Homework #4

Due date: 26 November 2017

1. (**20 pts**) The composite number n is the product of two large primes, namely p and q. Assume that nobody knows the factorization of n. Consider the following function

$$H(x) = x^2 \pmod n.$$

H(x) is one-way function since it is difficult to compute the square root of x modulo a composite number when the factorization is not known.

a. Explain why H(x) is <u>not</u> a good cryptographic hash function by discussing the properties of cryptographic hash functions. (**Hint:** Discuss non-invertibility, weak and strong collision resistance.) (**10 pts**)

If we just look into the non-invertibility property, H(x) is not a bad cryptographic function since it is not easy to compute the square roots of x modulus n without knowing the factorization of n is not known and it wouldn't be easy to invert the function without the factorization. But $H(x) = x^2 \pmod n$ doesn't have weak and strong collision resistance since for any given message x, it is computationally feasible to find $y \neq x$ such that $H(x) = H(y)$; if we select $y = -x$ then, $-x \neq x$ and $H(x) = x^2 = (-x)^2 = H(-x)$.
For strong collision resistance it should be computationally infeasible to find any pair (x,y) with $x \neq y$ such that $H(x) = H(y)$ but it is computationally feasible to find any pair (x,-x) such that $H(x) = H(-x)$. Without having weak and strong collision resistance we cannot specify $H(x) = x^2 \pmod n$ as a good cryptographic function.

b. Explain why H(x) is <u>not</u> even a good hash function by demonstrating its output is biased. Explain your answer using on the example, where n = 15, i.e. p = 3 and q = 5. (**10 pts**)

```
c:\program files\python\2.7.13\python.exe
x: 1    H(x) =  1
x: 2    H(x) =  4
x: 3    H(x) =  9
x: 4    H(x) =  1
x: 5    H(x) =  10
x: 6    H(x) =  6
x: 7    H(x) =  4
x: 8    H(x) =  4
x: 9    H(x) =  6
x: 10   H(x) =  10
x: 11   H(x) =  1
x: 12   H(x) =  9
x: 13   H(x) =  4
x: 14   H(x) =  1

Counter({1: 4, 4: 4, 10: 2, 6: 2, 9: 2})

Process returned 0 (0x0)        execution time : 0.106 s
Press any key to continue . . .
```

When we compute H(x) for all elements in Z_{15}^*, we get the result as above. 1 and 4 have occurred 4 times while 6, 9 and 10 occurred 2 times and other elements in Z_{15}^* such as

2, 3, 5, 7, 8, 11, 12, 13 and 14 haven't appeared. This implies the output is biased towards 1 and 4.

Q1.py

2. **(30 pts)** In order to increase security, Bob chooses the modulus n and two encryption exponents, e_1 and e_2. He asks Alice to perform double encryption, i.e. to encrypt her message m to him by first computing $c_1 \equiv m^{e_1} \pmod N$, then encrypting c_1 to get $c_2 \equiv c_1^{e_2} \pmod N$. Alice then sends c_2 to Bob.

a. The double encryption does not increase security over single encryption and in fact is equivalent to single encryption with a private key e_3. Explain why? (**10 pts**)

$c_1 \equiv m^{e_1} \pmod N$, $\quad c_2 \equiv c_1^{e_2} \pmod N$

$c_2 \equiv (m^{e_1} \pmod N)^{e_2} \pmod N \equiv m^{e_1 e_2} \pmod N$, where $e_3 \equiv e_1 * e_2 \pmod{\phi(N)}$

$c_2 \equiv m^{e_3} \pmod N$ # Double Encryption with $e_3 \equiv e_1 * e_2 \pmod{\phi(N)}$

$c_1 \equiv c_2^{d_2} \pmod N$, $\quad m \equiv c_1^{d_1} \pmod N$

$m \equiv (c_2^{d_2} \pmod N)^{d_1} \pmod N \equiv c_2^{d_1 d_2} \pmod N$, where $d_3 \equiv d_1 * d_2 \pmod{\phi(N)}$

$m \equiv c_2^{d_3} \pmod N$ # Double Decryption with $d_3 \equiv d_1 * d_2 \pmod{\phi(N)}$

Alice encrypts her message: $c_2 \equiv c_1^{e_2} \equiv m^{e_1 e_2} \pmod N$

Bob calculates $(c_2^{d_2})^{d_1} \pmod N$, where $e_1 * d_1 \equiv 1 \pmod{\phi(N)}$ and $e_2 * d_2 \equiv 1 \pmod{\phi(N)}$

But this procedure can be rewritten as follows:

Alice encrypts her message: $c_2 \equiv m^{e_3} \pmod N$

Bob decrypts the ciphertext: $m \equiv c_2^{d_3} \pmod N$, $e_3 \equiv e_1 * e_2 \pmod{\phi(N)}$ and $d_3 \equiv d_1 * d_2 \pmod{\phi(N)}$

($e_3 * d_3 \equiv e_1 * d_1 * e_2 * d_2 \equiv 1 \pmod{\phi(N)}$ and $c_2^{d_3} \equiv (m^{e_3})^{d_3} \equiv m \pmod N$)

New version is equivalent to RSA with different keys (n, e_3), (n, d_3). Therefore security has not increased.

b. Consider the following

p =
510199234220351635769753579003640646511269683368666689283064468492360
478957885883733073466895536294535102461614047593850516003701938960081
2468312274731287

q =
104677858040236631540811598909166707511976842840505964536630741129499
224685702253698304193500442968305523686358763901573500616740416774094
79215017880761471

$e_1 = 65537$

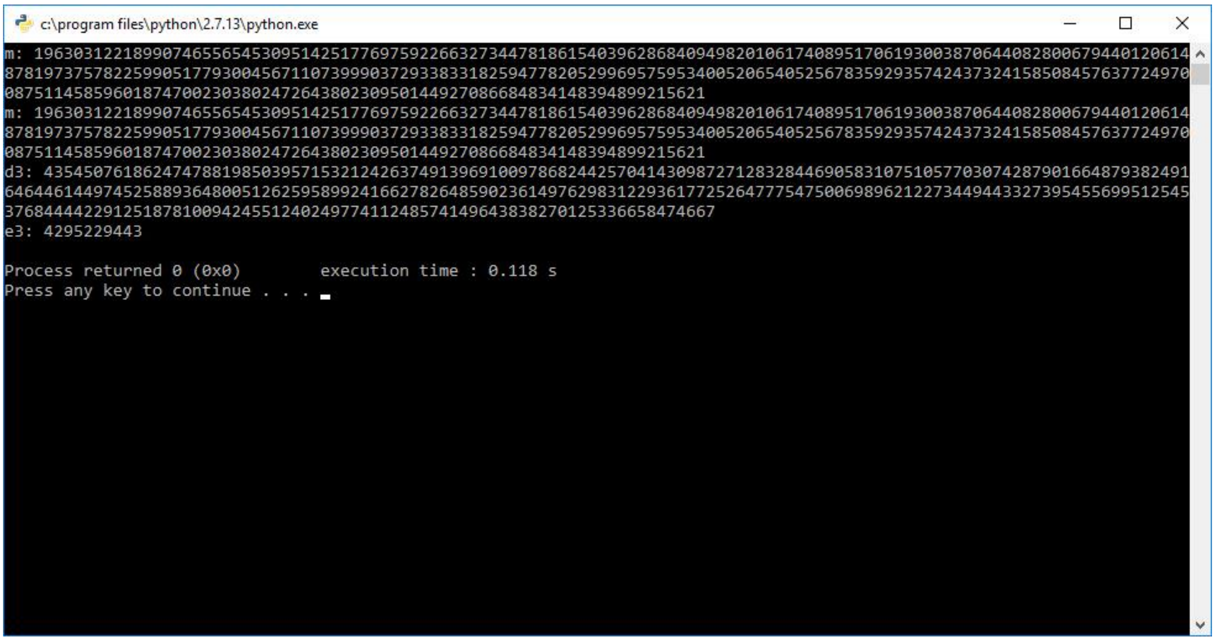
$e_2 = 65539$

c=
493463293694628543632717824483152110996815144346405999746218546713924
271393716352103453630314731535994643483208423227406355539606429535108
951960692292587865368917490218414372006590261848443504441655164271774
042488924390494483343994111890450561851024227047380917594733133111636
25490140702356058569205275317138

where c is the ciphertext as a result of the double RSA encryption with e_1 and e_2. Find the equivalent decryption key d_3 and decrypt the ciphertext with a single modular exponentiation operation. (**20 pts**)

$d_3 \equiv$
43545076186247478819850395715321242637491396910097868244257041430987271283284469058310751057703074287901664879382491646446144974525889364800512625958992416627826485902361497629831229361772526477754750069896212273449443327395455699512545376844442291251878100942455124024977411248574149643838270125336658474667

$m \equiv$
19630312218990746556545309514251776975922663273447818615403962868409498201061740895170619300387064408280067944012061487819737578225990517793004567110739990372933833182594778205299695759534005206540525678359293574243732415850845763772497008751145859601874700230380247264380230950144927086684834148394899215621



Q2.py

3. (**20 pts**) Consider the following RSA parameters:

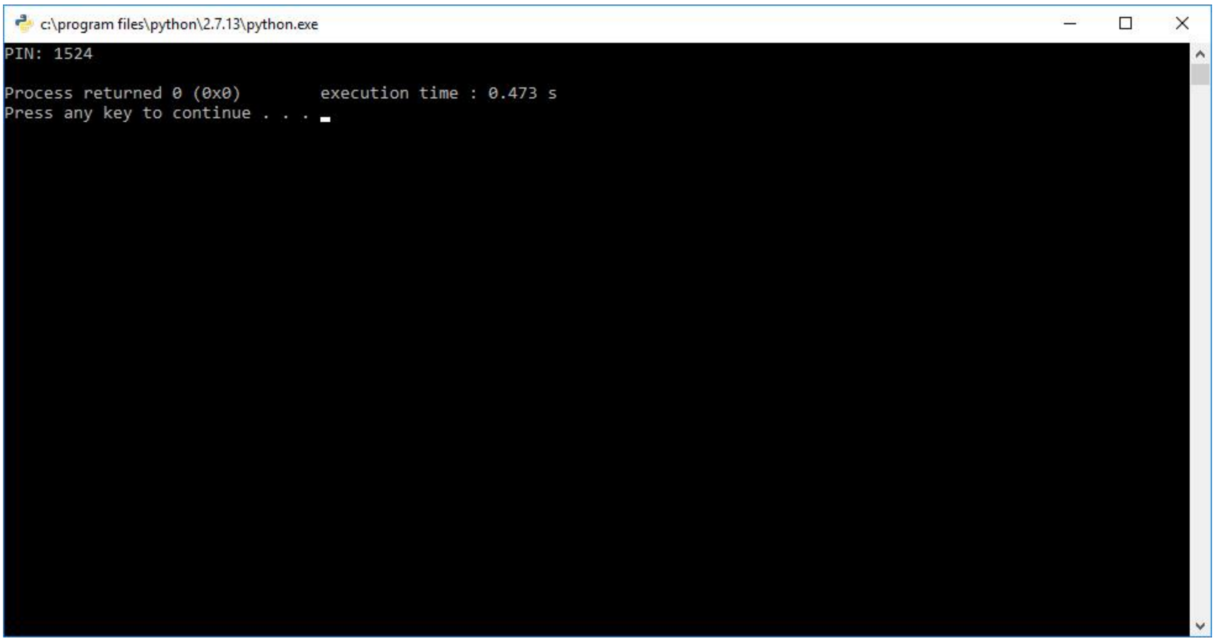
N:
59300007034639909939573758056469081496649095362931218335710263559636829491463214675399654309454248418861314244988626244340386564274798699371224710226178573100179350623455240677742280755859388396827300307411521308725481313615050599976223074746012316066224478374065904742491869819255200529839123356150617924773
e: 65537

Consider also the following ciphertext,

c:
47446751200838582684511548326026682588511648894181169756283983741637372571242927061754188680616949591184277685045578068075300868535431681867533253595068603738094583234430020018598479411043102722508296983050330949598947760334702343661636595593088242435796182825602462354582903741935054242424335737738087473281,

which is the encryption of my PIN of four digits. The textbook RSA without padding is used. Find my PIN.

Since there are only 10000 possible PIN's of four digits we can use brute force to compute $c' \equiv k^e \pmod N$ where k is in range [0000,9999] and compare if $c' = c$. When c' equals to c, meaning that we found our k and that's the four digit PIN.
PIN: 1524



Q3.py

4. (**30 pts**) Consider the following security game. Suppose that an attacker wants to decrypt the ciphertext c encrypted using the RSA algorithm and obtain the plaintext m, where $c = m^e \bmod N$. She knows neither the private key d nor the factorization of the modulus N. However, she can query an oracle (e.g., a program running on a remote server) with a ciphertext $c' \neq c$, and receives the corresponding plaintext $m' = c'^d \bmod N$.

   a. The attacker can decrypt c and recover m. Show how. (**10 pts**)
   If we pick such random integer r which is co-prime with N and multiply with c than we can compute $c' \equiv c * r^e \pmod N$. Send this c' to the Oracle and get the response
   $m' \equiv (c')^d \equiv (c * r^e)^d \equiv c^d * r^{ed} \equiv c^d * r^1 \pmod N$, where $e * d \equiv 1 \pmod{\phi(N)}$

$m' \equiv c^d * r \pmod N$. Then we can compute r^{-1} since it is invertible in modulus N because $\gcd(r,N) = 1$ and multiply m' with r^{-1} to get: $m' * r^{-1} \equiv c^d * r * r^{-1} \equiv c^d \equiv m \pmod N$, meaning that: $m \equiv m' * r^{-1} \pmod N$ and we have recovered the message m.

b. Consider the following RSA parameters and a challenge ciphertext

N:
1405516577481233118439046328334716692596774241775274294720766414591468679066619420682983795631811235875148981711983458427025180875779965334003708319528018833300358385638956728788170324290700704918018186863489724492590149701616910171477891255840236303807201070228410658811404013177269640375662812227489707122O3

e: 65537

c:
1073701452081810128821570359578312850076398611935021489585260889111451113123548578796513152984304917060579273635847036996508323650831497303880588177162273515806437805969230320212726501977058000133014358403790665132037058833111881196671090834779351294250383674721013894361656885598379010787464300281393535909888

The instructor (accessible via erkays@sabanciuniv.edu) is the oracle that you can submit your queries $c' \neq c$ via e-mail. Your queries must be in the following form

```
c’ = int("your query here as an integer")
```

I challenge you to find *m*. (**20 pts**)

m =
7523052463831744004978871986032816490556097266156744228028673908302271679740115365430299679545820622527654232994139872159117995532380124020235849063649090987061350187206754512852792883023374837010534110198777277506930972462956570211391375421406197414708793367649391719173730473732396590093983079425228189179

```
c:\program files\python\2.7.13\python.exe
m': 120402034061049960628218111909711136793099013099107787057903748477267562993246581824333179810106790393644009529285482451478168014966790953995488737879898316885941253266903643015596143055400220581436351563090027302223659679590697117170241040607927681665138200749531346830851443691882051320372085132068789965278
c': 396441850663747725629505969063777949730494373204477116310871989926615258049563902004538435433026301689641290832329178382765826274018019217476372643362236791311960983840145725700624366319406552501411354605252390965932193307084592952592475136165514277015687559774310208950793985072972713493063283085165425709 66
m: 752305246383174400497887198603281649055609726615674422802867390830227167974011536543029967954582062252765423299413987215911799553238012402023584906364909098706135018720675451285279288302337483701053411019877727750693097246295657021139137542140619741470879336764939171917373047373239659009398307942522818917 9

Process returned 0 (0x0)       execution time : 0.132 s
Press any key to continue . . .
```

Notes:

i. You can use the Python function **pow**(m, e, N) to compute modular exponentiation.

ii. You can use the following two Python functions to compute gcd and modular inverse

```
def egcd(a, b):
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q, r = b//a, b%a
        m, n = x-u*q, y-v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd, x, y

def modinv(a, m):
    gcd, x, y = egcd(a, m)
    if gcd != 1:
        return None  # modular inverse does not exist
    else:
        return x % m
```