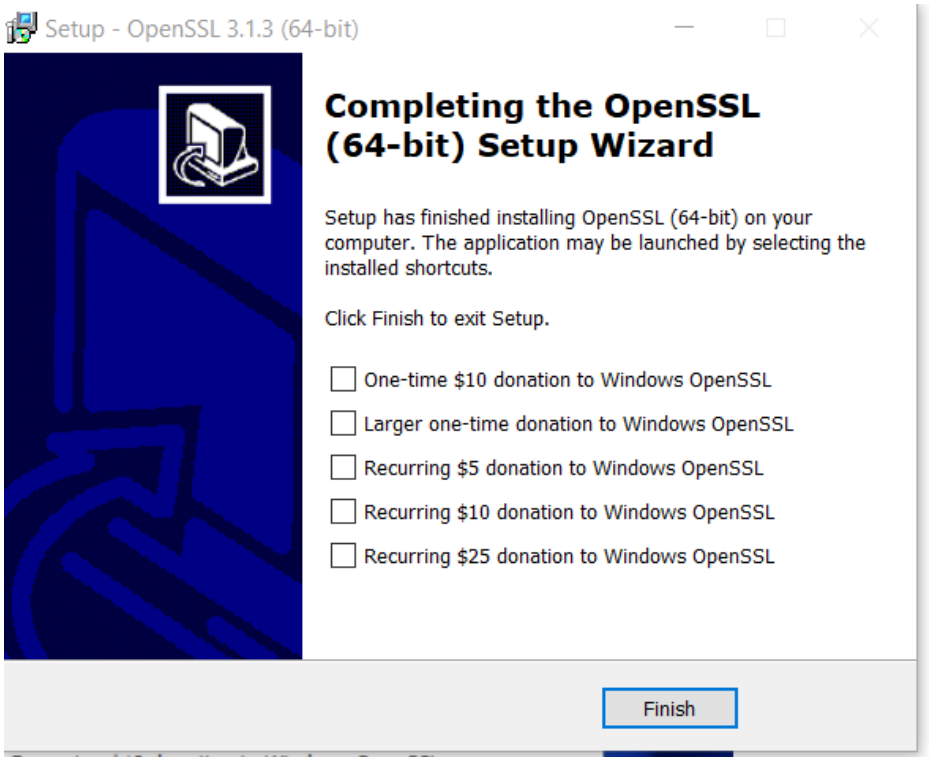
Download:

If you are looking to download the OpenSSL package for your Windows machine from its official website, you can't. It's because OpenSSL doesn't release authorized OpenSSL installers for Windows. You should depend on a few third-party distributors distributing OpenSSL installer files for Windows platforms. OpenSSL has published the list of all trusted third-party distributors on its Wiki page.

We chose this site
And this library version (v3.1.3)

Win64 OpenSSL v3.1.3 EXE | MSI	140MB Installer	Installs Win64 OpenSSL v3.1.3 (Recommended for software developers by the creators of OpenSSL). Only installs on 64-bit versions of Windows and targets Intel x64 chipsets. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.

Installation:



Configuration:

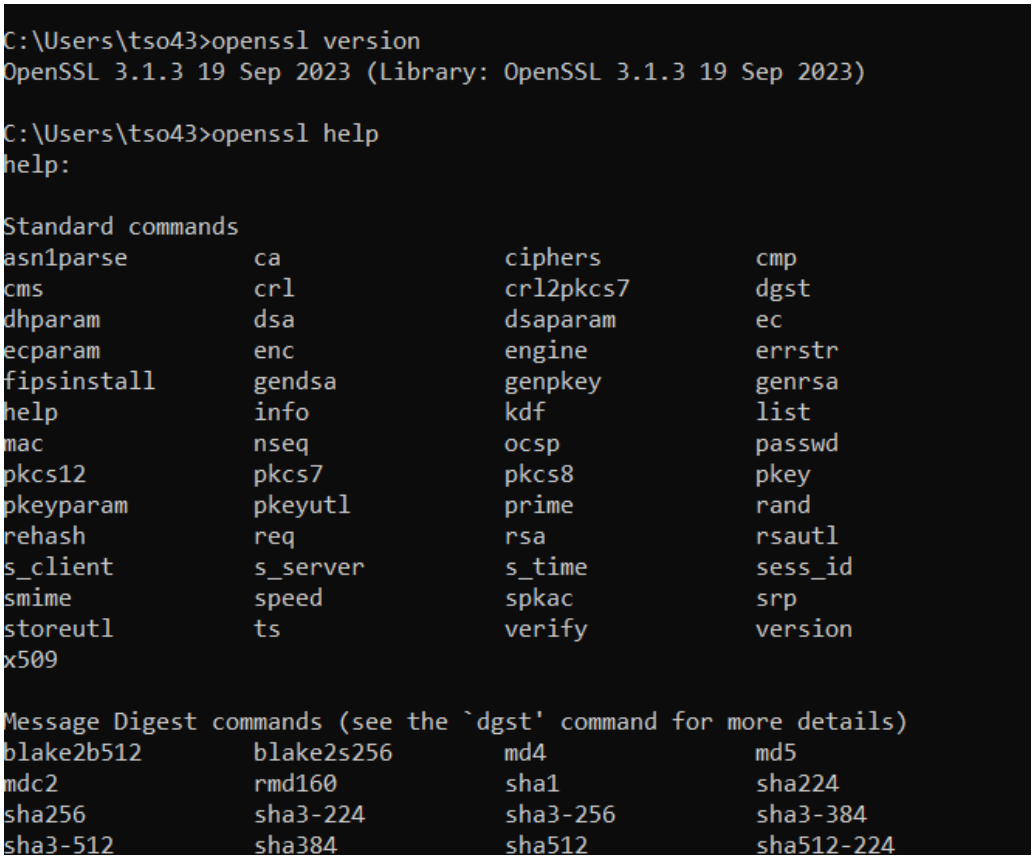
Set ENV_variables:

- set OPENSSL_CONF=E:\универ\6 - курс\Крипта\OpenSSL-Win64\bin\openssl.cfg
- set Path= E:\универ\6 - курс\Крипта\OpenSSL-Win64\bin

It didn`t help

So the other way to set ENV is to open win+r; type “sysdm.cpl”; go to Advanced > Environment Variable; set OPENSSL_CONF and Path with values like the above

Congrats, we installed the library



Crypto algorithms

aes-256

To encrypt a file we need to call the command in cmd

openssl enc -aes256 -pbkdf2 -in input.txt -out encrypted.txt -k *********

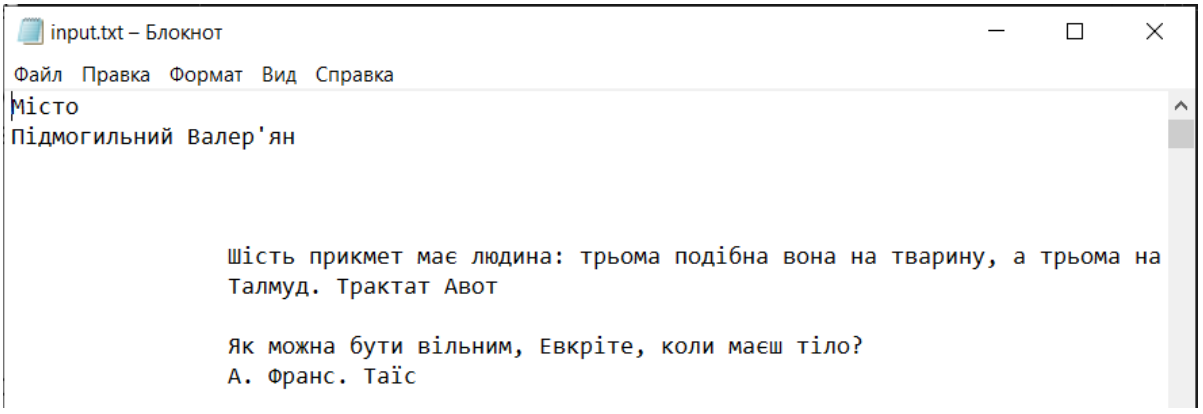
enc - encryption flag
-pbkdf2 - Password-based key derivation function 2: encryption option
-in - input file
-out - output file

PBKDF2 – is a cryptographic key derivation function that is commonly used to derive encryption keys and cryptographic secrets from a password. PBKDF2 is designed to make it computationally expensive and time-consuming for attackers to perform brute-force and dictionary attacks on the password.

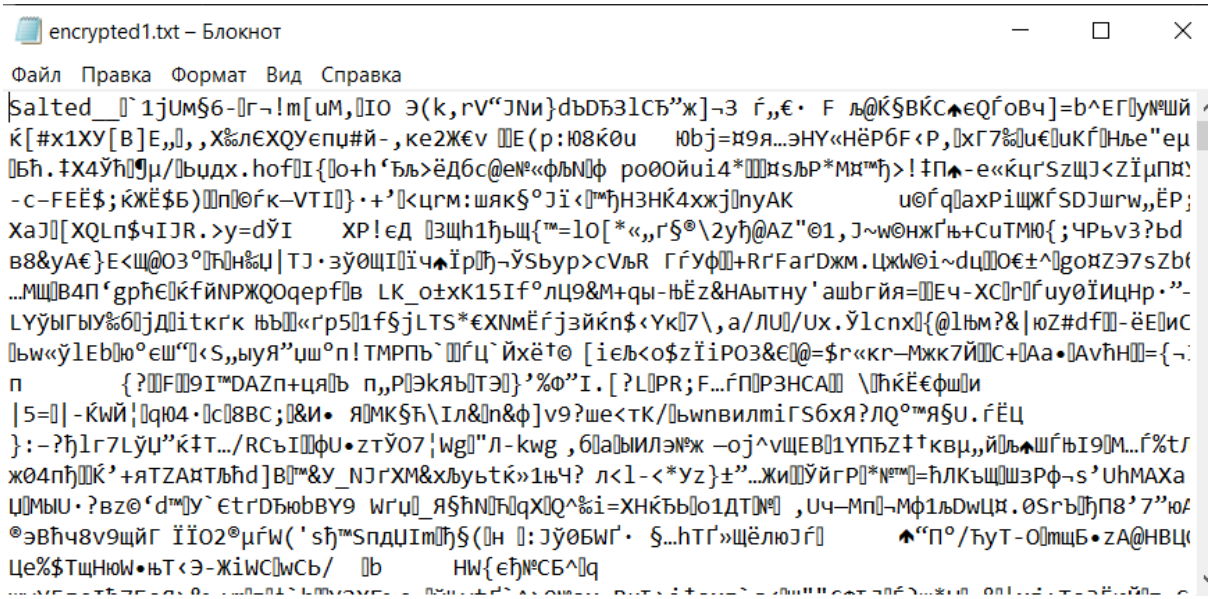
It takes several parameters:

- Password: The user's password is used as the input to the key derivation process. Passwords are generally not suitable for direct use as encryption keys because they tend to be too short and may not contain enough entropy. PBKDF2 makes it possible to derive a longer, more complex key from the password.
- Salt: A random value known as a salt is generated and used as an additional input. The salt adds randomness to the process and ensures that even if two users have the same password, their derived keys will be different due to the different salts. Salts are typically stored alongside the derived key or securely with the encrypted data.
- Iteration Count: PBKDF2 applies a hash function (e.g., HMAC-SHA1) repeatedly to the combination of the password and salt. The number of iterations determines how many times the hash function is applied. The purpose of the iteration count is to make the key derivation process time-consuming and computationally expensive, which makes it harder for attackers to perform exhaustive searches.
- Key Length: You specify the desired length for the derived key. The derived key will be of this length.

input.txt contains the text: novel Misto



After executing the command we get an encrypted.txt file with the next content



Btw, PBKDF2 as the default uses the -iter flag with value 10000 and -salt flag

Now let`s measure **encryption time**:

*Measure-Command {openssl enc -aes256 -pbkdf2 -in input.txt -out encrypted.txt -k *********}*

********* - password

```
PS E:\универ\6 - курс\Крипта\lab1> Measure-Command {openssl enc -aes256 -pbkdf2 -in input.txt -out encrypted.txt -k 1qaz2wsx0}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 79
Ticks             : 796465
TotalDays         : 9,21834490740741E-07
TotalHours        : 2,21240277777778E-05
TotalMinutes      : 0,00132744166666667
TotalSeconds      : 0,0796465
TotalMilliseconds : 79,6465
```

As a result, we see that it took 79 milliseconds === 0.079 seconds to encrypt the file

Decryption Memory measure:

To measure the memory usage of openssl process, we need to do a specific thing:

- As the process works ~0.08 seconds, we have no time to get this process and get memory statistics from it.

So the decision was to create a cycle of 1000 iterations that creates one openssl process in iteration. While this cycle runs, in another PowerShell we run the command that gets the process and memory info from ti.

The cycle that runs openssl process (DDos command):

```
 # Set the number of iterations (replace 10 with your desired number)
$iterations = 1000
# Loop through the command multiple times
for ($i = 1; $i -le $iterations; $i++) {
    $outputFileName = "encrypted$i.txt"
    Write-Host "Running iteration $i, output file: $outputFileName"
    $result = Measure-Command { openssl enc -aes256 -pbkdf2 -in
input.txt -out $outputFileName -k 1qaz2wsx0 }
    # Display the time taken for each iteration
    Write-Host "Iteration $i took $($result.TotalSeconds) seconds"
}
```

```
Running iteration 126, output file: encrypted126.txt
Iteration 126 took 0.0826064 seconds
Running iteration 127, output file: encrypted127.txt
Iteration 127 took 0.089454 seconds
Running iteration 128, output file: encrypted128.txt
Iteration 128 took 0.1012491 seconds
Running iteration 129, output file: encrypted129.txt
Iteration 129 took 0.0893012 seconds
Running iteration 130, output file: encrypted130.txt
Iteration 130 took 0.0813171 seconds
Running iteration 131, output file: encrypted131.txt
Iteration 131 took 0.0836664 seconds
```

Get memory command:

```
$opensslProcess = Get-Process -Name "openssl"
$peakWorkingSet = $opensslProcess.PeakWorkingSet64
$peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
$peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
$PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
"Peak Working Set Memory: $($peakWorkingSet) bytes"
"Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
"Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
"PrivateMemorySeze64 Size: $($PrivateMemorySize64) bytes"
```

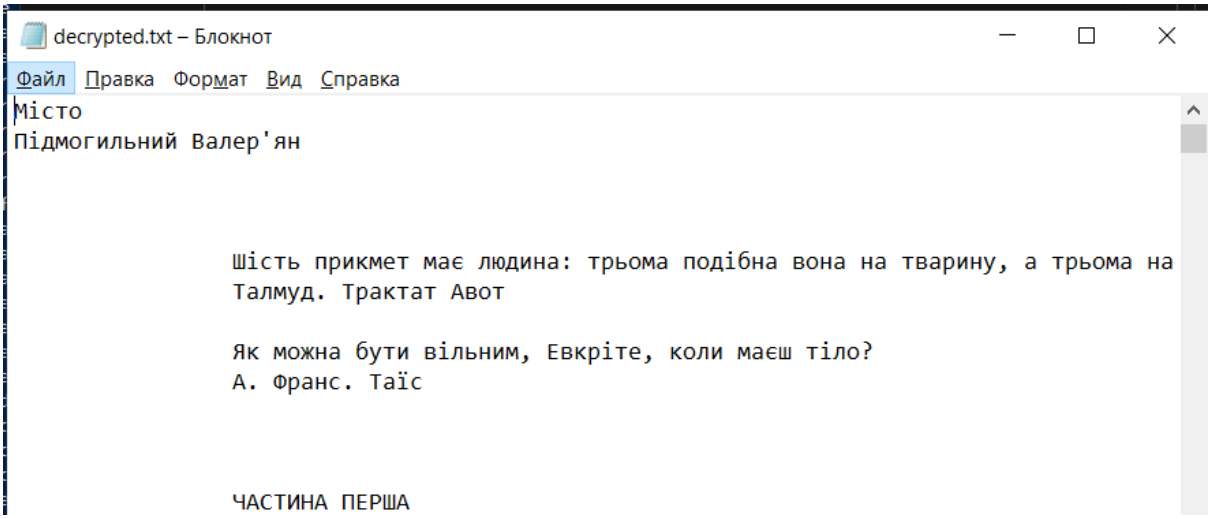
```
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySeze64 Size: $($PrivateMemorySize64) bytes"
Peak Working Set Memory: 7766016 bytes
Peak Paged Memory Size: 1695744 bytes
Peak Virtual Memory Size: 4390109184 bytes
PrivateMemorySeze64 Size: 1613824 bytes
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySeze64 Size: $($PrivateMemorySize64) bytes"
Peak Working Set Memory: 2076672 bytes
Peak Paged Memory Size: 430080 bytes
Peak Virtual Memory Size: 6352896 bytes
PrivateMemorySeze64 Size: 430080 bytes
```

Description of memory methods:

- *PeakWorkingSet64*: measures the maximum physical RAM used by the process.
- *PeakPagedMemorySize64*: measures the maximum combined RAM and disk space used by the process.
- *PeakVirtualMemorySize64*: measures the maximum virtual address space used by the process.
- *PrivateMemorySize64*: measures the current private memory usage, which is the portion of memory exclusively used by the process.

Let`s go to the **decryption part**
The command is:
openssl enc -d -aes256 -pbkdf2 -in encrypted.txt -out decrypted.txt -k
****
-d – flag that means decryption
Other – same as in the encryption part

As a result, we got a decrypted.txt file with text that was in the input.txt file



Decryption **time measure**:
Command:
Measure-Command {openssl enc -d -aes256 -pbkdf2 -in encrypted.txt -out decrypted.txt -k 1qaz2wsx0}

```
PS E:\универ\6 - курс\Крипта\lab1> Measure-Command {openssl enc -d -aes256 -pbkdf2 -in encrypted.txt -out decrypted.txt -k 1qaz2wsx0}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 84
Ticks             : 845634
TotalDays         : 9,78743055555555E-07
TotalHours        : 2,34898333333333E-05
TotalMinutes      : 0,00140939
TotalSeconds      : 0,0845634
TotalMilliseconds : 84,5634
```

As a result, we see that it took 84milliseonds === 0.084 seconds to encrypt the file

Decryption memory measure

The same algorithm as in encryption memory measure only changes the encryption command to decryption

DDos script:

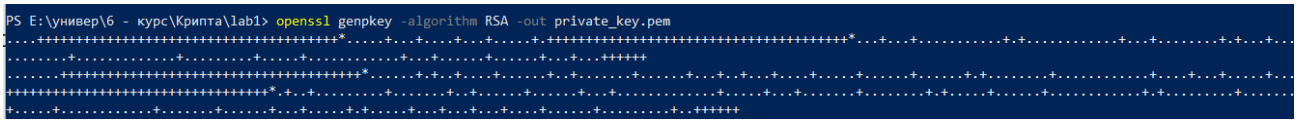
```
PS E:\универ\6 - курс\Крипта\lab1>  # Set the number of iterations (replace 10 with your desired number)
>> $iterations = 1000
>> # Loop through the command multiple times
>> for ($i = 1; $i -le $iterations; $i++) {
>>     $outputFileName = "decrypted$i.txt"
>>     Write-Host "Running iteration $i, output file: $outputFileName"
>>     $result = Measure-Command { openssl enc -aes256 -pbkdf2 -in input.txt -out $outputFileName -k 1qaz2wsx0 }
>>     # Display the time taken for each iteration
>>     Write-Host "Iteration $i took $($result.TotalSeconds) seconds"
>> }
>>
Running iteration 1, output file: decrypted1.txt
Iteration 1 took 0.0941415 seconds
Running iteration 2, output file: decrypted2.txt
Iteration 2 took 0.0916372 seconds
Running iteration 3, output file: decrypted3.txt
Iteration 3 took 0.0852341 seconds
Running iteration 4, output file: decrypted4.txt
Iteration 4 took 0.0822213 seconds
```

Memory measure:

```
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySize64 Size: $($PrivateMemorySize64) bytes"
Peak Working Set Memory: 2076672 bytes
Peak Paged Memory Size: 434176 bytes
Peak Virtual Memory Size: 6352896 bytes
PrivateMemorySize64 Size: 434176 bytes
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySize64 Size: $($PrivateMemorySize64) bytes"
Peak Working Set Memory: 2076672 bytes
Peak Paged Memory Size: 425984 bytes
Peak Virtual Memory Size: 6352896 bytes
PrivateMemorySize64 Size: 425984 bytes
```

rsa

First of all, we need to create a private key that will be used in the next steps to encrypt and decrypt file
To do it, we need to call the next command
openssl genpkey -algorithm RSA -out private_key.pem

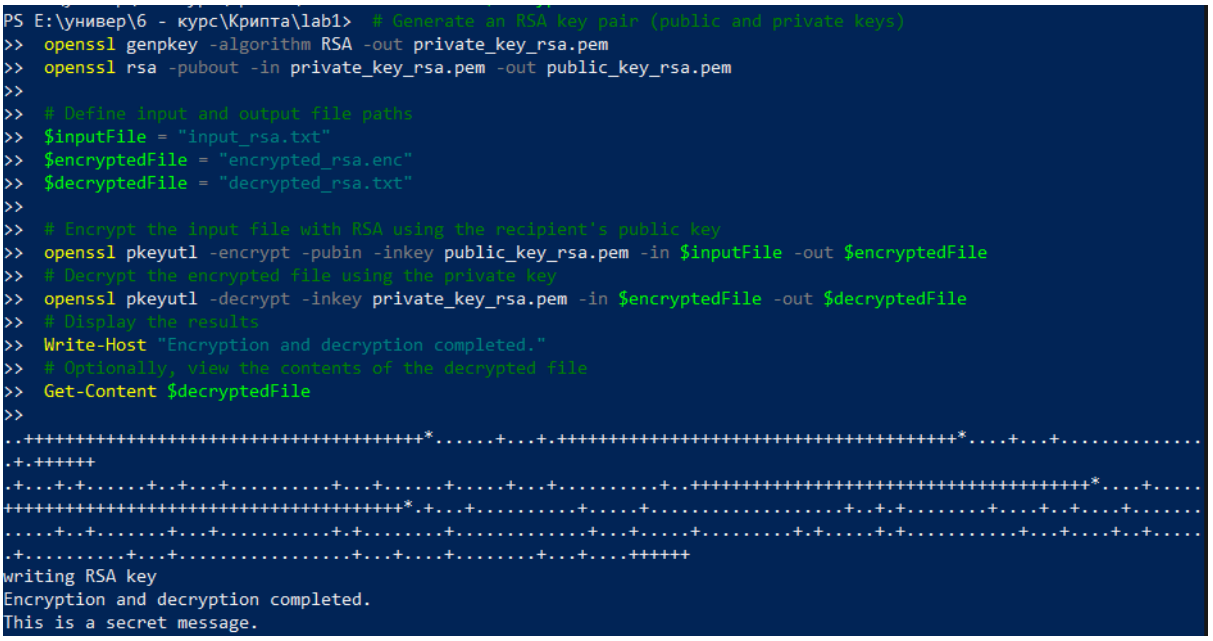


BTW, this library has commands to manipulate with this key
For example, this command prints out key components in standard outputs
openssl rsa -in private_key.pem -text -noout

```
PS E:\универ\6 - курс\Крипта\lab1> openssl rsa -in private_key.pem -text -noout
Private-Key: (2048 bit, 2 primes)
modulus:
    00:d7:51:62:4d:85:98:69:f1:15:5a:72:c8:02:7e:
    ba:11:56:06:a3:90:ee:17:12:37:c8:39:95:d2:df:
    3b:79:3b:12:0f:e0:13:2b:34:c1:62:26:7a:2c:0c:
    4b:4e:72:5c:cf:b0:67:24:a8:26:72:70:98:c2:8a:
    54:65:a9:ee:37:d5:4d:27:92:bf:91:63:41:36:74:
    2e:ee:74:32:ea:0b:8a:d6:e9:03:1f:11:19:28:da:
    a1:69:4f:55:bf:a5:03:35:7d:33:a2:2c:e3:74:08:
    27:18:2d:c0:6d:e4:ec:5c:42:42:3e:c9:c1:60:1b:
    a4:f4:76:23:94:c3:9a:78:23:2c:ff:94:9f:5a:79:
    54:00:04:38:3b:68:dd:3c:22:72:44:d0:81:71:53:
    a0:35:d6:4b:01:32:47:10:0f:6f:0a:1f:09:0c:3e:
    e7:45:5b:52:95:15:93:70:70:a7:cd:8f:88:20:e2:
    45:b2:61:93:c5:fa:47:80:d5:67:fb:40:9b:29:4f:
    81:46:fa:56:41:36:7b:5b:fb:06:33:29:a4:1f:6f:
    a9:6b:ff:1f:c5:63:65:f8:5f:d3:d6:4d:72:24:fc:
    ca:22:bb:01:95:3c:83:08:a1:cd:a8:fc:76:54:84:
    5c:09:77:04:f0:df:de:b1:44:00:ef:e6:37:f3:e1:
    f0:99
publicExponent: 65537 (0x10001)
privateExponent:
    63:35:08:4a:17:a7:e6:08:42:82:6e:28:61:9d:ea:
    35:5a:a1:ac:73:76:80:02:d0:d8:dc:4f:7f:dd:83:
    15:94:75:fa:02:90:52:73:f8:36:34:ec:4d:a7:4c:
    7b:54:8e:16:d3:7c:72:93:57:43:9b:45:87:3d:1d:
    2d:ca:62:0c:2f:a7:60:f8:68:73:a0:7b:c7:8f:38:
    a0:7d:c1:87:bc:d1:b7:bb:9c:c8:8f:a4:3c:02:a0:
    24:59:7d:74:d3:9e:ea:e2:49:0c:0c:0a:b0:c7:a0:
    db:af:18:de:89:7f:ad:f2:89:7b:4c:92:60:a9:dc:
    13:16:eb:c5:fb:6a:0c:b7:af:8b:c1:c4:f8:1f:60:
    e8:48:e2:ae:09:25:eb:ff:8d:94:82:00:7d:02:7e:
```

So, the next code is running all logic of:
- Creating RSA keys
- Encrypt *input.txt* file
- Decrypt *encrypted.enc*
- Print *decrypted.txt*

```
# Generate an RSA key pair (public and private keys)
openssl genpkey -algorithm RSA -out private_key_rsa.pem
openssl rsa -pubout -in private_key_rsa.pem -out public_key_rsa.pem
# Define input and output file paths
$inputFile = "inputRSA.txt"
$encryptedFile = "encrypted_rsa.enc"
$decryptedFile = "decrypted_rsa.txt"
# Encrypt the input file with RSA using the recipient's public key
openssl pkeyutl -encrypt -pubin -inkey public_key_rsa.pem -in $inputFile -out $encryptedFile
# Decrypt the encrypted file using the private key
openssl pkeyutl -decrypt -inkey private_key_rsa.pem -in $encryptedFile -out $decryptedFile
# Display the results
Write-Host "Encryption and decryption completed."
# Optionally, view the contents of the decrypted file
Get-Content $decryptedFile
```



To measure the time and memory usage:
Call DDos code

```
# Set the number of iterations (replace 10 with your desired number)
$iterations = 1000
# Loop through the command multiple times
for ($i = 1; $i -le $iterations; $i++) {
    $outputFileName = "encrypted_rsa$i.txt"
    Write-Host "Running iteration $i, output file: $outputFileName"
    $result = Measure-Command { openssl  pkeyutl -encrypt -pubin
-inkey public_key_rsa.pem -in input_rsa.txt -out encrypted_rsa$i.txt }
    # Display the time taken for each iteration
    Write-Host "Iteration $i took $($result.TotalSeconds) seconds"
 }
```

```
PS E:\универ\6 - курс\Крипта\lab1> # Set the number of iterations (replace 10 with your desired number)
>>  $iterations = 1000
>>  # Loop through the command multiple times
>>  for ($i = 1; $i -le $iterations; $i++) {
>>      $outputFileName = "encrypted_rsa$i.txt"
>>      Write-Host "Running iteration $i, output file: $outputFileName"
>>      $result = Measure-Command { openssl  pkeyutl -encrypt -pubin -inkey public_key_rsa.pem -in input_rsa.txt -out
 encrypted_rsa$i.txt }
>>      # Display the time taken for each iteration
>>      Write-Host "Iteration $i took $($result.TotalSeconds) seconds"
>>  }
>>
Running iteration 1, output file: encrypted_rsa1.txt
Iteration 1 took 0.0662305 seconds
Running iteration 2, output file: encrypted_rsa2.txt
Iteration 2 took 0.0412114 seconds
Running iteration 3, output file: encrypted_rsa3.txt
Iteration 3 took 0.0339117 seconds
```

And we can see here, that it took ~0.04 seconds to encrypt the file
Memory usage:

```
>>
Peak Working Set Memory: 7712768 bytes
Peak Paged Memory Size: 1622016 bytes
Peak Virtual Memory Size: 4390182912 bytes
PrivateMemorySeze64 Size: 1380352 bytes
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySeze64 Size: $($PrivateMemorySize64) bytes"
>>
Peak Working Set Memory: 6242304 bytes
Peak Paged Memory Size: 1290240 bytes
Peak Virtual Memory Size: 4385611776 bytes
PrivateMemorySeze64 Size: 1290240 bytes
```

Same for decryption

```
PS E:\универ\6 - курс\Крипта\lab1> # Set the number of iterations (replace 10 with your desired number)
>>  $iterations = 1000
>>  # Loop through the command multiple times
>>  for ($i = 1; $i -le $iterations; $i++) {
>>      $outputFileName = "decrypted_rsa$i.txt"
>>      Write-Host "Running iteration $i, output file: $outputFileName"
>>      $result = Measure-Command { openssl  pkeyutl -decrypt -inkey private_key_rsa.pem -in encrypted_rsa.enc -out d
ecrypted_rsa$i.txt }
>>      # Display the time taken for each iteration
>>      Write-Host "Iteration $i took $($result.TotalSeconds) seconds"
>>  }
>>
Running iteration 1, output file: decrypted_rsa1.txt
Iteration 1 took 0.0514238 seconds
Running iteration 2, output file: decrypted_rsa2.txt
Iteration 2 took 0.0274343 seconds
Running iteration 3, output file: decrypted_rsa3.txt
Iteration 3 took 0.0279789 seconds
Running iteration 4, output file: decrypted_rsa4.txt
Iteration 4 took 0.0274293 seconds
```

Time to encrypt ~ 0.035 seconds

Memory used

```
Peak Working Set Memory: 7634944 bytes
Peak Paged Memory Size: 1581056 bytes
Peak Virtual Memory Size: 4390109184 bytes
PrivateMemorySeze64 Size: 1581056 bytes
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySeze64 Size: $($PrivateMemorySize64) bytes"
>>
Peak Working Set Memory: 7774208 bytes
Peak Paged Memory Size: 1572864 bytes
Peak Virtual Memory Size: 4389134336 bytes
PrivateMemorySeze64 Size: 1359872 bytes
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySeze64 Size: $($PrivateMemorySize64) bytes"
>>
Peak Working Set Memory: 4608000 bytes
Peak Paged Memory Size: 1118208 bytes
Peak Virtual Memory Size: 4384563200 bytes
PrivateMemorySeze64 Size: 1110016 bytes
```

Key generation time: 0.724

```
PS E:\универ\6 - курс\Крипта\lab1> Measure-Command {openssl genpkey -algorithm RSA -out private_key.pem}
..+.........+...+..........++++++++++++++++++++++++++++++++++++++++*...+...+.........+..+.+.......+.+...+..+.....
.+...+.........+..+.............+.....+.............++++++++++++++++++++++++++++++++++++++++*...+.+......+.........
...+..+...+......+.+.........+.....+......+.+.................+...+...+.+........+.....+.+...+.....+.......+......+.
.......+...+................+....+.............+.......+.........+.............++++++
.............+.............+..+......+....+..+...+.+......+..........+....++++++++++++++++++++++++++++++++++++*.
.+...+....+...+..+...+.........+......++++++++++++++++++++++++++++++++++++*.................+.......+........+...+
.......+...+...+.....+.+.........+.....+....+...+..+.......................+...+..+....+..+....+.............+......+
..+......+................+......+..+......+.........+...+..........+.......................+...+..+......+.........
.+.......+......+...+.+.....+.+.....+......+....+.....+...+.............+.....+...+......+...+.+.....+.....+...+++++
+

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 724
Ticks             : 7242628
TotalDays         : 8,3826712962963E-06
TotalHours        : 0,00020184111111111
TotalMinutes      : 0,0120710466666667
TotalSeconds      : 0,7242628
TotalMilliseconds : 724,2628
```

sha-256

Secure Hash Algorithm is a cryptographic hash function with a 256-bit output. OpenSSL provides an interface to perform SHA-256 hashing operations, among other cryptographic functions.
SHA-256 takes an input (message) and produces a fixed-size 256-bit (32-byte) hash value, which is typically represented as a hexadecimal string.
It is widely used in various security applications, including digital signatures, certificate authorities, and data integrity verification.

Usage
openssl dgst -sha256 -hex input.txt
input.txt - novel Misto
dgst - The digest functions output the message digest of a supplied file or files in hexadecimal. The digest functions also generate and verify digital signatures using message digests.

The generic name, dgst, may be used with an option specifying the algorithm to be used.

```
PS E:\универ\6 - курс\Крипта\lab1> openssl dgst -sha256 -hex input.txt
SHA2-256(input.txt)= 038d3b396bbd6c5b06d5640e9586629dc7e876583728c04bb5c007b271396218
```

Time & memory measure

DDOS func:

```
 # Set the number of iterations (replace 10 with your desired number)
$iterations = 1000
# Loop through the command multiple times
for ($i = 1; $i -le $iterations; $i++) {
    # Calculate the SHA-256 hash and store it in the $hash variable
    $hash = & openssl dgst -sha256 -hex input.txt
    # Display the time taken for each iteration
    Write-Host "Iteration $i took $($hash.TotalSeconds) seconds"
    # Display the SHA-256 hash
    Write-Host "SHA-256 Hash: $hash"
}
```

```
Iteration 131 took 0.0216625 seconds
Result: 00:00:00.0216625
Iteration 132 took 0.0227428 seconds
Result: 00:00:00.0227428
Iteration 133 took 0.0216201 seconds
Result: 00:00:00.0216201
Iteration 134 took 0.0225828 seconds
Result: 00:00:00.0225828
PS E:\универ\6 - курс\Крипта\lab1> # Set the number of iterations (replace 10 with your desired number)
>> $iterations = 1000
>> # Loop through the command multiple times
>> for ($i = 1; $i -le $iterations; $i++) {
>>     # Calculate the SHA-256 hash and store it in the $hash variable
>>     $hash = & openssl dgst -sha256 -hex input.txt
>>     # Display the time taken for each iteration
>>     Write-Host "Iteration $i took $($hash.TotalSeconds) seconds"
>>     # Display the SHA-256 hash
>>     Write-Host "SHA-256 Hash: $hash"
>> }
Iteration 1 took  seconds
SHA-256 Hash: SHA2-256(input.txt)= 038d3b396bbd6c5b06d5640e9586629dc7e876583728c04bb5c007b271396218
Iteration 2 took  seconds
SHA-256 Hash: SHA2-256(input.txt)= 038d3b396bbd6c5b06d5640e9586629dc7e876583728c04bb5c007b271396218
Iteration 3 took  seconds
SHA-256 Hash: SHA2-256(input.txt)= 038d3b396bbd6c5b06d5640e9586629dc7e876583728c04bb5c007b271396218
Iteration 4 took  seconds
SHA-256 Hash: SHA2-256(input.txt)= 038d3b396bbd6c5b06d5640e9586629dc7e876583728c04bb5c007b271396218
```

Time ~ 0.22 seconds

```
Peak Working Set Memory: 2076672 bytes
Peak Paged Memory Size: 430080 bytes
Peak Virtual Memory Size: 6352896 bytes
PrivateMemorySeze64 Size: 430080 bytes
PS E:\универ\6 - курс\Крипта\lab1> $opensslProcess = Get-Process -Name "openssl"
>> $peakWorkingSet = $opensslProcess.PeakWorkingSet64
>> $peakPagedMemorySize = $opensslProcess.PeakPagedMemorySize64
>> $peakVirtualMemorySize = $opensslProcess.PeakVirtualMemorySize64
>> $PrivateMemorySize64 = $opensslProcess.PrivateMemorySize64
>> "Peak Working Set Memory: $($peakWorkingSet) bytes"
>> "Peak Paged Memory Size: $($peakPagedMemorySize) bytes"
>> "Peak Virtual Memory Size: $($peakVirtualMemorySize) bytes"
>> "PrivateMemorySeze64 Size: $($PrivateMemorySize64) bytes"
>>
Peak Working Set Memory: 2076672 bytes
Peak Paged Memory Size: 425984 bytes
Peak Virtual Memory Size: 6352896 bytes
PrivateMemorySeze64 Size: 425984 bytes
```

PyCryptodome

PyCryptodome stands as a self-contained Python package, delivering an array of low-level cryptographic tools. Supporting Python versions 2.7, 3.5, and newer, as well as PyPy, it distinguishes itself by being an independent Python library, not reliant on external C libraries like OpenSSL. The implementation mainly resides in pure Python, except for crucial performance-centric segments, like block ciphers, which are crafted as C extensions. This approach ensures a comprehensive set of cryptographic primitives while prioritizing flexibility and ease of use for securing data and communications within Python applications.

We utilized Pympler to observe memory usage, a valuable development tool for measuring, monitoring, and analyzing the memory patterns of Python objects within an active Python application. By using Pympler on a Python application, it provides comprehensive insights into the size and lifespan of Python objects, enabling the easy identification of issues such as memory bloat and other unexpected runtime behaviors.

Furthermore, we will utilize the memory-profiler tool to obtain a deeper understanding of our application's memory usage. Memory-profiler is a Python module designed to monitor a process's memory consumption and provide a detailed line-by-line memory consumption analysis for Python programs. It is a pure Python module and relies on the psutil module for its functionality.

Installations:

You can install PyCryptodome using the command line by executing:

```
pip install pycryptodome
```

To install Pympler, use this command in your terminal:

```
pip install pympler
```

To install memory-profiler, you can use this command:

```
pip install memory-profiler
```

AES

AES (Advanced Encryption Standard) stands as a symmetric block cipher standardized by NIST, boasting a set data block size of 16 bytes. The available key lengths for AES are 128, 192, or 256 bits. Known for its exceptional speed and high security, AES has become the widely accepted standard for symmetric encryption.

Time & memory measure:

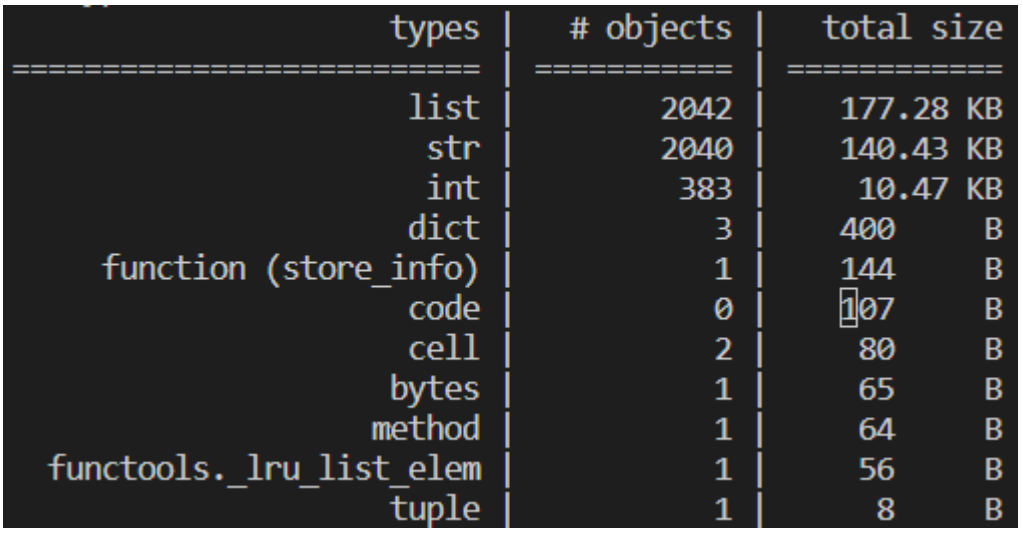
In PyCryptodome, utilizing AES, we can encrypt a .txt file containing approximately 520,523 characters in an average of around 0.006 seconds, and decrypt it in a similar time frame on average.

```
Encryption Time: 0.006006428480148319 seconds
Decryption Time: 0.0064232490062713 58 seconds
```

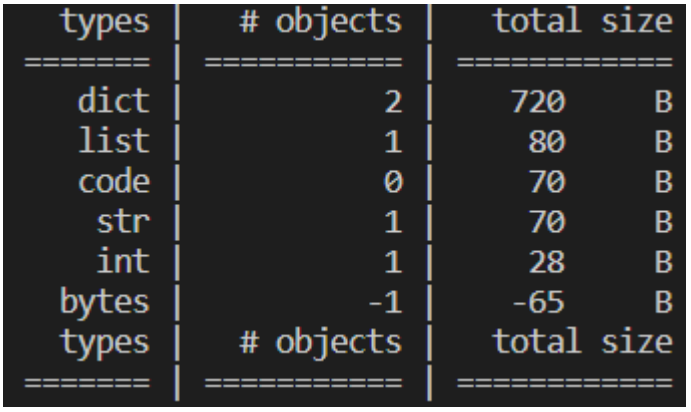
Memory profiling in the case of PyCryptodome's AES shows that:

- There are numerous `list` and `str` objects that consume a substantial amount of memory. These may correspond to the internal workings of the library where lists could be used to manage blocks of data during encryption and decryption, and strings could be involved in handling the keys or intermediate representations of the data.
- The presence of `functools._lru_list_elem` objects indicates that some form of caching is being used, which is typical for functions that need to maintain state or for performance optimizations.
- The memory used by `bytes` decreases over time, which suggests that the encryption and decryption processes create temporary byte arrays that are later garbage collected after use.

From the memory profiling, it's evident that the PyCryptodome AES operations are memory-intensive processes, creating and disposing of several objects within the process. The EAX mode, due to its nature of providing authentication in addition to encryption, might add to the memory overhead compared to simpler modes that provide encryption without authentication.



types	# objects	total size
list	2042	177.28 KB
str	2040	140.43 KB
int	383	10.47 KB
dict	3	400 B
function (store_info)	1	144 B
code	0	107 B
cell	2	80 B
bytes	1	65 B
method	1	64 B
functools._lru_list_elem	1	56 B
tuple	1	8 B



types	# objects	total size
dict	2	720 B
list	1	80 B
code	0	70 B
str	1	70 B
int	1	28 B
bytes	-1	-65 B
types	# objects	total size

The memory profiling for AES encryption/decryption in Python shows stable memory usage with no leaks over 1000 iterations, and negative memory increments likely due to garbage collection. This indicates that the PyCryptodome AES implementation is memory-efficient.

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    18     22.7 MiB     22.7 MiB           1   @profile
    19                                         def main():
    20     23.7 MiB      0.0 MiB           2       with open('input.txt', 'r', encoding='utf-8') as file:
    21     23.7 MiB      1.0 MiB           1           plaintext = file.read()
    22
    23     23.7 MiB      0.0 MiB           1       encryption_time = 0
    24     23.7 MiB      0.0 MiB           1       decryption_time = 0
    25
    26     27.4 MiB   -980.7 MiB        1001       for i in range(1000):
    27     27.4 MiB   -978.9 MiB        1000           key = get_random_bytes(32)
    28
    29     27.4 MiB   -978.9 MiB        1000           start_time = time.time()
    30     27.4 MiB   -744.8 MiB        1000           cipher, ciphertext, tag = encrypt_data(plaintext, key)
    31     27.4 MiB  -1077.3 MiB        1000           end_time = time.time()
    32
    33     27.4 MiB  -1077.3 MiB        1000           execution_time = end_time - start_time
    34     27.4 MiB  -1077.3 MiB        1000           encryption_time = encryption_time + (execution_time - encryption_time) / (i + 1)
    35
    36     27.4 MiB  -1077.3 MiB        1000           start_time = time.time()
    37     27.4 MiB  -1309.5 MiB        1000           decrypted_text = decrypt_data(cipher, ciphertext, tag, key)
    38     27.4 MiB   -980.7 MiB        1000           end_time = time.time()
    39
    40     27.4 MiB   -980.7 MiB        1000           execution_time = end_time - start_time
    41     27.4 MiB   -980.7 MiB        1000           decryption_time = decryption_time + (execution_time - decryption_time) / (i + 1)
    42
    43     25.6 MiB     -1.8 MiB           1       print(f"Encryption Time: {encryption_time} seconds")
    44     25.6 MiB      0.0 MiB           1       print(f"Decryption Time: {decryption_time} seconds")
```

SHA256

SHA-256, a member of the SHA-2 family of cryptographic hash functions, generates a 256-bit hash value from an input message. It is important to note that SHA-256 is susceptible to length-extension attacks. This vulnerability is particularly concerning when the hash involves a secret message.

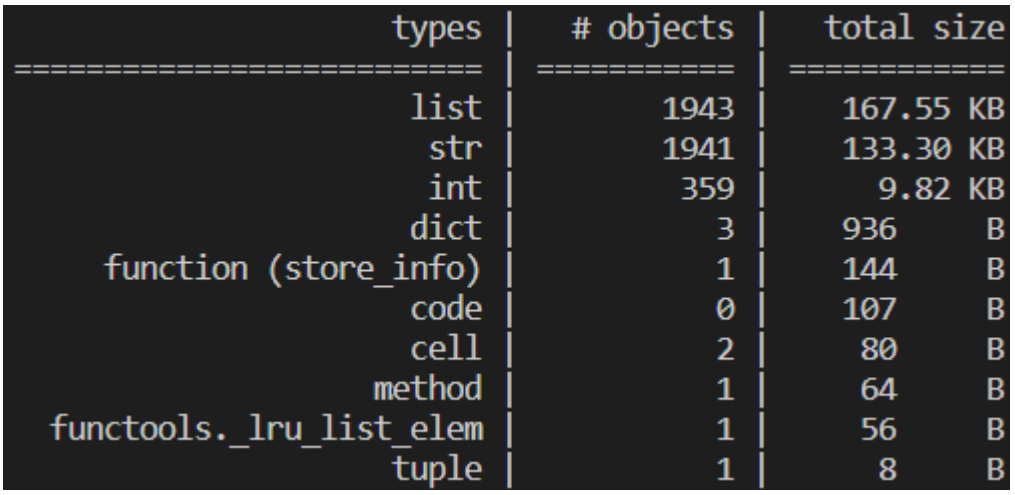
Time & memory measure:

In PyCryptodome, utilizing SHA-256, we can create a hash for a .txt file containing approximately 520,523 characters in an average of around 0.006 seconds.

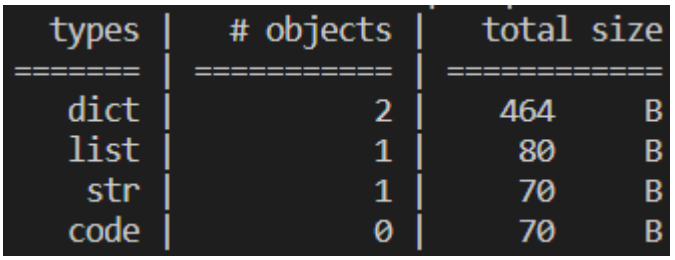
```
Time: 0.006276529788970951 seconds
SHA-256 Hash: 038d3b396bbd6c5b06d5640e9586629dc7e876583728c04bb5c007b271396218
```

According to the Pympler memory tracker results, the program initially allocates significant memory across various object types, notably lists and strings, which consume approximately 167.55 KB and 133.30 KB, respectively. After the first memory tracking checkpoint, there is an observed allocation of additional memory, primarily in strings and integers, which is likely associated with the hashing process.

Upon rehashing, the memory tracker indicates a modest increase in memory consumption, particularly in dictionaries and strings. This suggests that both the hash recalculation and the subsequent tracking of memory modifications are efficiently managed operations. The script demonstrates effective memory management during hash computation, as evidenced by the absence of significant memory leaks or indirect allocations between the two checkpoints.



types	# objects	total size
list	1943	167.55 KB
str	1941	133.30 KB
int	359	9.82 KB
dict	3	936 B
function (store_info)	1	144 B
code	0	107 B
cell	2	80 B
method	1	64 B
functools._lru_list_elem	1	56 B
tuple	1	8 B



types	# objects	total size
dict	2	464 B
list	1	80 B
str	1	70 B
code	0	70 B

The memory profiling indicates efficient and stable memory usage. Initially, the program uses 20.8 MiB, which increases by a mere 1.0 MiB after reading the input file. Throughout the hashing process loop, the memory usage remains consistent around 21.9 MiB, with no significant increases, suggesting no memory leaks. Post-computation, the memory footprint returns to 21.8 MiB, confirming effective memory management.

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    11     20.8 MiB     20.8 MiB           1   @profile
    12                                         def main():
    13     21.8 MiB      0.0 MiB           2       with open('input.txt', 'r', encoding='utf-8') as file:
    14     21.8 MiB      1.0 MiB           1           plaintext = file.read()
    15
    16     21.8 MiB      0.0 MiB           1       hash_time = 0
    17
    18     21.9 MiB    -16.9 MiB       10001       for i in range(10000):
    19     21.9 MiB    -16.8 MiB       10000           start_time = time.time()
    20     21.9 MiB    -16.8 MiB       10000           hash_result = calculate_sha256_hash(plaintext)
    21     21.9 MiB    -16.9 MiB       10000           end_time = time.time()
    22
    23     21.9 MiB    -16.9 MiB       10000           execution_time = end_time - start_time
    24     21.9 MiB    -16.9 MiB       10000           hash_time = hash_time + (execution_time - hash_time) / (i + 1)
    25
    26     21.8 MiB     -0.0 MiB           1       print(f"Time: {hash_time} seconds")
    27     21.8 MiB      0.0 MiB           1       print(f"SHA-256 Hash: {hash_result}")
```

RSA

RSA (Rivest-Shamir-Adleman) is the most widespread and used public key algorithm. Its security is based on the difficulty of factoring large integers. The algorithm has withstood attacks for more than 30 years, and it is therefore considered reasonably secure for new designs. The algorithm can be used for both confidentiality (encryption) and authentication (digital signature). It is worth noting that signing and decryption are significantly slower than verification and encryption. The cryptographic strength is primarily linked to the length of the RSA modulus *n*.

Time & memory measure:

The RSA key generation process averages approximately 0.990 seconds, indicating the computational intensity of generating a secure key pair. Encryption using the RSA algorithm with PKCS1_OAEP padding is notably faster, averaging around 0.001 seconds. Decryption requires slightly more time, averaging about 0.004 seconds.

```
RSA Key Generation Time: 0.9903863096237179 seconds
RSA Encryption Time: 0.0007824563980102539 seconds
RSA Decryption Time: 0.004362616539001466 seconds
Decrypted Text: This is a secret message.
```

The memory profiling results reveal a substantial allocation for list and string types, with 2139 list objects occupying 184.86 KB and 2137

string objects using 147.25 KB of memory. Other object types like integers and dictionaries are using significantly less memory.

As operations progress, the number of objects and total memory size for each type decrease dramatically. After RSA key generation, only two dictionary objects remain, consuming 720 B of memory, and after encryption and decryption, the memory usage is minimized even further, indicating no significant memory leaks and that the garbage collector is effectively reclaiming memory.

types	# objects	total size
list	2139	184.86 KB
str	2137	147.25 KB
int	413	11.29 KB
dict	3	400 B
function (store_info)	1	144 B
code	0	107 B
cell	2	80 B
method	1	64 B
functools._lru_list_elem	1	56 B
tuple	1	8 B

types	# objects	total size
dict	2	720 B
list	1	80 B
str	1	70 B
code	0	70 B

| types | # objects | total size |

types	# objects	total size

Memory profiling in the case of PyCryptodome's RSA shows that:

- The memory usage starts at around 22.9 MiB and remains fairly constant throughout the RSA key generation, encryption, and decryption loop.
- There are numerous entries with large negative increments (e.g., -5252.1 MiB). These negative values are not typical

and likely represent a reporting error from the memory profiling tool or a visual bug in the output.

- After the loop has completed (the loop with 1000 iterations), the memory usage drops by 10.2 MiB to 12.9 MiB. This decrease could indicate that a large number of temporary objects created during the loop have been deallocated.
- The ‘Increment’ column shows that the memory increase per operation within the loop is essentially zero, indicating that the memory footprint for each iteration remains stable and there is no memory leak associated with each individual encryption/decryption cycle.
- The occurrences count at 1000 for the loop lines suggests that the profiler has correctly tracked the number of iterations.

Overall, apart from the anomalous negative increments, the stable memory usage implies that the RSA operations are not causing any sustained increase in memory usage.

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    20     22.9 MiB     22.9 MiB           1   @profile
    21                                         def main():
    22     22.9 MiB      0.0 MiB           1       plaintext = "This is a secret message."
    23
    24     22.9 MiB      0.0 MiB           1       generation_time = 0
    25     22.9 MiB      0.0 MiB           1       encryption_time = 0
    26     22.9 MiB      0.0 MiB           1       decryption_time = 0
    27
    28     23.2 MiB  -5252.1 MiB        1001       for i in range(1000):
    29     23.2 MiB  -5241.8 MiB        1000           start_time = time.time()
    30     23.2 MiB  -5251.8 MiB        1000           rsa_key = generate_rsa_key_pair()
    31     23.2 MiB  -5252.0 MiB        1000           end_time = time.time()
    32     23.2 MiB  -5252.0 MiB        1000           execution_time = end_time - start_time
    33     23.2 MiB  -5252.0 MiB        1000           generation_time = generation_time + (execution_time - generation_time) / (i + 1)
    34
    35     23.2 MiB  -5252.0 MiB        1000           start_time_encryption = time.time()
    36     23.2 MiB  -5251.9 MiB        1000           ciphertext = rsa_encrypt(plaintext, rsa_key.publickey())
    37     23.2 MiB  -5251.9 MiB        1000           end_time_encryption = time.time()
    38     23.2 MiB  -5251.9 MiB        1000           execution_time = end_time_encryption - start_time_encryption
    39     23.2 MiB  -5251.9 MiB        1000           encryption_time = encryption_time + (execution_time - encryption_time) / (i + 1)
    40
    41     23.2 MiB  -5251.9 MiB        1000           start_time_decryption = time.time()
    42     23.2 MiB  -5252.1 MiB        1000           decrypted_text = rsa_decrypt(ciphertext, rsa_key)
    43     23.2 MiB  -5252.1 MiB        1000           end_time_decryption = time.time()
    44     23.2 MiB  -5252.1 MiB        1000           execution_time = end_time_decryption - start_time_decryption
    45     23.2 MiB  -5252.1 MiB        1000           decryption_time = decryption_time + (execution_time - decryption_time) / (i + 1)
    46
    47     12.9 MiB    -10.2 MiB           1       print(f"RSA Key Generation Time: {generation_time} seconds")
    48     12.9 MiB      0.0 MiB           1       print(f"RSA Encryption Time: {encryption_time} seconds")
    49     12.9 MiB      0.0 MiB           1       print(f"RSA Decryption Time: {decryption_time} seconds")
    50     12.9 MiB      0.0 MiB           1       print(f"Decrypted Text: {decrypted_text}")
```

Other libraries

In this part, we will look at the use of popular encryption algorithms using other cryptographic libraries in Python.

- *python rsa;*
- *hashlib;*
- *cryptography.*

RSA

Each pair of the RSA algorithm has two keys, i.e. a public key and a private key. One key is used for encrypting the message which can only be decrypted by the other key.

```
Encryption time: 0.0019946009832763672 seconds
Decryption time: 0.5595366954803467 seconds
Filename: rsa.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     6   41.586 MiB   41.586 MiB           1   @profile
     7                                         def main_rsa():
     8   41.602 MiB    0.016 MiB           1       fileObj = codecs.open( "input_rsa.txt", "r", "utf_8_sig" )
     9   41.602 MiB    0.000 MiB           1       text_rsa = fileObj.read() # или читайте по строке
    10   41.602 MiB    0.000 MiB           1       fileObj.close()
    11
    12   41.602 MiB    0.000 MiB           1       key_pair = rsa.generate_key_pair(1024)
    13
    14   41.602 MiB    0.000 MiB           1       start = time.time()
    15   41.613 MiB    0.012 MiB           1       cipher = rsa.encrypt(text_rsa, key_pair["public"], key_pair["modulus"
])
    16   41.613 MiB    0.000 MiB           1       end = time.time()
    17   41.613 MiB    0.000 MiB           1       encryption_time = end - start
    18
    19   41.613 MiB    0.000 MiB           1       start = time.time()
    20   41.613 MiB    0.000 MiB           1       decrypted_message = rsa.decrypt(cipher, key_pair["private"], key_pair
["modulus"])
    21   41.613 MiB    0.000 MiB           1       end = time.time()
    22   41.613 MiB    0.000 MiB           1       decryption_time = end - start
    23
    24   41.617 MiB    0.004 MiB           1       print(f"Encryption time: {encryption_time} seconds")
    25   41.617 MiB    0.000 MiB           1       print(f"Decryption time: {decryption_time} seconds")
```

SHA256

Python has a built-in library, `hashlib`, that is designed to provide a common interface to different secure hashing algorithms. The module provides constructor methods for each type of hash. For example, the `.sha256()` constructor is used to create a SHA256 hash.

The `sha256` constructor takes a byte-like input, returning a hashed value. The function has a number of associated with hashing values, which are especially useful given that normal strings can’t easily be processed:

- `encode()` which is used to convert a string to bytes, meaning that the string can be passed into the `sha256` function
- `hexdigest()` which is used to convert our data into hexadecimal format

```
Hash: 038d3b396bbd6c5b06d5640e9586629dc7e876583728c04bb5c007b271396218
Hash time: 0.0029845237731933594 seconds
Filename: sha.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     6   41.664 MiB   41.664 MiB           1   @profile
     7                                         def main_sha():
     8   41.680 MiB    0.016 MiB           1       fileObj = codecs.open( "input.txt", "r", "utf_8_sig" )
     9   42.621 MiB    0.941 MiB           1       text = fileObj.read() # или читайте по строке
    10   42.621 MiB    0.000 MiB           1       fileObj.close()
    11
    12   42.621 MiB    0.000 MiB           1       start = time.time()
    13   42.645 MiB    0.023 MiB           1       res = sha256(text.encode('utf-8')).hexdigest()
    14   42.645 MiB    0.000 MiB           1       end = time.time()
    15   42.645 MiB    0.000 MiB           1       hash_time = end - start
    16
    17
    18   42.645 MiB    0.000 MiB           1       print(f"Hash: {res}")
    19   42.648 MiB    0.004 MiB           1       print(f"Hash time: {hash_time} seconds")
```

AES

The `cryptography.hazmat.primitives.ciphers` module provides various symmetric encryption algorithms such as AES, DES, and Blowfish. These algorithms allow for encryption and decryption of messages using a shared secret key.

```
Encryption time: 0.7322797775268555 seconds
Decryption time: 0.0029892921447753906 seconds
Filename: aes.py
```

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     6   41.723 MiB   41.723 MiB           1   @profile
     7                                         def main_aes():
     8   41.738 MiB    0.016 MiB           1       fileObj = codecs.open( "input.txt", "r", "utf_8_sig" )
     9   42.668 MiB    0.930 MiB           1       text = fileObj.read() # или читайте по строке
    10   42.668 MiB    0.000 MiB           1       fileObj.close()
    11
    12   43.562 MiB    0.895 MiB           1       new_text = bytes(text + " " * 9,'UTF-8')
    13   43.562 MiB    0.000 MiB           1       key = os.urandom(32)
    14   43.562 MiB    0.000 MiB           1       iv = os.urandom(16)
    15   43.562 MiB    0.000 MiB           1       cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    16
    17   43.562 MiB    0.000 MiB           1       start = time.time()
    18   47.586 MiB    4.023 MiB           1       encryptor = cipher.encryptor()
    19   48.465 MiB    0.879 MiB           1       ct = encryptor.update(new_text) + encryptor.finalize()
    20   48.465 MiB    0.000 MiB           1       end = time.time()
    21   48.465 MiB    0.000 MiB           1       encryption_time = end - start
    22
    23   48.465 MiB    0.000 MiB           1       start = time.time()
    24   48.465 MiB    0.000 MiB           1       decryptor = cipher.decryptor()
    25   49.367 MiB    0.902 MiB           1       dec = decryptor.update(ct) + decryptor.finalize()
    26   50.363 MiB    0.996 MiB           1       text2 = dec.decode('utf-8')
    27   50.363 MiB    0.000 MiB           1       end = time.time()
    28   50.363 MiB    0.000 MiB           1       decryption_time = end - start
    29
    30   50.367 MiB    0.004 MiB           1       print(f"Encryption time: {encryption_time} seconds")
    31   50.367 MiB    0.000 MiB           1       print(f"Decryption time: {decryption_time} seconds")
```

Conclusions

In this work, we conducted a comparative analysis of RSA, AES-256, and SHA-256 cryptographic algorithms implemented in the OpenSSL and PyCryptodome libraries. Our focus was on evaluating performance in terms of execution time and memory usage.

OpenSSL Library:

1. AES-256:
    - Encryption and decryption times were approximately 0.079 seconds and 0.084 seconds, respectively.
    - Memory usage was around 430000 bytes for both encryption and decryption.
2. RSA:
    - RSA key generation took 0.724 seconds.
    - Encryption and decryption times were about 0.04 seconds each.
    - Memory usage was approximately 1300000 bytes for encryption and 1350000 bytes for decryption.
3. SHA-256:

- Hashing time was about 0.22 seconds.
- Memory usage averaged at ~427000 bytes.

PyCryptodome Library:

1. AES:
    - AES encryption and decryption were remarkably fast at around 0.006 seconds.
    - Memory profiling indicated substantial use of list and string objects.
2. SHA-256:
    - Efficient hashing process, taking about 0.006 seconds.
    - Memory usage was stable, with an initial allocation mainly in strings and integers and modest increases upon rehashing.
3. RSA:
    - RSA key generation was slower at 0.990 seconds.
    - Encryption and decryption were much faster at 0.001 seconds and 0.004 seconds, respectively.
    - Memory profiling showed a stable memory usage of around 22.9 MiB, with no significant leaks.

Other Python Libraries:

1. AES:
    - AES encryption is about 0.7 seconds, and decryption is ~0.002 seconds.
2. SHA-256:
    - SHA-256 hashing takes ~0.002 seconds.
3. RSA:
    - Encryption took ~0.001 seconds and decryption took ~0.55 seconds.

Comparison and Conclusions:

- Execution Time: PyCryptodome outperforms OpenSSL in terms of speed, especially in AES and SHA-256 operations. However, RSA key generation in PyCryptodome is slower compared to OpenSSL.

- Memory Usage: If estimate the memory consumption of Python libraries based on the cost of the interpreter for executing a line of code, and OpenSSL estimates the memory consumption for executing the process, then based on the results, it can be seen that the PyCryptodome library used an average of 22 times more memory, and the other libs used 44 times more memory.
- Overall Assessment: PyCryptodome excels in speed for AES and SHA-256 and demonstrates efficient memory usage across all operations. OpenSSL, while slightly slower in AES and SHA-256, performs well in RSA key generation and maintains reasonable memory usage. The choice between the two libraries may depend on specific requirements for speed versus memory efficiency in cryptographic operations.

In conclusion, both OpenSSL and PyCryptodome libraries offer robust cryptographic functions, each with its own strengths. PyCryptodome stands out for its speed in AES and SHA-256 algorithms, while OpenSSL is more efficient in RSA key generation. But in the memory case, the Python interpreter eats too much memory, unlike the OpenSSL library. The decision on which library to use should be guided by the specific cryptographic needs and constraints of the application.