Cryptography - Lab 5 - Asymmetric Crypto

Javier Antxon Garrues Apecechea

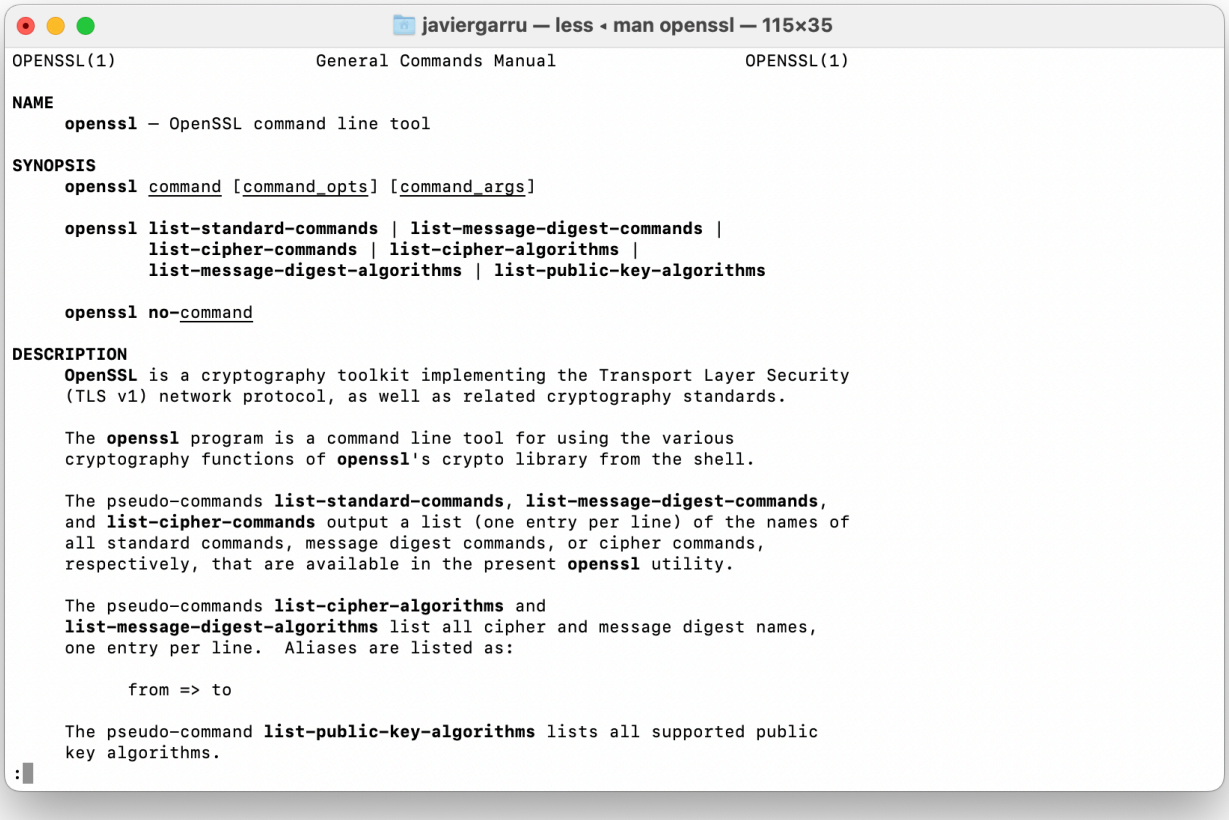
Student ID: 24647808

In this lab we will get familiar with Diffie Hellman Key Exchange and the RSA algorithm.

Part Two: openssl

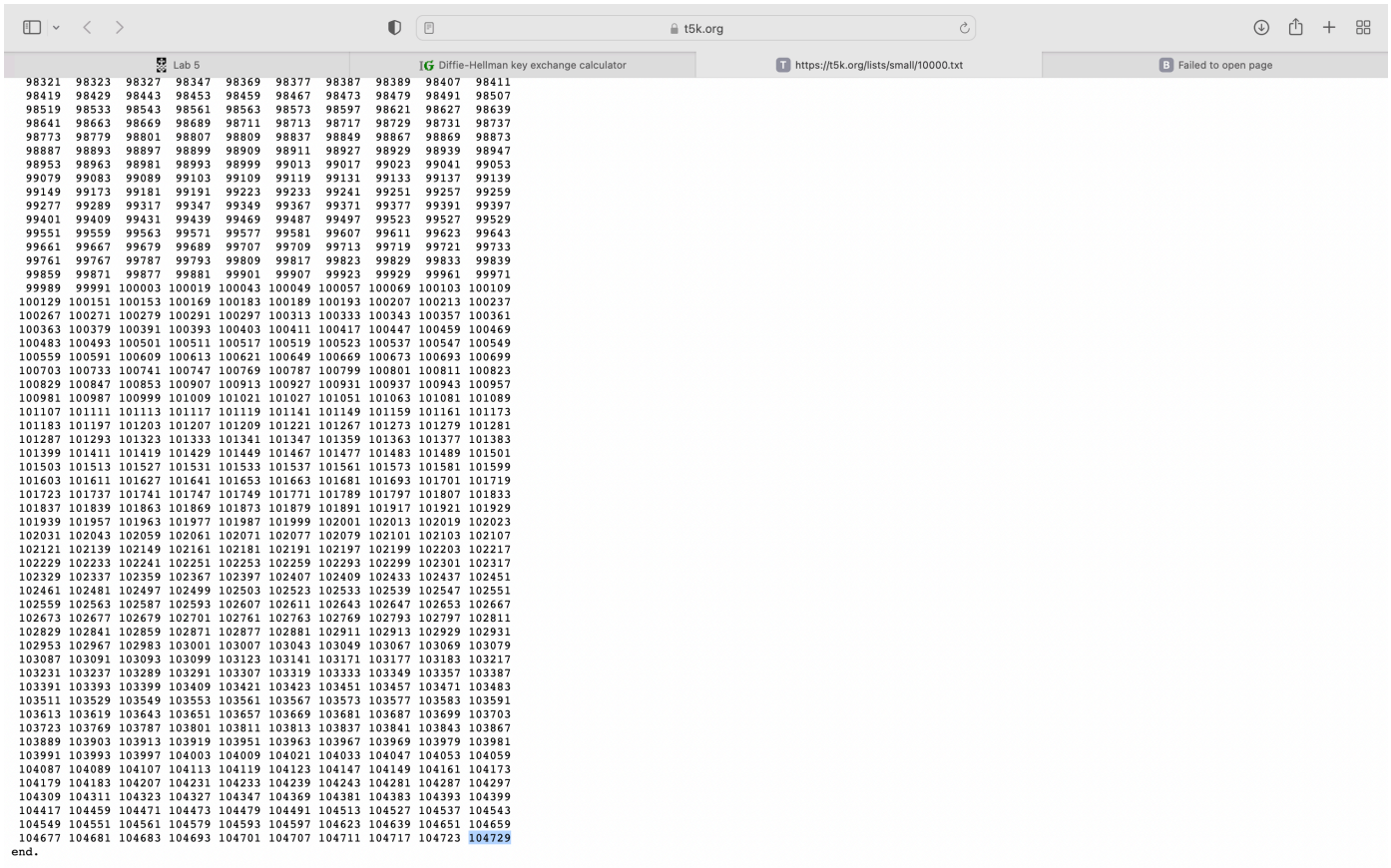
In previous labs we had used this same tool so there is no need to download it again. We can have a look at the manual to check the command options:

```
man openssl
```



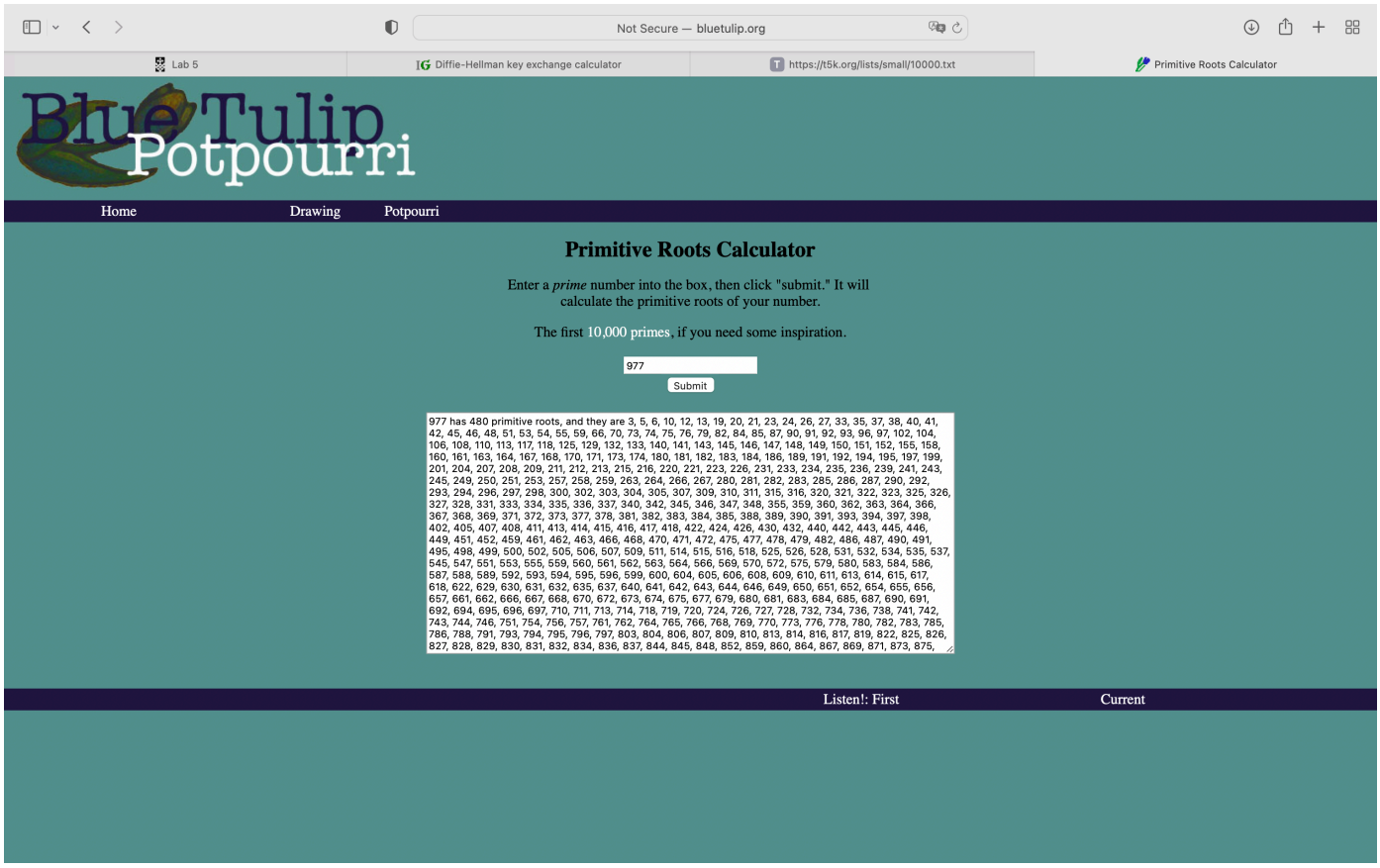
Part Three: Diffie Hellman Key Exchange

We select a prime number from the following website `https://t5k.org/lists/small/10000.txt`. In our case, we have firstly selected the last one available **104729**, but it is too big and the next step takes longer than expected. For this reason, we changed and chose **977**.

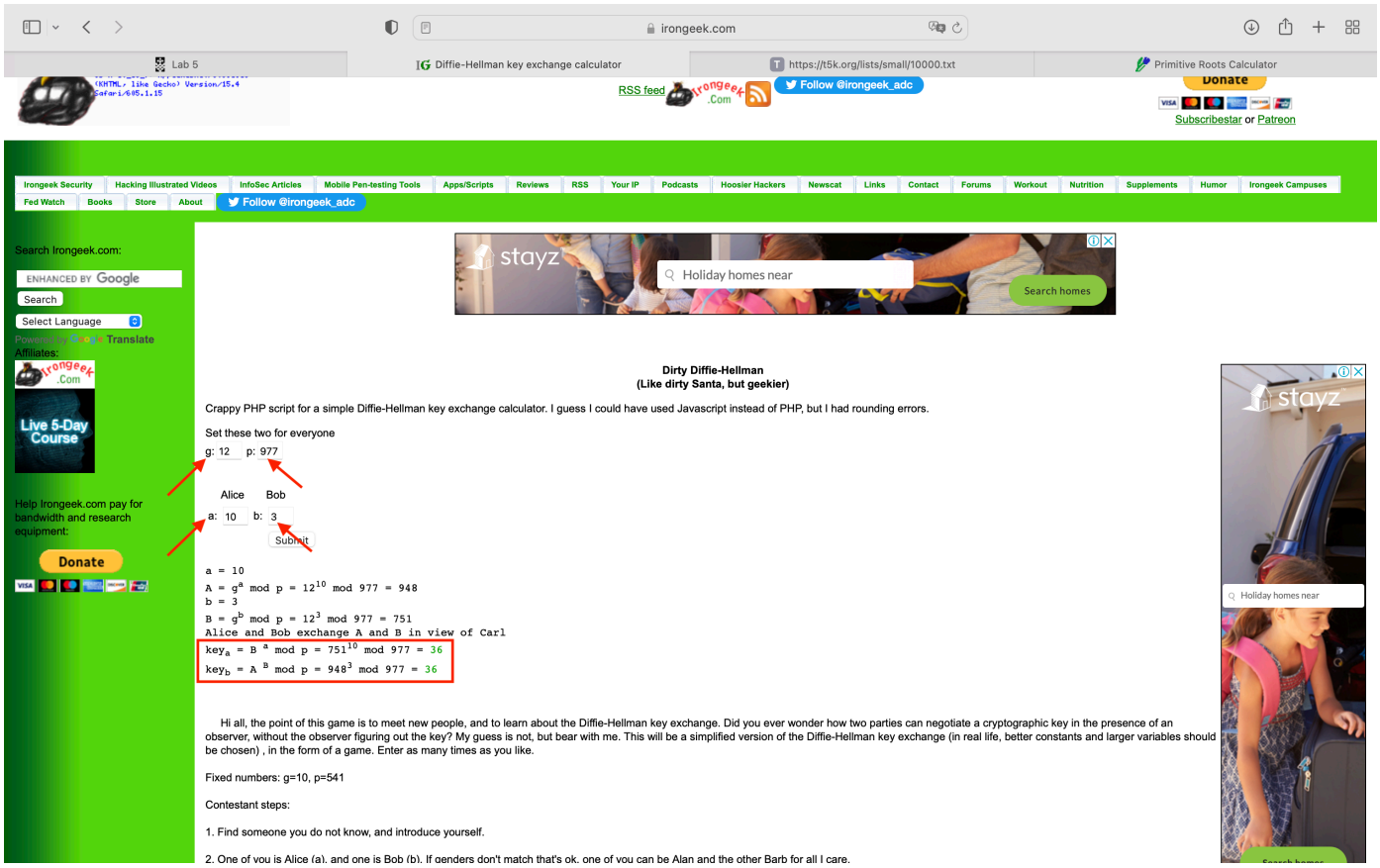


Then, we go to the following website `http://www.bluetulip.org/2014/programs/primitive.html` and introduce the chosen prime. This website returns all the primitive roots of our number. For 977 there are 480 primitive roots. Remember that a **primitive root of** a prime number **p** is another number **g** such that all the numbers **1,2,...,p-1** can be obtained through **gn mod p** with n being any number. For instance, 2 is a primitive root of 5:

- 1 = 2*8 mod 5
- 2 = 2*11 mod 5
- 3 = 2*9 mod 5
- 4 = 2*12 mod 5



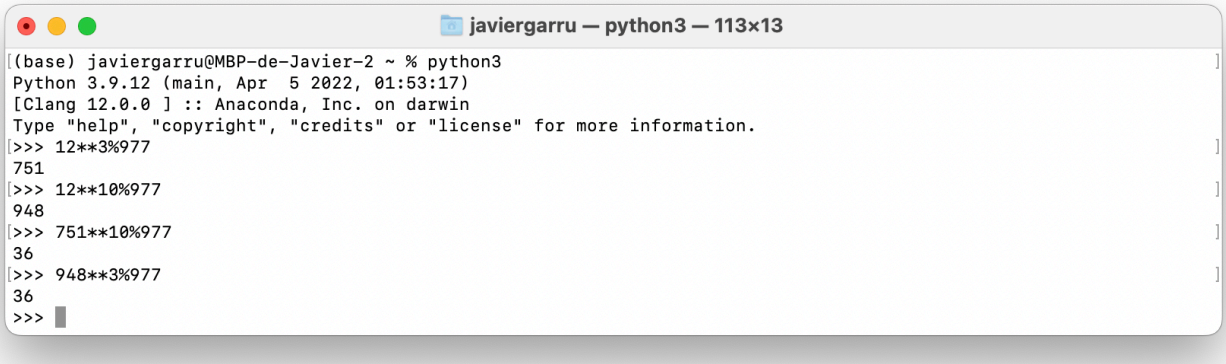
And we **select** one from this list such as **12**. Now, we go to the following site `https://www.irongeek.com/diffie-hellman.php` and introduce the selected values. Apart from those values, we also choose one value for Alice and another for Bob. We will choose **10** and **3**.



A summary of the process can be seen here:

- A prime an one primitive root of the prime are known
- Alice chooses a random number
- Bob chooses a random number
- Alice calculates the primitive root powered to her chosen number modulo the primenumber
- Bob calculates the primitive root powered to his chosen number modulo the prime number
- Alice and Bob exchange their calculated values
- Alice powers Bob's value to her secret number and finds its modulo prime number
- Bob powers Alice's value to his secret number and finds its modulo prime number
- Alice and Bob now share the same secret key without having to exchange it

We can also check the process and calculations manually with python:



Part Four: Manual RSA Encryption and Decryption

4.1.Deriving the private key:

We download the file **RSA.py** and note the values given in the lab document:

- p = 191
- q = 281
- e = 33
- e x d = 1 mod (phi(n))
- phi(n) = (p-1)*(q-1)

Public key: (e,n)

Private key: (d,n)

Question: what is the numerical values of the private and public key?

We execute the program and obtain the results:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [31644]
decrypt [200]
(base) javiergarru@MBP-de-Javier-2 w8 %
```

Public key: (33, 53671)

Private key: (12897,53671)

Summary of the steps taken to obtain the keys:

1. Choose two prime numbers: p and q
2. Multiply the two numbers: n = p*q
3. Calculate the totient of n: phi(n) = (p-1)*(q-1)
4. Select a random relative prime of the totient: gcd(e,n) = 1
5. Find the multiplicative inverse modulo n of e: d*e mod n = 1
6. Generate public key (e,n) and private key (d,n)

4.2.Encrypting a message:

Question: what is the cipher-text of 100?

The ciphertext of 100 is 31644.

Question: try to encrypt a big number and describe your findings

We change the 100 to **10000** in the code and execute the program. In addition to this, we also change the ciphertext we want to decrypt to check that everything works correctly:

```
81   print ("public key",pub)
82   print ("private key",pri)
83   print ("encrypt",encrypt(pub,[10000]))
84   print ("decrypt",decrypt(pri,[2889]))
85
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [2889]
decrypt [10000]
(base) javiergarru@MBP-de-Javier-2 w8 %
```

Let's now try with a bigger number like **100000**:

```
81    print ("public key",pub)
82    print ("private key",pri)
83    print ("encrypt",encrypt(pub,[100000]))
84    print ("decrypt",decrypt(pri,[47736]))
85
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [47736]
decrypt [46329]
(base) javiergarru@MBP-de-Javier-2 w8 %
```

As we can see, the encryption seems to work well, but when we decrypt the obtained ciphertext (47736) we don't obtain the original value. The reason behind this problem is that we are working with a relatively low n (53671). The encryption process in RSA consists of powering the introduced number (100000) to the first value of the public key and finding its module n. The operation would look something like this:

Plaintext_value^e mod n = ciphertext_value

100000^33 mod 53671 = 47736

Because 100000 is greater than 53671 and, using the modulo properties, we know that 100000^33 mod 53671 is the same as (100000 mod 53671)^33 mod 53671, which is the same as (46329)^33 mod 53671. And that is how we would obtain the ciphertext associated to 46329, which means that the ciphertext that we obtained for 1000000 is indeed the one associated to another number. Let's see this with the program:

```
69  def encrypt(key, bstr):
70      return [rsa(key,c) for c in bstr]
71
72  #"Not everything that can be counted counts, and not everyth
73  #"Make everything as simple as possible, but not simpler." -
74  #"There are only two ways to live your life. One is as thoug
75
76  # Decrypt. Takes a key and an array of numbers.
77  # Runs RSA on each of the values. Returns a byte string.
78  def decrypt(key, arr):
79      return [rsa(key,a) for a in arr]
80
81  print ("public key",pub)
82  print ("private key",pri)
83  print ("encrypt",encrypt(pub,[46329]))
84  print ("decrypt",decrypt(pri,[47736]))
85
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [47736]
decrypt [46329]
(base) javiergarru@MBP-de-Javier-2 w8 %
```

As we can see the encryption and decryption of the value now works as expected and the ciphertext obtained by encrypting 46329 is the same as the one obtained when encrypting 100000.

4.3.Decrypting a message:

C = 11759

Question: what is the plain-text of C?

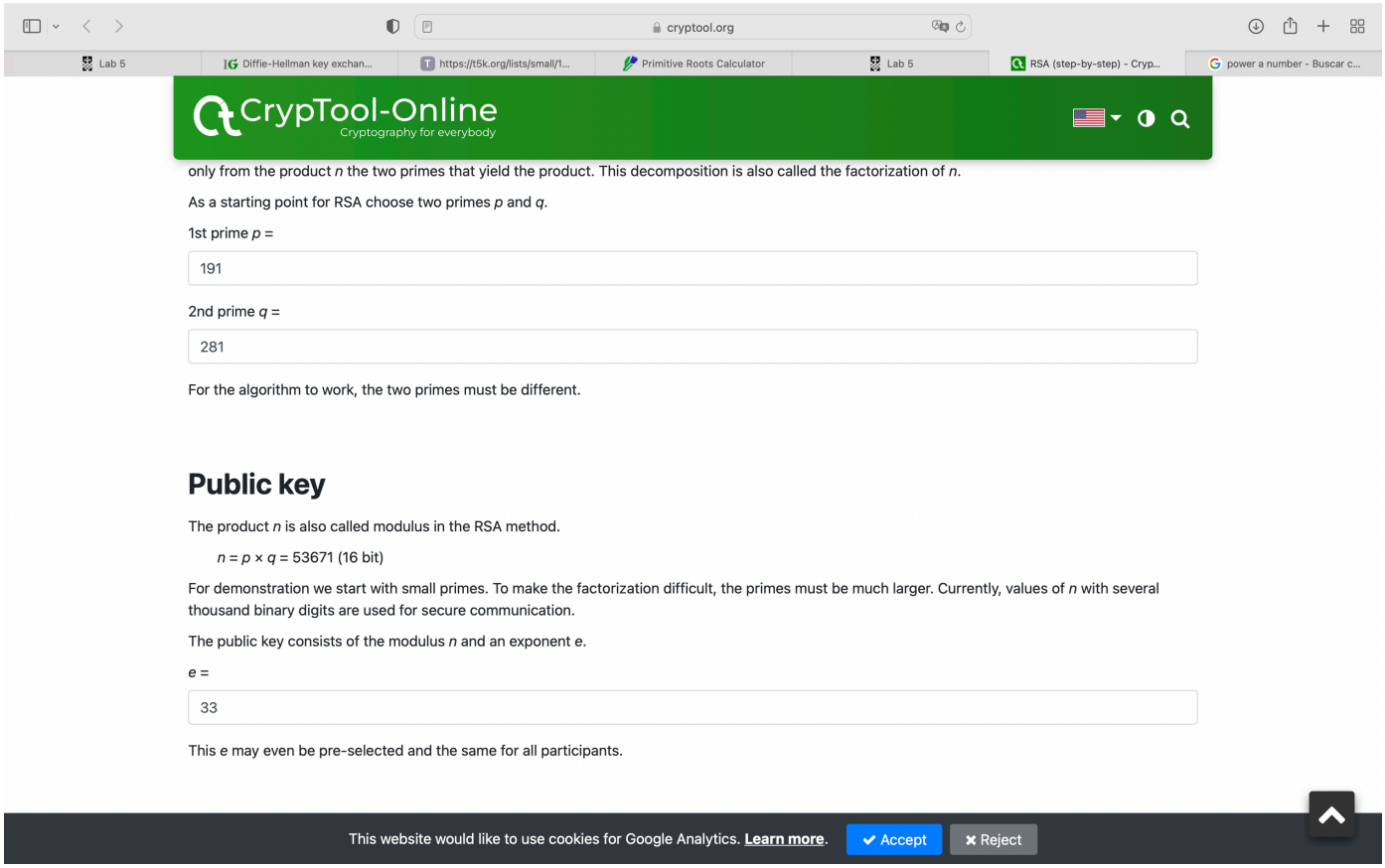
We execute the code again and see that the associated decryption is **200**:

```
75
76  # Decrypt. Takes a key and an array of numbers.
77  # Runs RSA on each of the values. Returns a byte string.
78  def decrypt(key, arr):
79      return [rsa(key,a) for a in arr]
80
81  print ("public key",pub)
82  print ("private key",pri)
83  print ("encrypt",encrypt(pub,[200]))
84  print ("decrypt",decrypt(pri,[11759]))
85
```

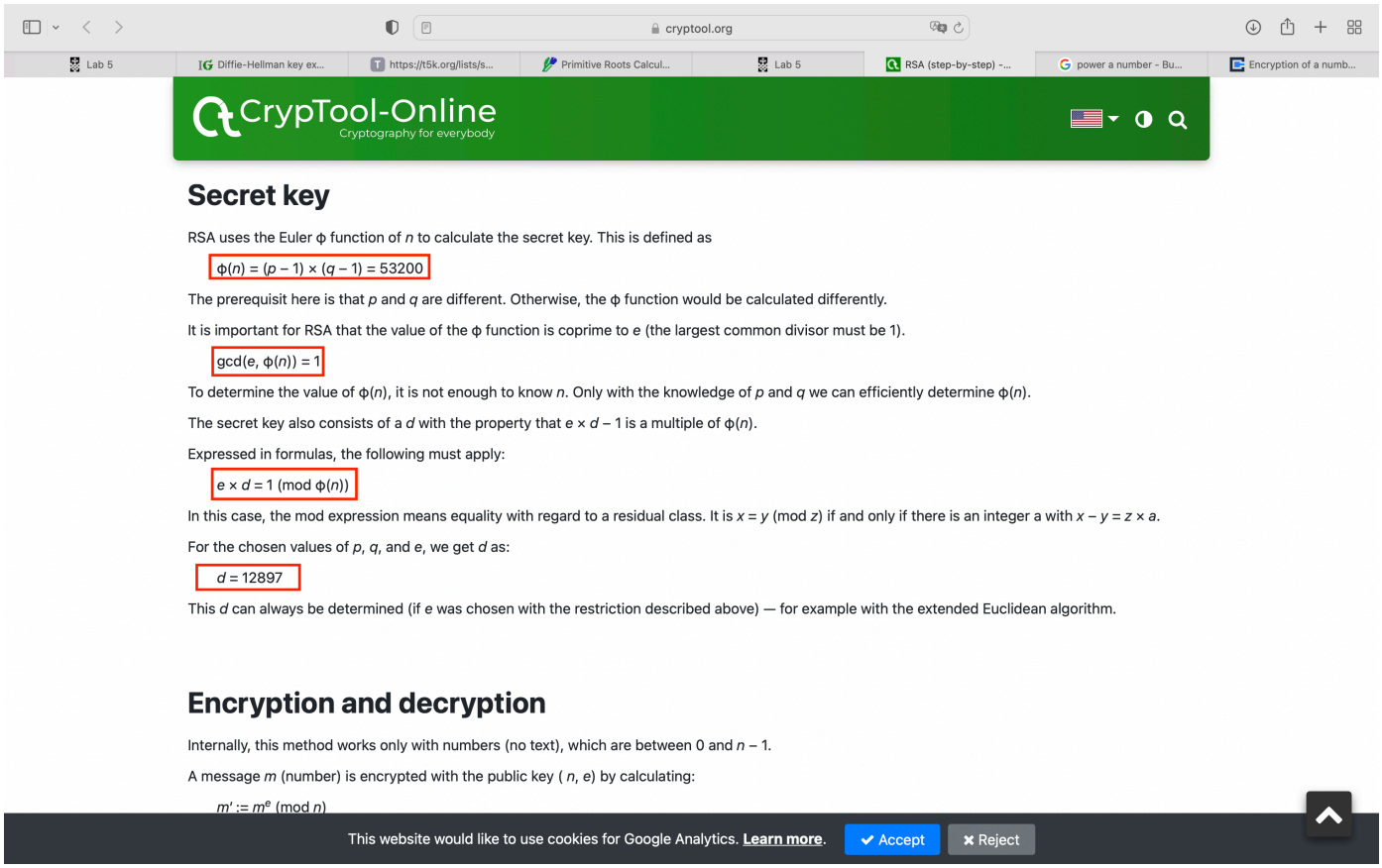
```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    JUPYTER

(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [11759]
decrypt [200]
(base) javiergarru@MBP-de-Javier-2 w8 %
```

4.4.RSA key generation and encryption with an online tool:

Now, we can visit the following website `https://www.cryptool.org/en/cto/rsa-step-by-step.html` and introduce the same values and see how the different keys are generated:



Here we can see how p, q and e are introduced and n is calculated. Let's now see how the keys are generated with these values:



The euler function is used to calculated the number of relative primes smaller than the value n:

phi(n) = phi(p*q) = phi(p)*phi(q) = (p-1)*(q-1) =190 * 280 = 53200

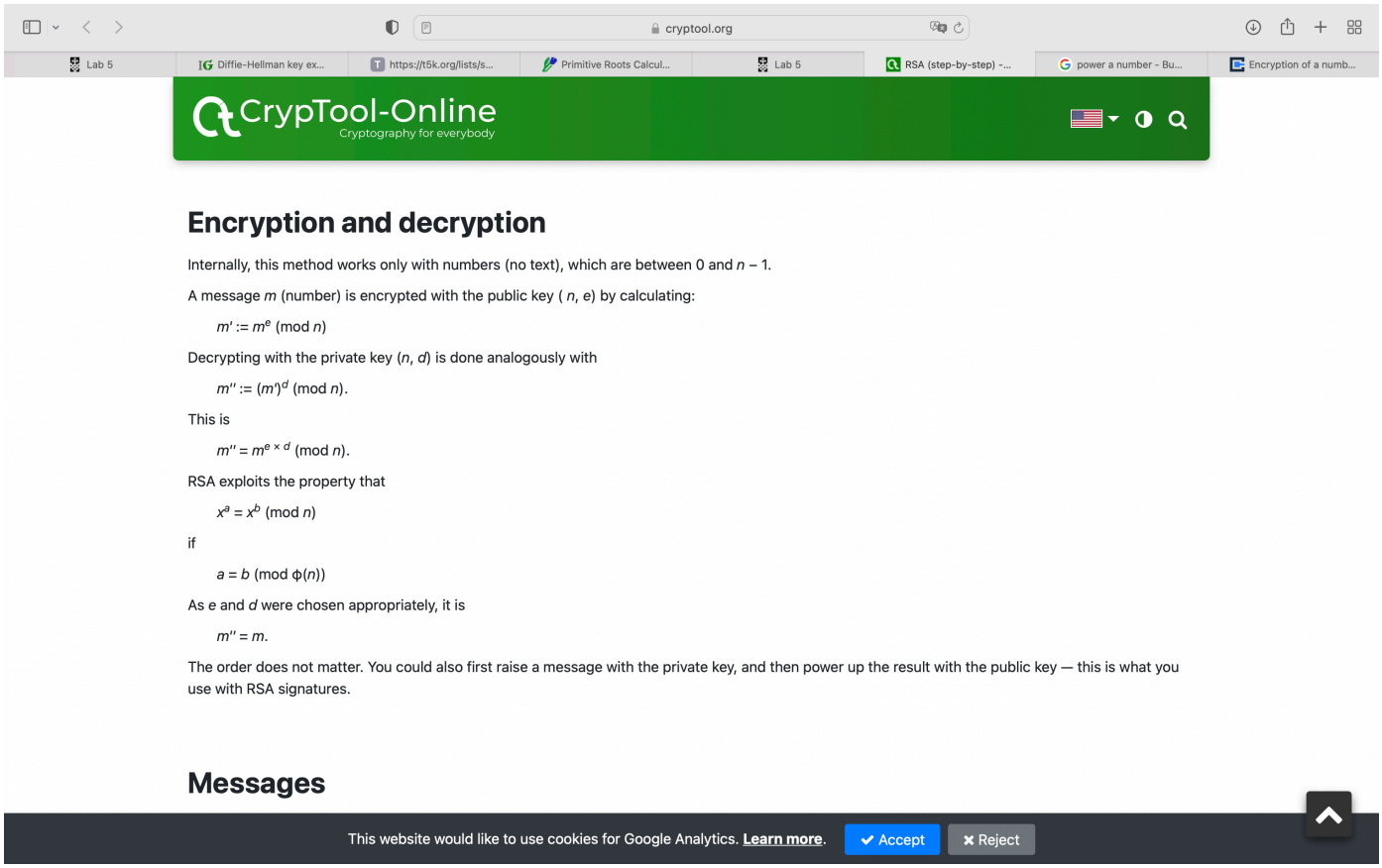
Then it is checked that gcd(e, phi(n)) = 1 = gcd(33,53200)

If phi(n) would have to be calculated without knowing p and q it would be significantly more complex and incredibly computationally heavy. This provides security to the algorithm.

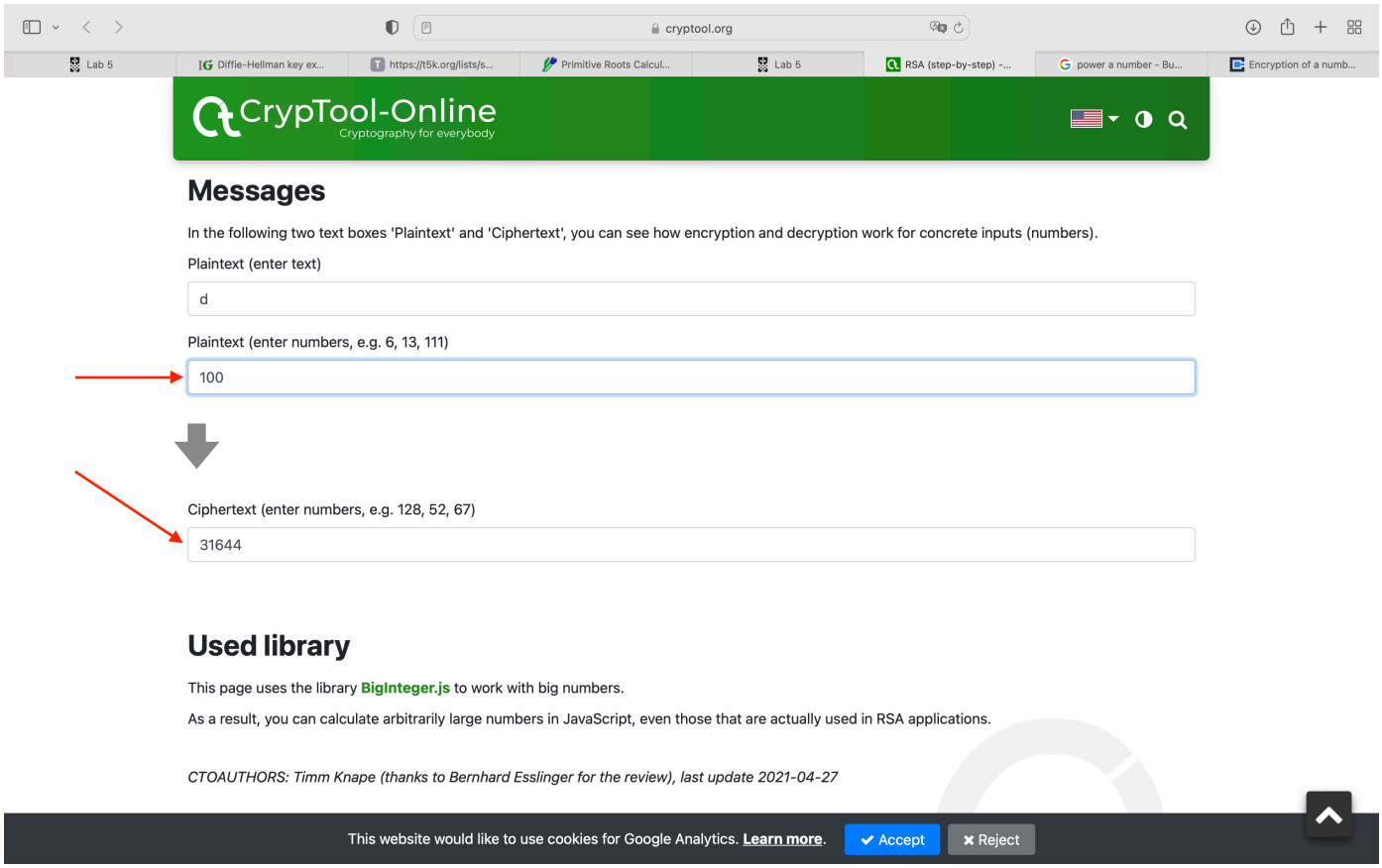
Then d is found such that is the multiplicative inverse of e modulo n:

e * d = 1 mod phi(n) or in our case 33 * 12897 = 1 mod 53200

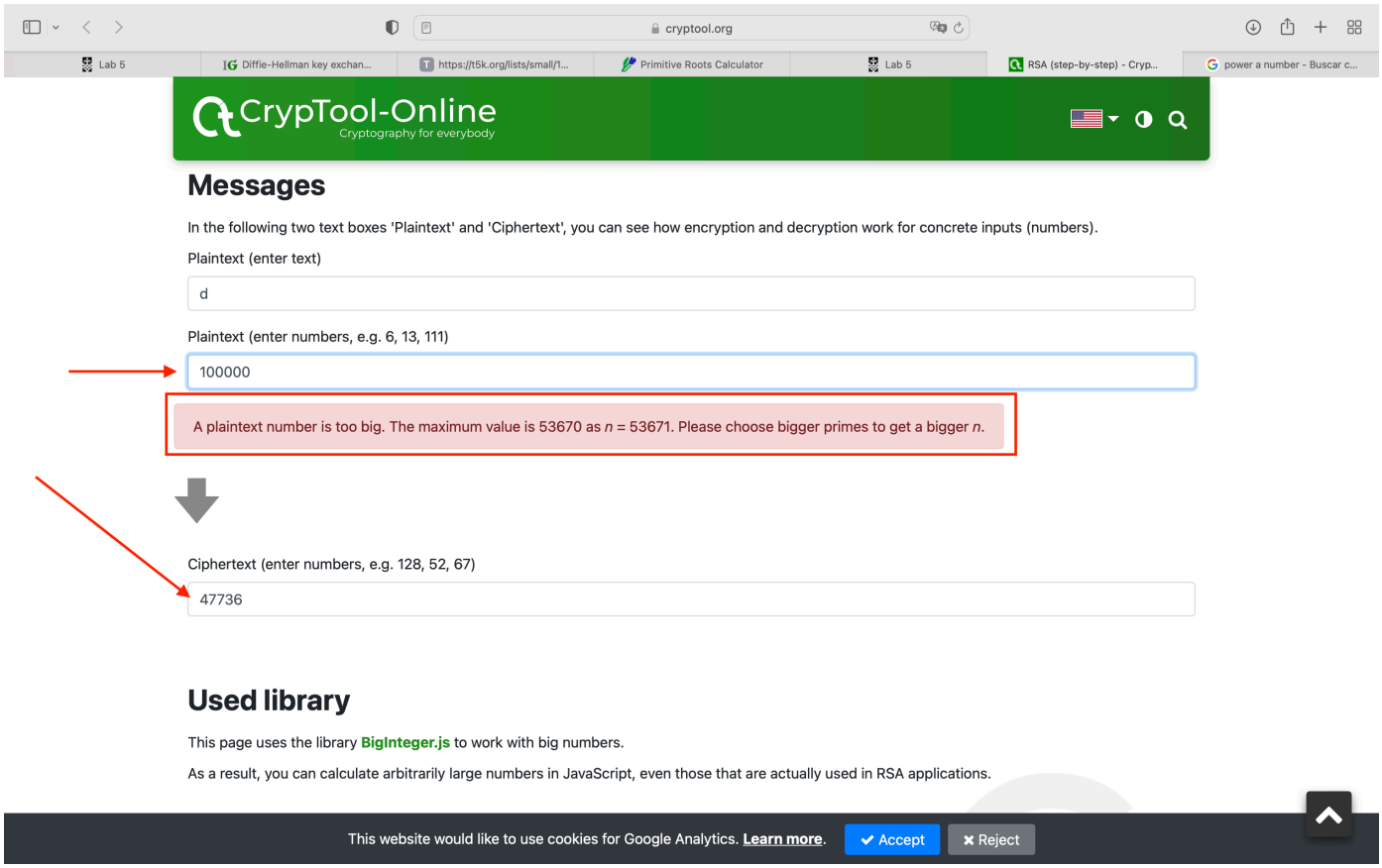
This d is calculated with the help of the extended Euclidean algorithm.



Now, the website explain how encryption and decryption occurs and we can proceed to encrypt messages:



When we try to encrypt a value greater than n, the program tells us that this cannot be done. Both python program and website allow us to do it but the results are not correct for the reasons given when answering the question:

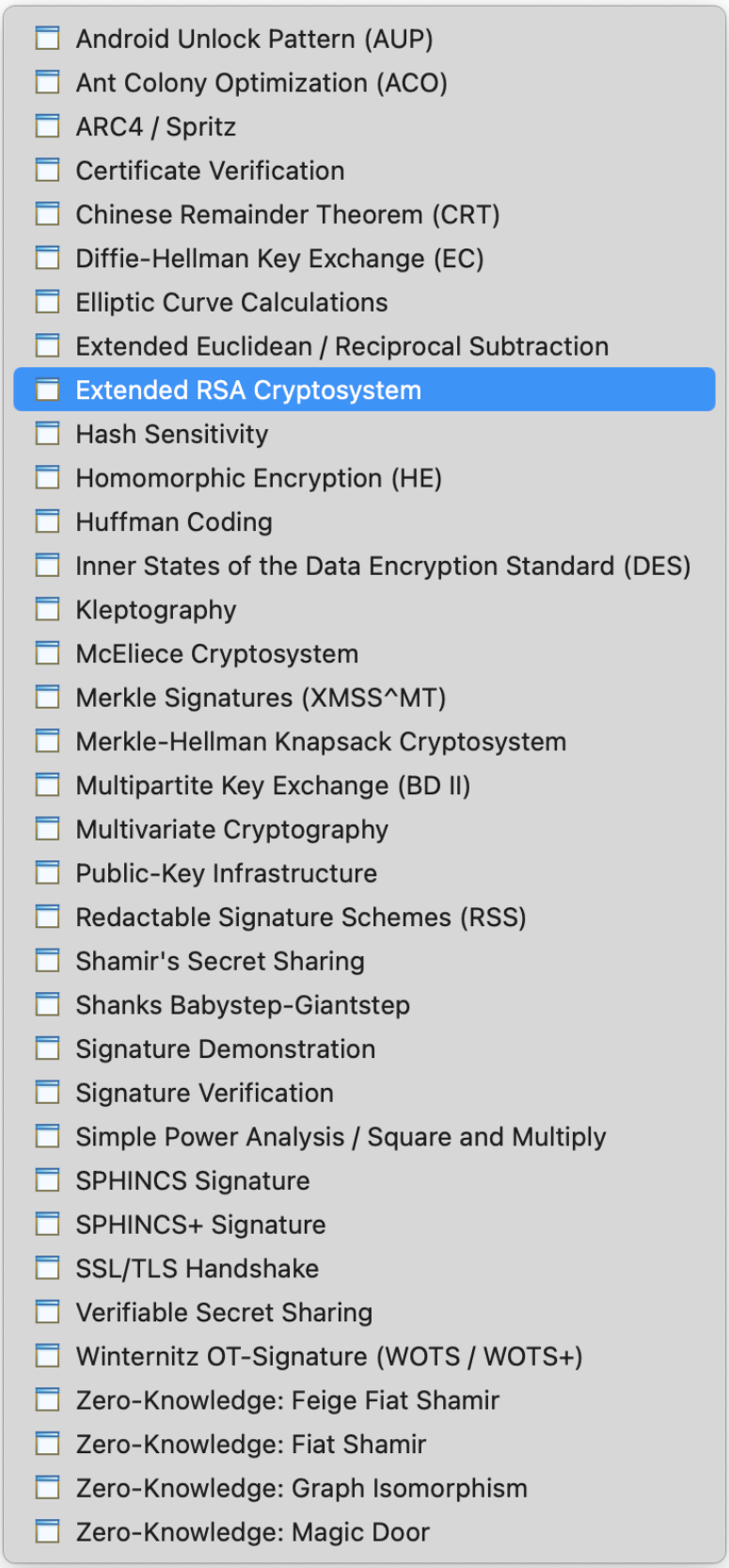


Part Five: Attack RSA Keys

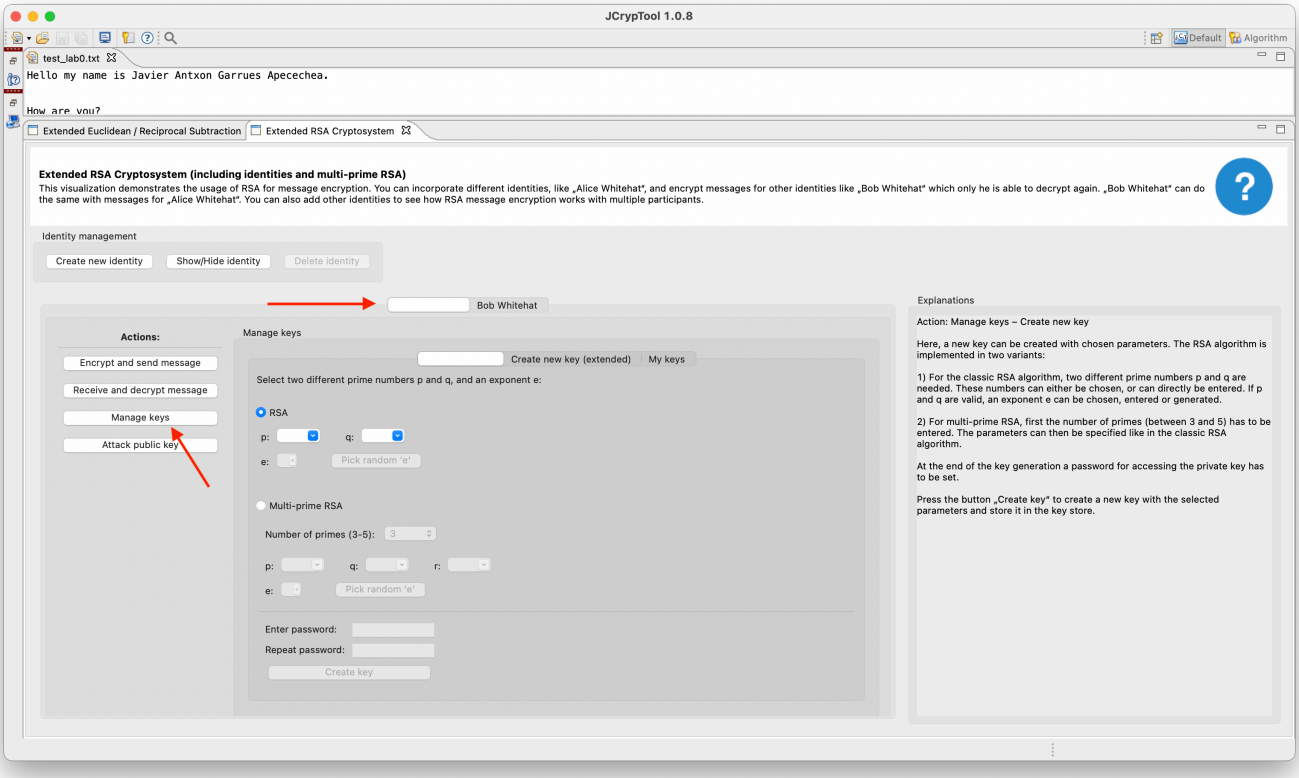
The strength of RSA is based on the hardness of the IFP (integrer factorization problem). An efficient non-quantum solution to this problem when the numbers are large haven't been found, but it has been impossible to prove that this algorithm does not exist.

Now, we will see that RSA can be cracked in an exhaustive-way search.

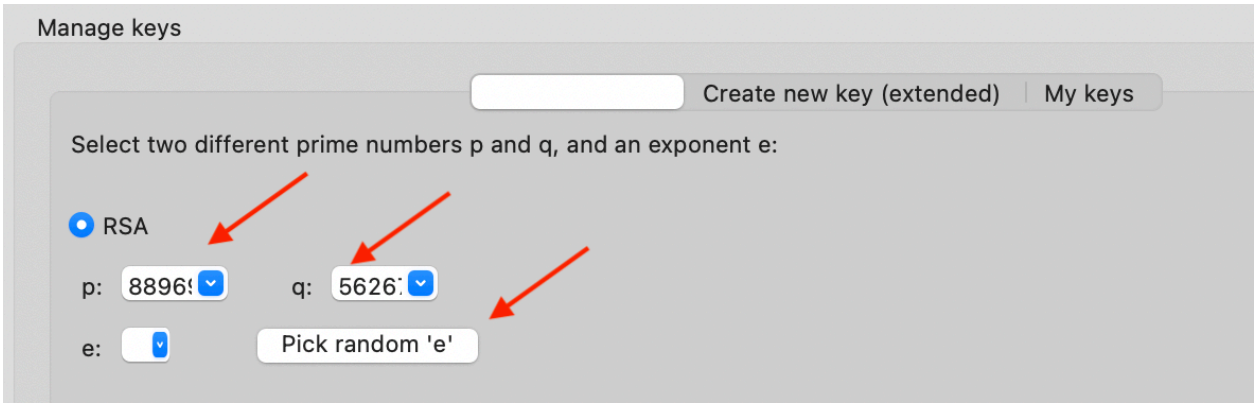
To do this task we will use JCrypTools. Once in there, we selexct the option `Visuals -> Extended RSA Cryptosystem`:



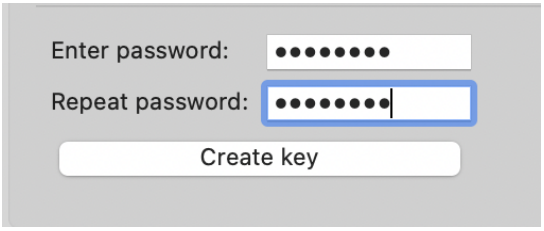
Then, we click on Alice whitehat and select the `Manage keys` option:



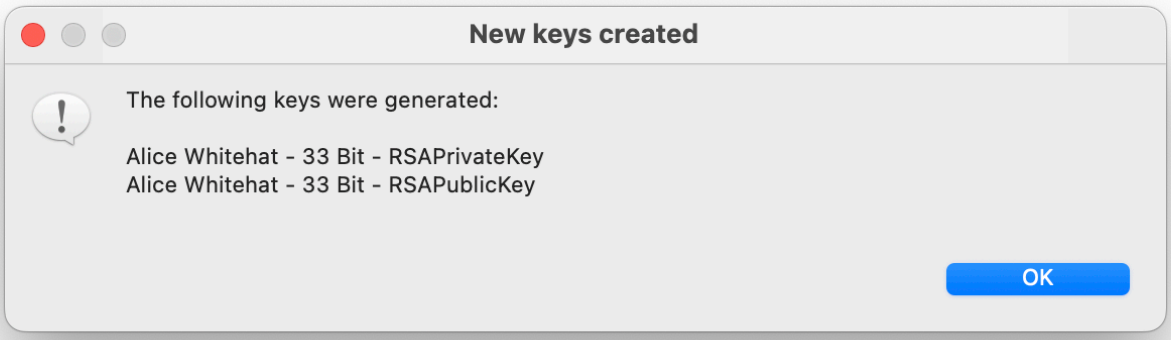
And proceed to enter the primes and select `Pick random 'e'`. The chosen primes are **88969** and **56267**:



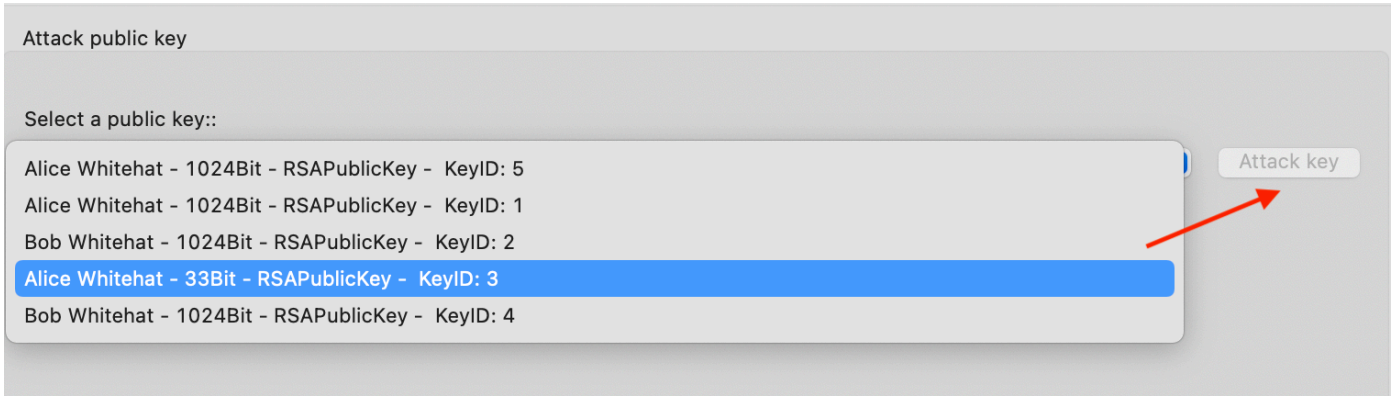
Then a password is selected and we click on `Create key`:



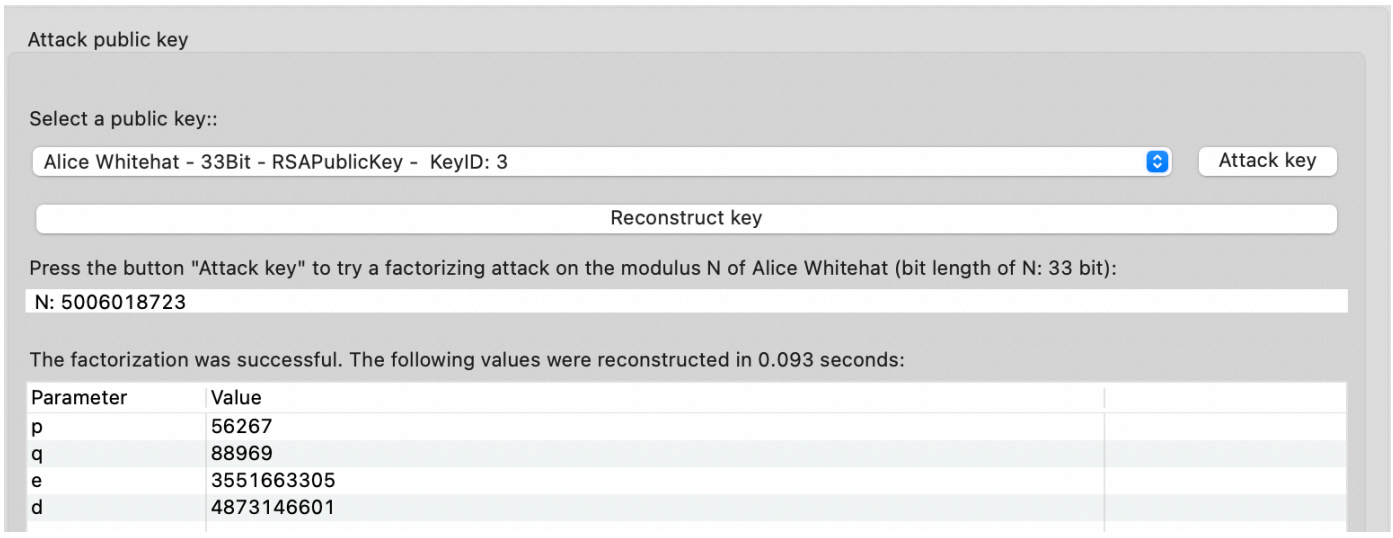
The lenght of the the key depends on the chosen prime numbers:



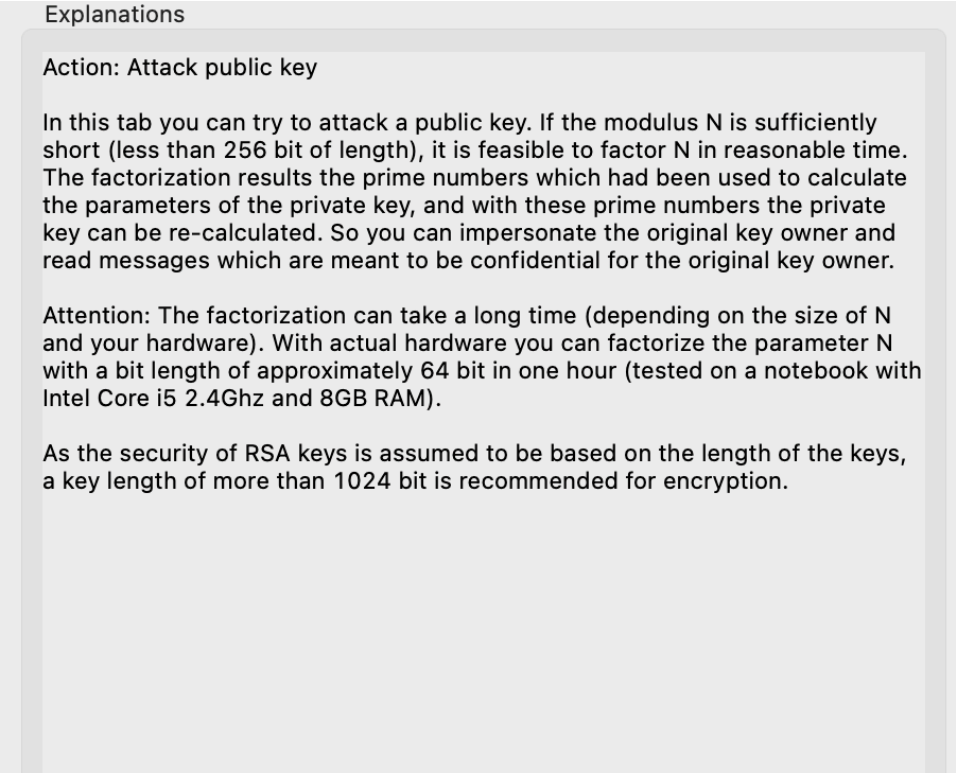
Once created, we can proceed to attack it:



The attack is successful and surprisingly fast due to the relatively *small* size of the chosen primes:

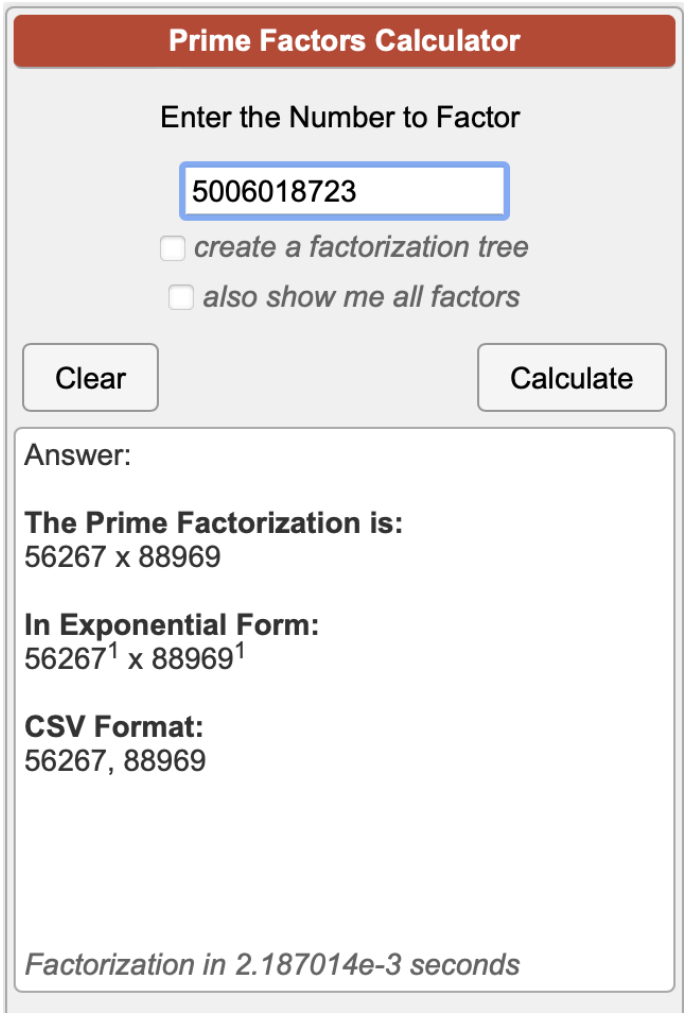


We can read the explanation provided by JCrypTools to understand what has happened:



It says a key **length** of more than **1024 bits is recommended**, something that our key clearly didn't satisfy.

There are also other online tools that can be used to factor numbers in primes such as `https://www.calculatorsoup.com/calculators/math/prime-factors.php`:



Question: to factor a number N using the exhaustive-search, i.e., try factors in 1,2,3...N how many numbers at most should be tried?

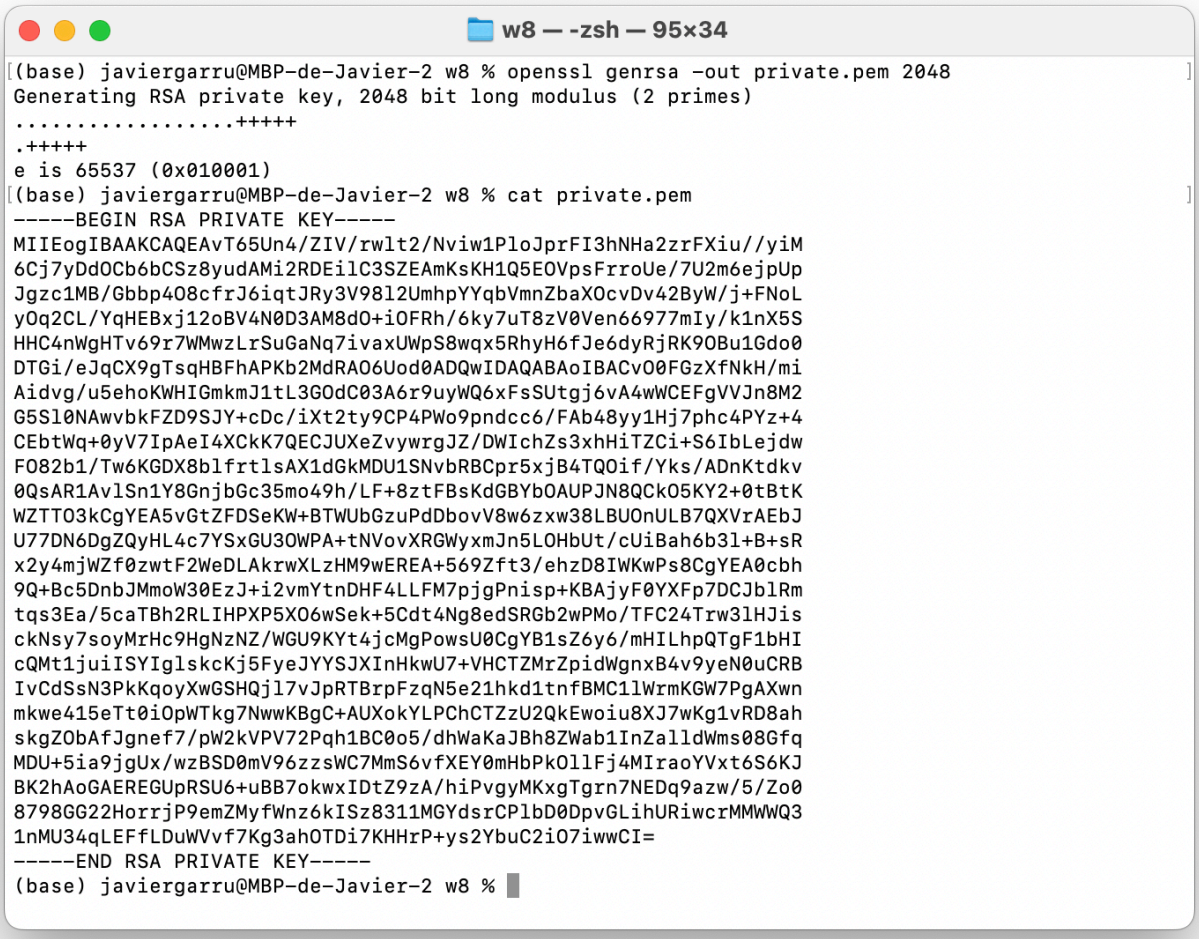
Supposing the question means to find p and q such that p*q=N when it says factor a number N, then the maximum number of primes that should be tried is sqrt(N)/2. The reason for this is that if N has a prime factor which is greater than its square root then it has prime factor that is smaller than its square root. This means that we would have found the smaller prime before reaching the bigger primer. The half comes due to the fact that all even numbers are not prime. If after trying sqrt(N)/2 numbers we haven't obtained a prime then N would be prime which cannot be as N = p*q and p and q cannot be 1 as they are prime. The actual number varies depending on the value of N as the distribution of prime number is not homogeneus on the real line. It could be also used that approximately the number of primes smaller than N is N/ln(N).

Part Six: RSA Encryption and Decryption

6.1.Generate RSA private and public key pair:

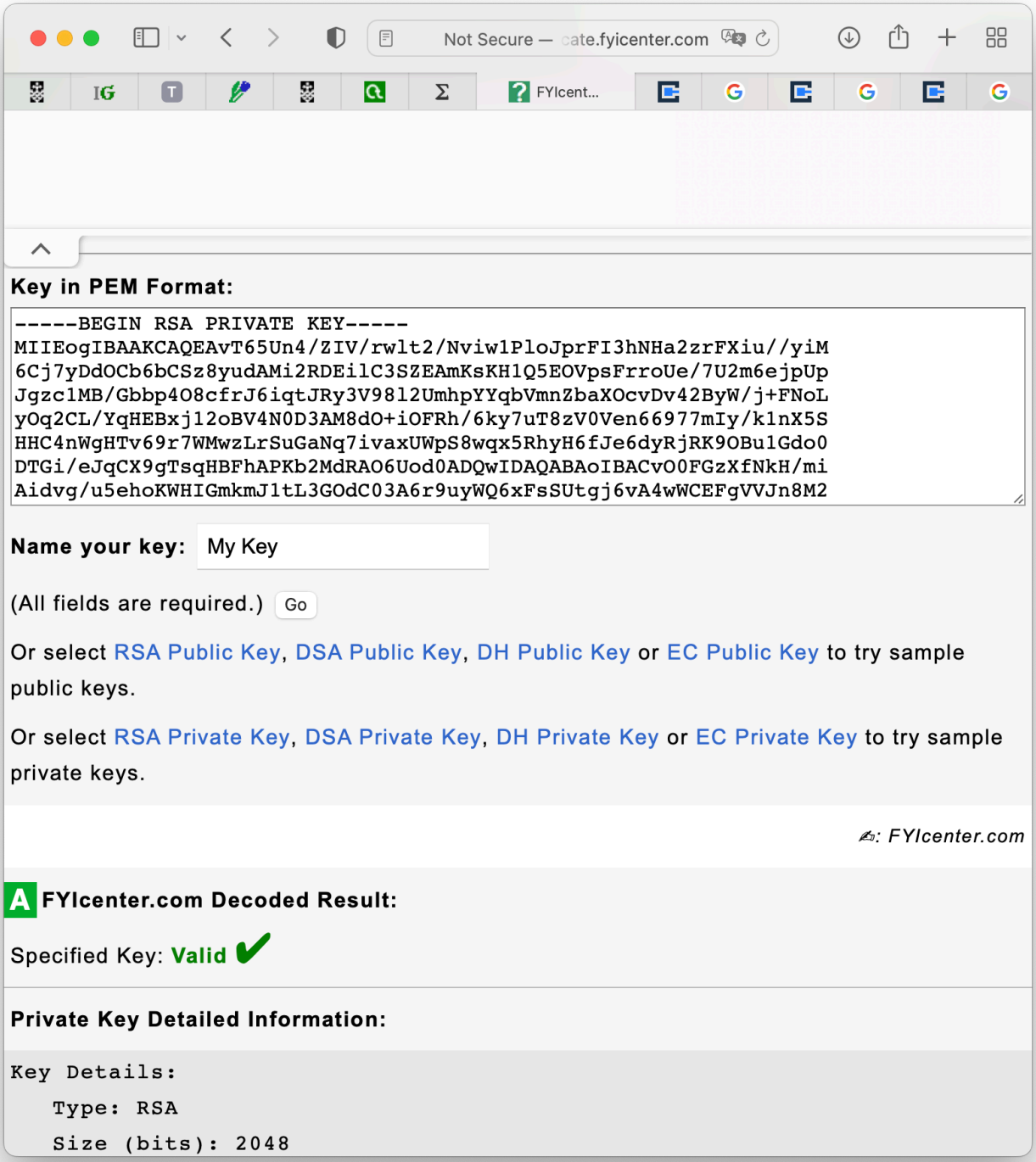
Let's now work with openssl. First we generate a private key:

```
openssl genrsa –out private.pem 2048
cat private.pem
```

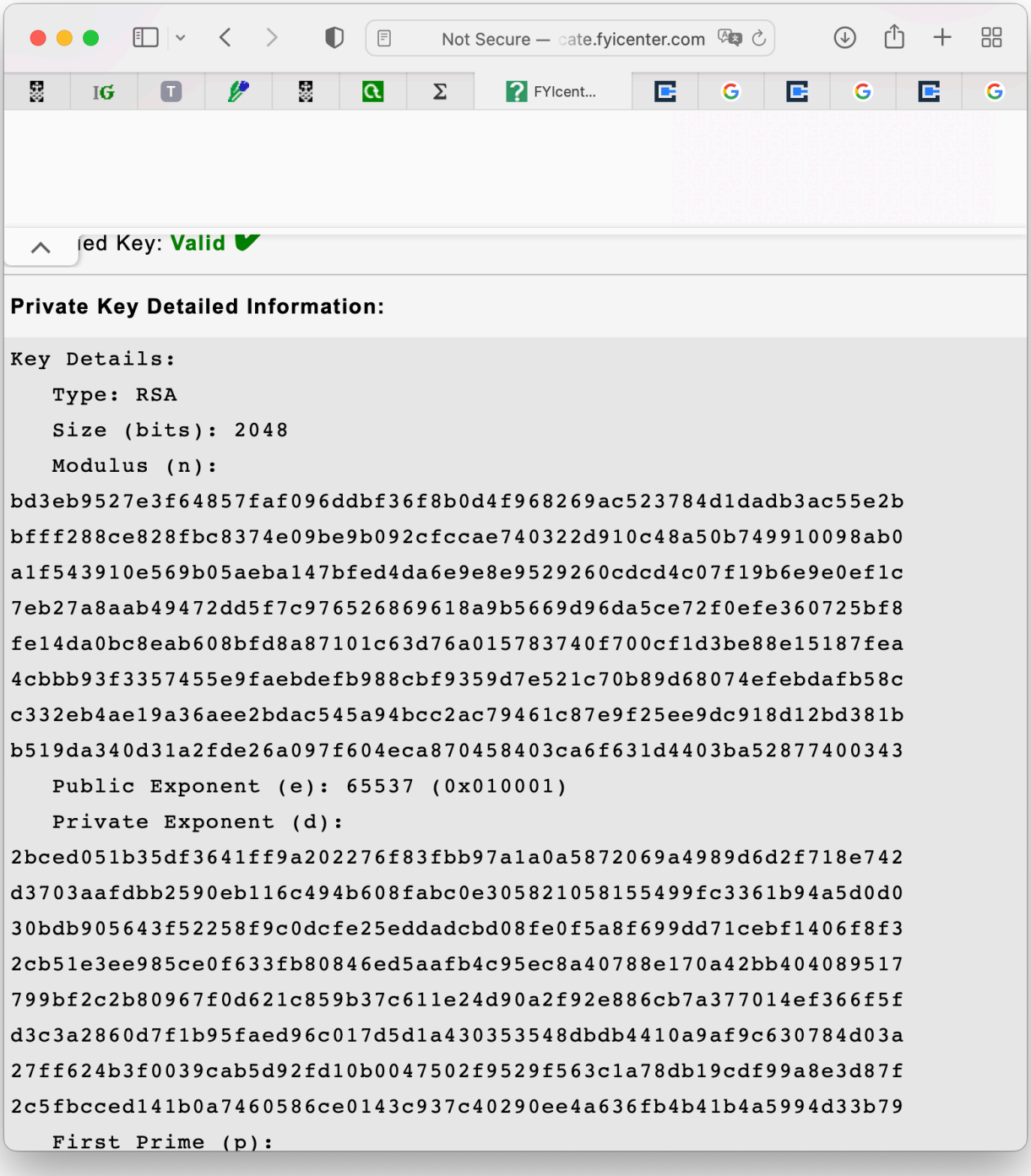


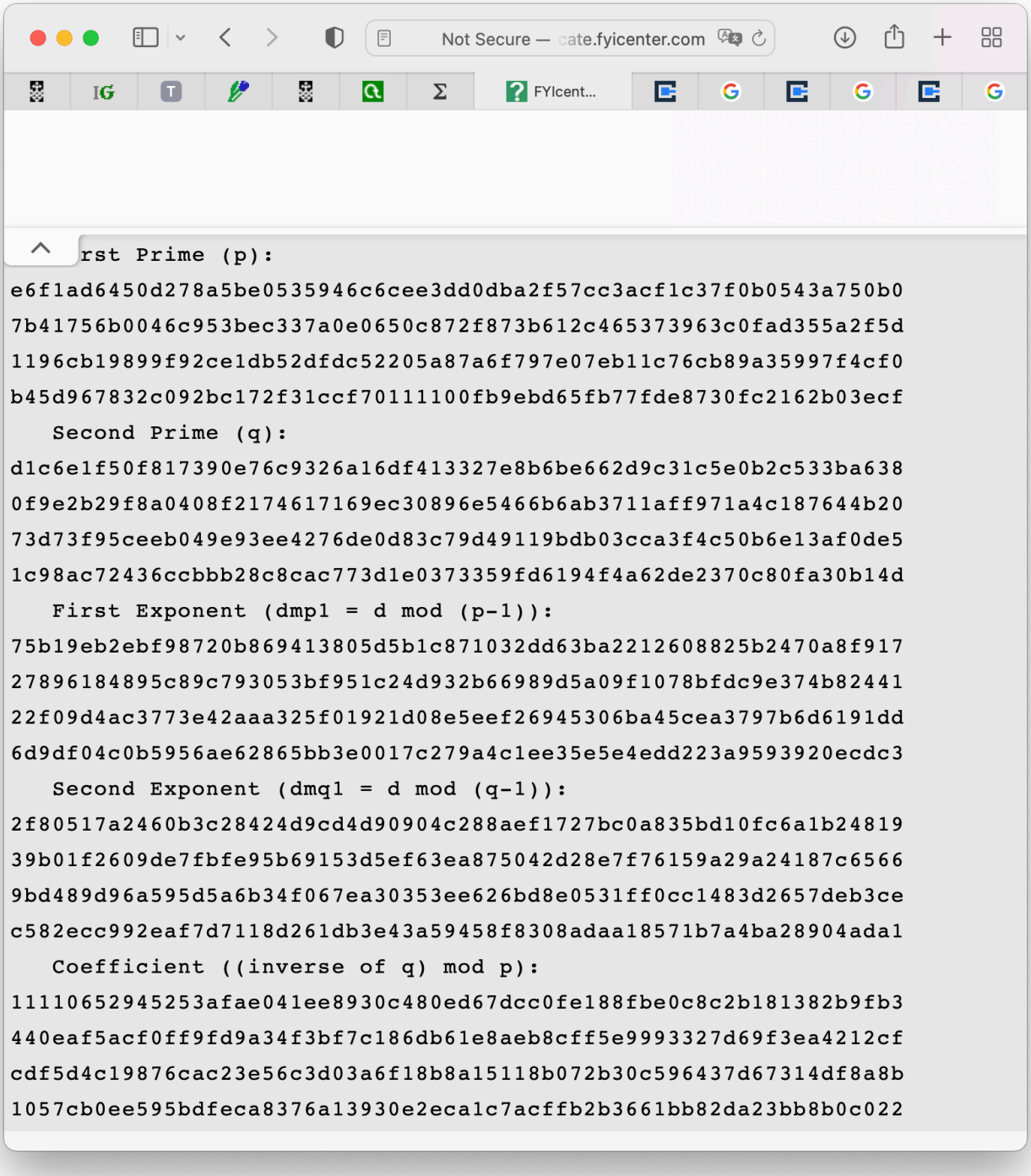
The format is not what we were expecting due to us saving the key in a .pem file which follows and specific standard that encodes the keys. We can recover the values in an online site `http://certificate.fyicenter.com/2145_FYIcenter_Public_Private_Key_Decoder_and_Viewer.html`.

We copy and paste the key:


Not Secure — cate.fyicenter.com
FYIcent...
Key in PEM Format:
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAvT65Un4/ZIV/rwlt2/Nviw1PloJprFI3hNHa2zrFXiu//yiM
6Cj7yDdOCb6bCSz8yudAMi2RDEilC3SZEAmKsKH1Q5EOVpsFrroUe/7U2m6ejpUp
Jgzc1MB/Gbbp4O8cfrJ6iqtJRy3V98l2UmhpYYqbVmnZbaXOcvDv42ByW/j+FNoL
yOq2CL/YqHEBxj12oBV4N0D3AM8dO+iOFRh/6ky7uT8zV0Ven66977mIy/k1nX5S
HHC4nWgHTv69r7WMwzLrSuGaNq7ivaxUWpS8wqx5RhyH6fJe6dyRjRK9OBu1Gdo0
DTGi/eJqCX9gTsqHBFhAPKb2MdRAO6Uod0ADQwIDAQABAoIBACvO0FGzXfNkH/mi
Aidvg/u5ehoKWHIGmkmJ1tL3GOdC03A6r9uyWQ6xFsSUtgj6vA4wWCEFgVVJn8M2
Name your key:
My Key
(All fields are required.)
Go
Or select RSA Public Key, DSA Public Key, DH Public Key or EC Public Key to try sample public keys.
Or select RSA Private Key, DSA Private Key, DH Private Key or EC Private Key to try sample private keys.
✍: FYIcenter.com
A FYIcenter.com Decoded Result:
Specified Key: Valid ✔
Private Key Detailed Information:
Key Details:
Type: RSA
Size (bits): 2048

And the read the generated values:



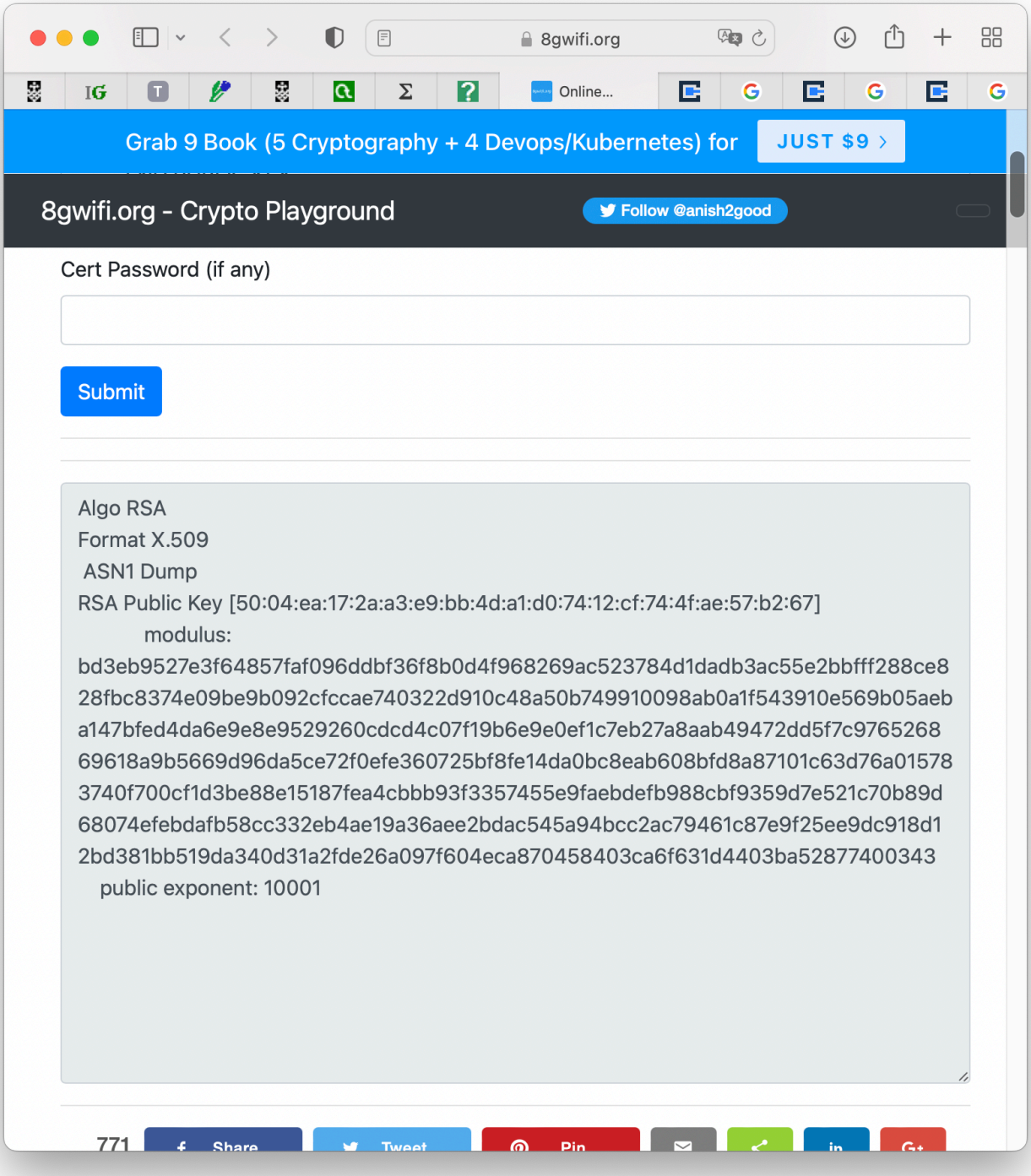


We can also use the website `https://8gwifi.org/PemParserFunctions.jsp` . We type in the following key:

Public Key in PEM Format:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvT65Un4/ZIV/rwlt2/Nv
iw1PloJprFI3hNHa2zrFXiu//yiM6Cj7yDdOCb6bCSz8yudAMi2RDEilC3SZEAmK
sKH1Q5EOVpsFrroUe/7U2m6ejpUpJgzc1MB/Gbbp4O8cfrJ6iqtJRy3V98l2Umhp
YYqbVmnZbaXOcvDv42ByW/j+FNoLyOq2CL/YqHEBxj12oBV4N0D3AM8dO+iOFRh/
6ky7uT8zV0Ven66977mIy/k1nX5SHHC4nWgHTv69r7WMwzLrSuGaNq7ivaxUWpS8
wqx5RhyH6fJe6dyRjRK9OBu1Gdo0DTGi/eJqCX9gTsqHBFhAPKb2MdRAO6Uod0AD
QwIDAQAB
-----END PUBLIC KEY-----
```

And click submit:



Question: Try the following tools and confirm you can recover RSA components

From this websites we can extract the RSA components. The values of n,d,p and q are inmense and we display them here in their hexadecimal and decimal format:

e:

```
# IN DECIMAL
65537
```

d:

```
# IN DECIMAL
5530230772998165819290828129059033908687049149197062569206423865599499316060113910545532082867780861289386209130198495726371165430862166604235406437132190330183779173896301210047885474126599966648677996246619164897925652253174232757119728919901864894203538825798574898770095701360024530601409127872078175810075579853001275069027039867199136046178287532487660800489564346890073950140228636218073214039481990935983345080666968545375208811152041980136776122108258372231453147289579590270371370394485819360003408633968818563561110059525628110834621370077442516724927489477022924924368871188409427977062525835909283396473
```

```
# IN HEX
2bced051b35df3641ff9a202276f83fbb97a1a0a5872069a4989d6d2f718e742
d3703aafdbb2590eb116c494b608fabc0e305821058155499fc3361b94a5d0d0
30bdb905643f52258f9c0dcfe25eddadcbd08fe0f5a8f699dd71cebf1406f8f3
2cb51e3ee985ce0f633fb80846ed5aafb4c95ec8a40788e170a42bb404089517
799bf2c2b80967f0d621c859b37c611e24d90a2f92e886cb7a377014ef366f5f
d3c3a2860d7f1b95faed96c017d5d1a430353548dbdb4410a9af9c630784d03a
27ff624b3f0039cab5d92fd10b0047502f9529f563c1a78db19cdf99a8e3d87f
2c5fbcced141b0a7460586ce0143c937c40290ee4a636fb4b41b4a5994d33b79
```

n:

```
# IN DECIMAL
2388996995385807087857511061196637698725351922028395554664039291278059367718882640270401002668945714233224599484317571777860306300457542764101086491795744233532755492186684413703172304514105741310753475973994332673623053666312568006086320441787677276194168637705894160837754676554162070147152791598130560991823996244320091844623798659059032379365190538571822093567910868029950617957962560801521856464615284955131121021229852843986555296997418465879420462350800009145266614240506774538503812152738960471732217090551624981687283976999418406599997155570049394080496525499408832853034526316033103316676007877404234621125443
```

```
# IN HEX
bd3eb9527e3f64857faf096ddbf36f8b0d4f968269ac523784d1dadb3ac55e2b
bfff288ce828fbc8374e09be9b092cfccae740322d910c48a50b749910098ab0
a1f543910e569b05aeba147bfed4da6e9e8e9529260cdcd4c07f19b6e9e0ef1c
7eb27a8aab49472dd5f7c976526869618a9b5669d96da5ce72f0efe360725bf8
fe14da0bc8eab608bfd8a87101c63d76a015783740f700cf1d3be88e15187fea
4cbbb93f3357455e9faebdefb988cbf9359d7e521c70b89d68074efebdafb58c
c332eb4ae19a36aee2bdac545a94bcc2ac79461c87e9f25ee9dc918d12bd381b
b519da340d31a2fde26a097f604eca870458403ca6f631d4403ba52877400343
```

p:

```
# IN DECIMAL
162174428444261123897833406675287093215145740839845914043759244991833062109813200715192
475680887384379020555080038015469458026049937325153929567808717842030507786289200269381
992955534694844897372446491764140886841665167011134562043574487815839313669940206236376
981687593830007920038694029093897844403886636751
```

```
# IN HEX
e6f1ad6450d278a5be0535946c6cee3dd0dba2f57cc3acf1c37f0b0543a750b0
7b41756b0046c953bec337a0e0650c872f873b612c465373963c0fad355a2f5d
1196cb19899f92ce1db52dfdc52205a87a6f797e07eb11c76cb89a35997f4cf0
b45d967832c092bc172f31ccf70111100fb9ebd65fb77fde8730fc2162b03ecf
```

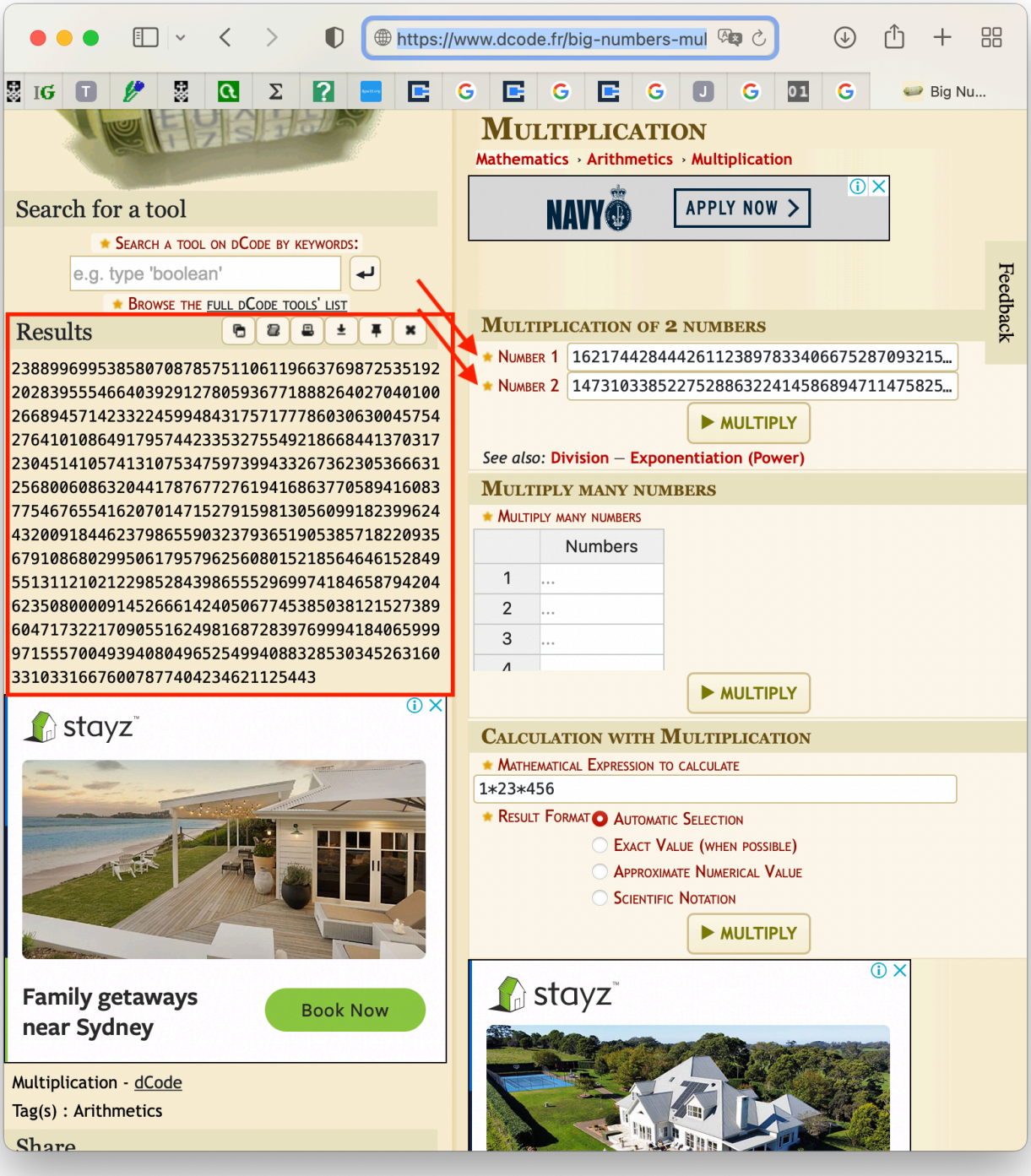
q:

```
# IN DECIMAL
147310338522752886322414586894711475825736907401480012862258834254495365146784636478415
077952555158256448202848010654925491242349730736978063309233624654166308973300069462271
562644337934471045759378838742370756094046437041798372965903349269550839080479094763799
391114635320083007614178949547506966858174607693
```

```
# IN HEX
d1c6e1f50f817390e76c9326a16df413327e8b6be662d9c31c5e0b2c533ba638
0f9e2b29f8a0408f2174617169ec30896e5466b6ab3711aff971a4c187644b20
73d73f95ceeb049e93ee4276de0d83c79d49119bdb03cca3f4c50b6e13af0de5
1c98ac72436ccbbb28c8cac773d1e0373359fd6194f4a62de2370c80fa30b14d
```

The size of the numbers make impossible visually checking if n is indeed the product of p and q. We can use the following website to check it `https://www.dcode.fr/big-numbers-multiplication`:

As we can see the obtained value for n is the expected value.



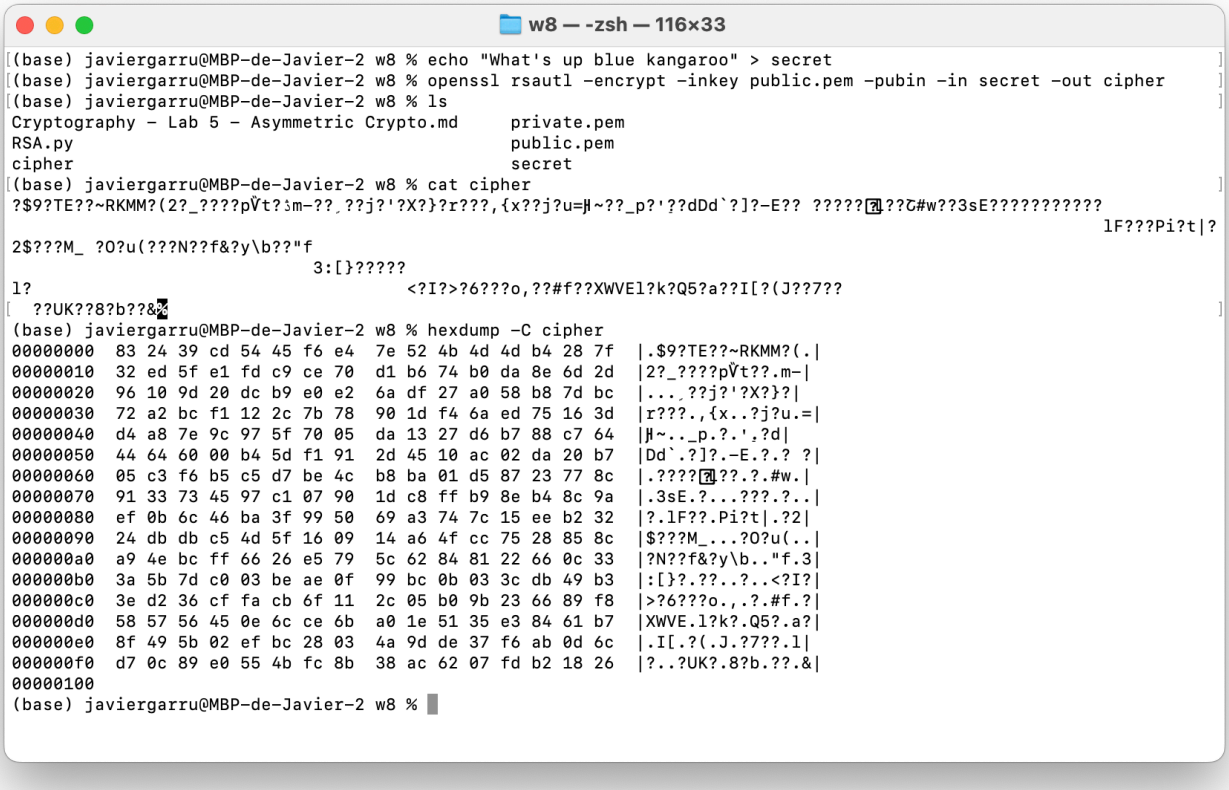
To conclude we can derive the public key from the private key:

```
openssl rsa -in private.pem -outform PEM -pobout -out public.pem
ls
cat public.pem
```

6.2.Encrypt a small file with the public key and decrypt with the private key:

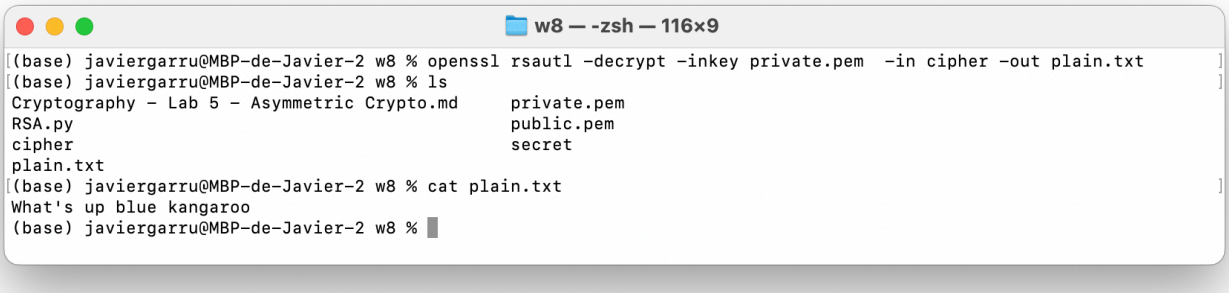
We encrypt using the public key a file called secret and save it in another one called cipher:

```
echo "What's up blue kagaroo" > secret
openssl rsautl -encrypt -inkey public.pem -pubin -in secret -out cipher
ls
cat cipher
```



Now, we can proceed to decrypt it using the private key:

```
openssl rsautl -decrypt -inkey private.pem  -in cipher -out plain.txt
ls
cat plain.txt
```

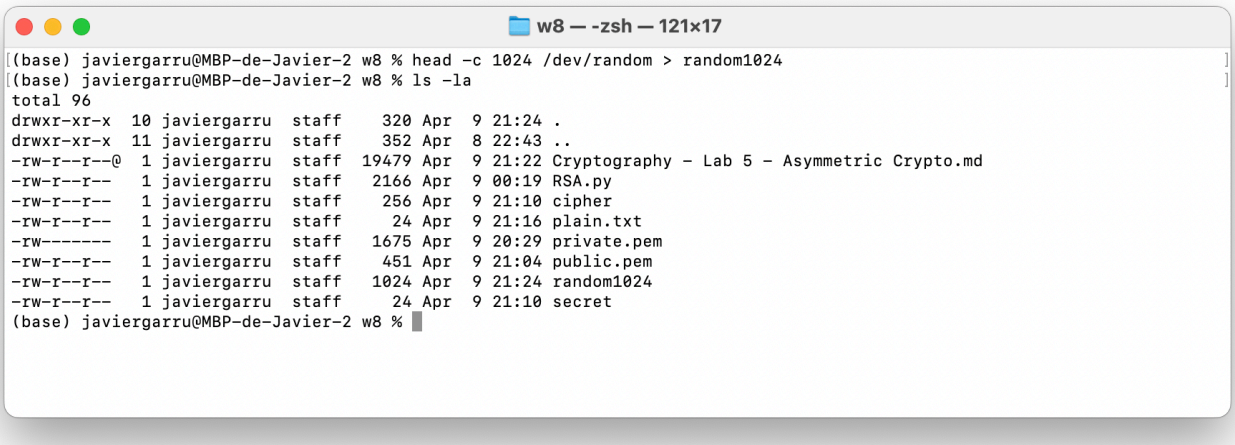


As we can see the plaintext that was contained in the file has been correctly deciphered.

6.3.Size limitations of RSA encryption:

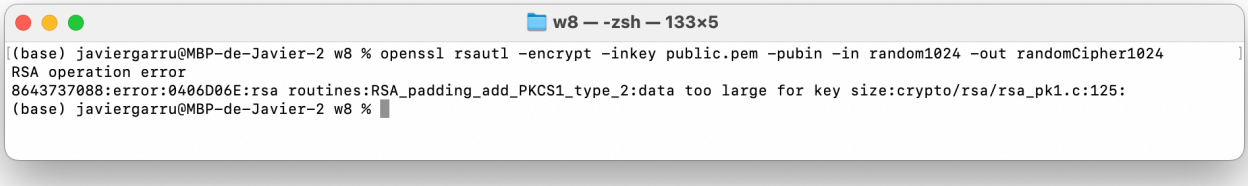
RSA is not usually used for encrypting large files. Encrypting a 1024 bytes file with a 2048-bit key will produce an error. We first create a file full of 1024 random bytes and call it random1024:

```
head -c 1024 /dev/random > random1024
```

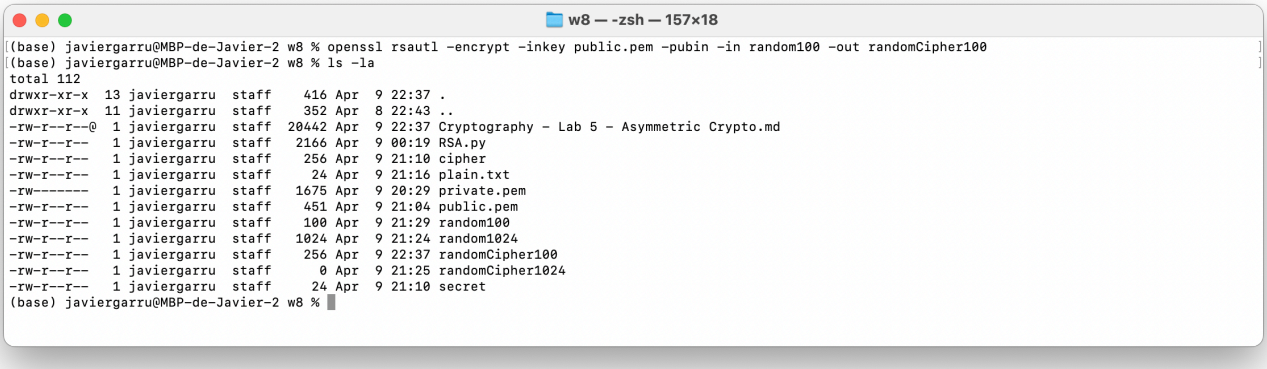


Then we try to encrypt it using our 2048-bit key:

```
openssl rsautl -encrypt -inkey public.pem -pubin -in random1024 -out randomCipher1024
```

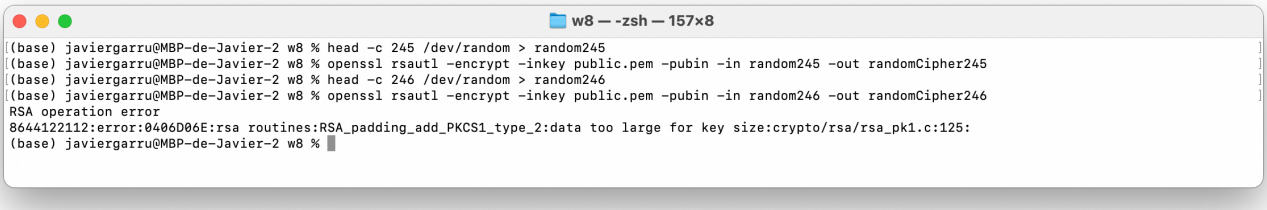


And an error shows indicating that the dat is too large for our key. With smaller values like 100 this won't happen:



Question: How many bytes at maximum can be encrypted with 2048-bit key RSA? Why?

The maximum encryptable size is the same length of the key, that is 256-11= 245 bytes (this value changes depending on the padding algorithm being used). This is because before encrypting, the message must be converted into a number including 11 bytes of padding for security reasons. If the message is larger it cannot be converted to a number. As the documentation of openssl rsautl indicates, the RSA algorithm is directly used on the message (no blocking mode is used). If the message cannot be converted into a 245 byte-number, then when applyting the module to the power of that number the encrypted value will match the encrypted value of another number too (this was seen when woeking with the script before). We can test that 11bytes is the exact padding value easily:



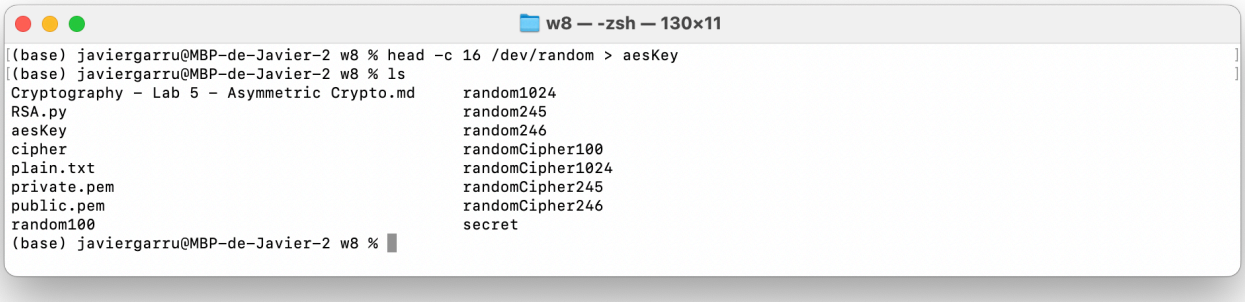
If we pay more attention to the specific error, we can see that is says it failed at *RSA_padding_add_PKCS1_type_2*, confirming that it fails when it isn't capable of adding the minimum padding necessary for security reasons.

6.3.Protect big files:

In order to avoid the above shown problem, a solution is proposed. This solution consists on generating a symmetric key, encrypting the file with a symmetric cipher mode, encrypying the key with RSA and sending the encrypted message and key to the receiver.

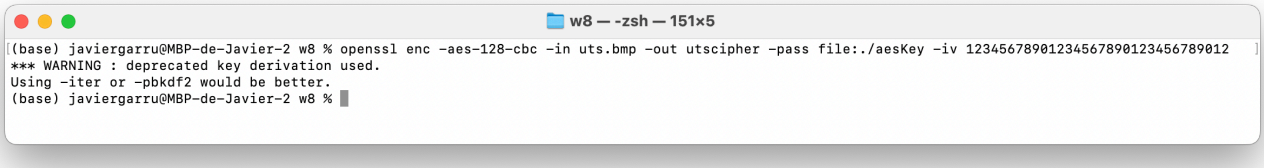
First, we generate a random encryption key:

```
head -c 16 /dev/random > aesKey
```



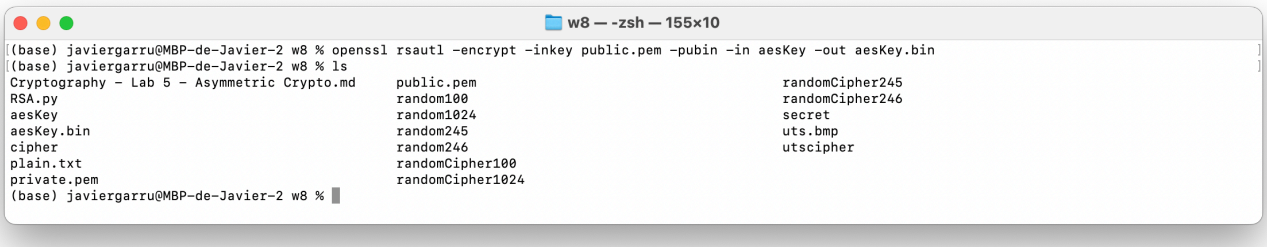
Then, we encrypt the desired file with the generated key and the algorithm selected (aes-128-cbc):

```
openssl enc -aes-128-cbc -in uts.bmp -out utscipher -pass file:./aesKey -iv
12345678901234567890123456789012
```



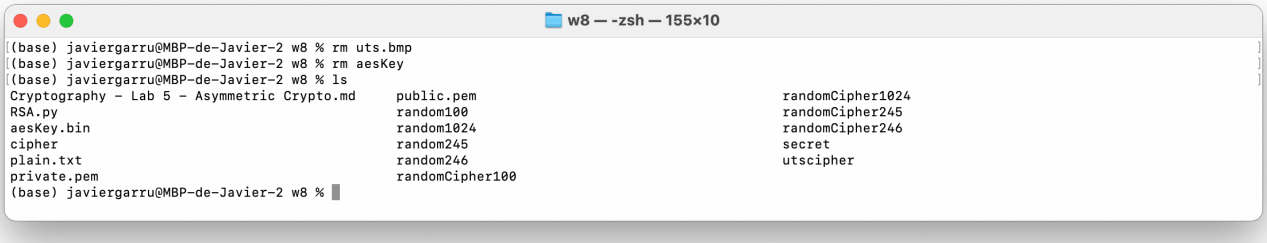
Now, we can ecrypt the key using RSA:

```
openssl rsautl -encrypt -inkey public.pem -pubin -in aesKey -out aesKey.bin
```



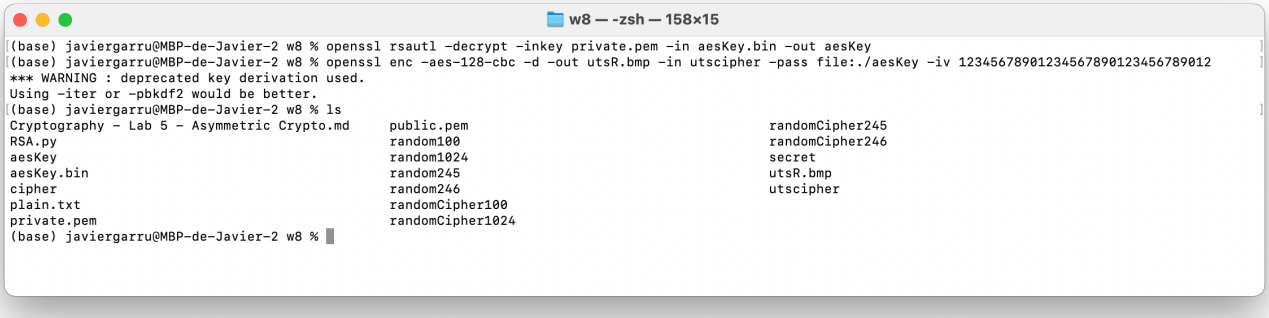
And end up deleting the key and the original file:

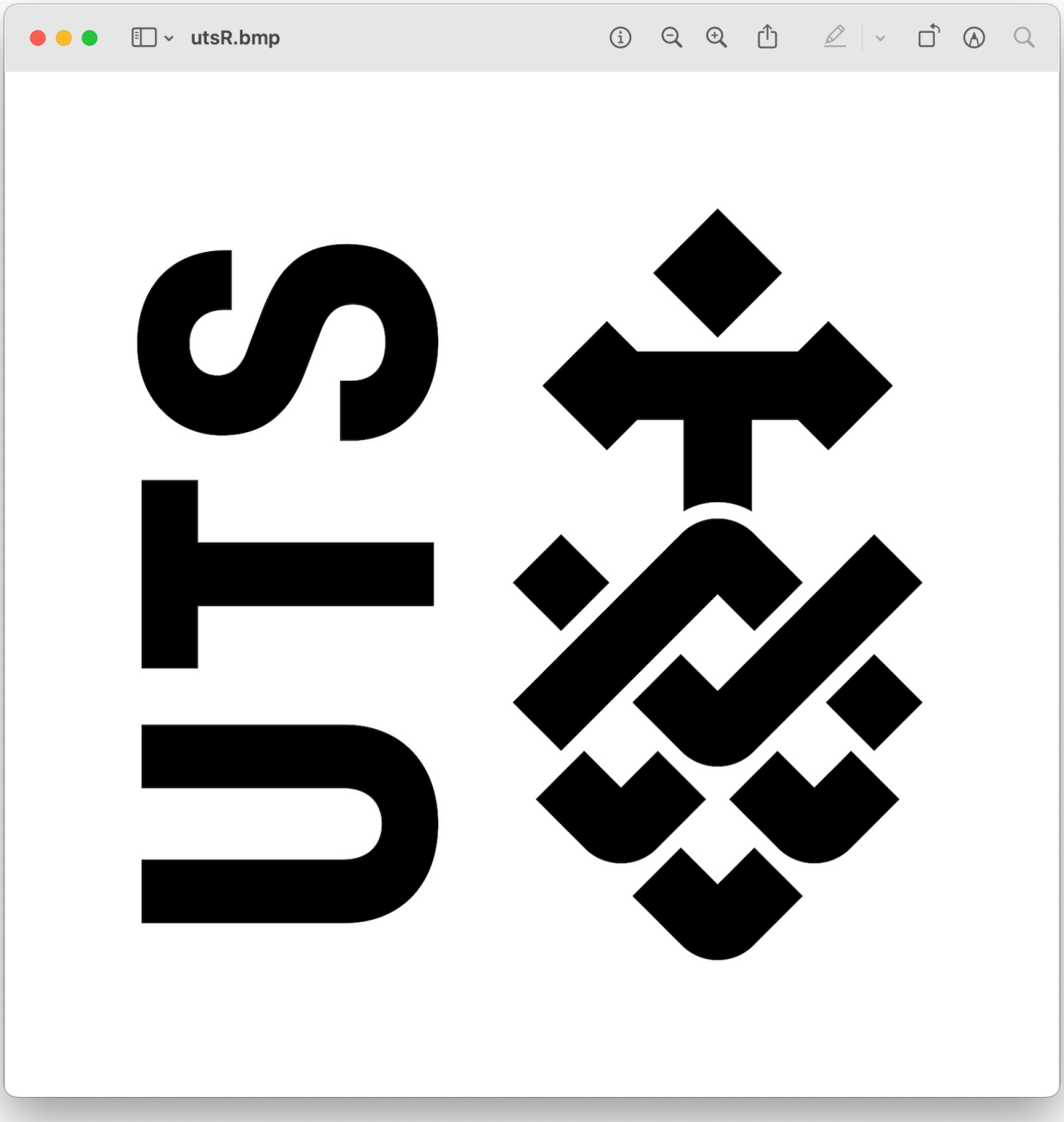
```
rm aesKey
rm uts.bmp
```



Now, supposed we are at the receiver side. We just need to use the RSA algorithm to decrypt the key (using the private key) and then the aes algorithm with the key to decrypt the file:

```
openssl rsautl -decrypt -inkey private.pem -in aesKey.bin -out aesKey
openssl enc -aes-128-cbc -d -out utsR.bmp -i utscipher -pass file:./aesKey -iv
12345678901234567890123456789012
ls
```





Question: Why the private keys in asymmetric ciphers are ofthen generated rathern than specified?

To make the whole process more secure. If the keys were specified by the user, the chosen numbers being used wouln't be (most of the times) as random as the ones a pseudorandom number generator based on a good source of entropy can be. For instance, users might choose primes that are small like 7, while the generator would usually generate big and random primes. The harder is to guess the value of the used primes, the strongest is the encryption algorithm used to encrypt the symmetric key using RSA. If this key was discovered, then the message could be also discovered.

Part Seven: Lab Summary

In this lab we have learnt how the Diffie Hellman Key Exchange algorithm works. We have also in depth study the RSA algorithm and what is the maximum length that can be encrypted depending on the key-size. We have seen how the keys are generated and the strength of using large primes due to the difficulty of factoring numbers using the exhaustive search. At last, we have learnt about the usual procedure that combines asymmetric and symmetric encryption techniques to *securely* transmit messages.