CO 487 - Assignment 4

Bilal Khan (b54khan)
bilal.khan@student.uwaterloo.ca

November 14, 2024

Contents

1 1

1.1 (a)

- using the timestamp of when the file was encrypted as the random seed, if the attacker knows around when the file was encrypted they can more easily bruteforce the key, you should use a proper PRNG here
- Generating the prime p by sampling a prime randomly from around $\sqrt{n}$ and then choosing q as the next prime after it. We can exploit this by only searching for primes close to $\sqrt{n}$ and the next prime after that. You should generate two independently random primes.
- Using ECB mode, identical plaintext blocks will be encrypted to identical ciphertext blocks.
- The encryption code checks if $\gcd(e, n) = 1$, not if $\gcd(e, \phi(n)) = 1$.

1.2 (b)

Calling factorint here is faster than using the sqrt method.

```
def do_decryption():
  # Read and parse the JSON data structure
  with open("encrypted_assignment.json.txt", 'r') as fh:
      input = json.loads(fh.read())
  n = input["n"]
  e = input["e"]
  c_1 = input["c_1"]
  c_2 = string2bytes(input["c_2"])

  # TODO after midterms are over

  p, q = factorint(n).keys()
  #p = int(math.sqrt(n) + 1)
  #while n % p != 0:
  #    p -= 1
  #q = n // p
```

```
assert (p * q == n)
assert (isprime(p))
assert (isprime(q))

phi = (p - 1) * (q - 1)
d = mod_inverse(e, phi)

aes_key_int = pow(c_1, d, n)
aes_key = aes_key_int.to_bytes(16, byteorder='big')

cipher = Cipher(algorithms.AES(aes_key), modes.ECB()).decryptor()
unpadder = padding.PKCS7(128).unpadder()

plaintext = cipher.update(c_2) + cipher.finalize()
plaintext = unpadder.update(plaintext) + unpadder.finalize()

# write the decrypted assignment to a file
with open("assignment_out.pdf", 'wb') as fh:
    fh.write(plaintext)
```

1.3 (c)

The secret is 17853

2 2

2.1 (a)

To prove the contrapositive, if we have an algorithm that can solve CDH then given g^a, g^b, we can compute g^{ab}, and return the triple who's third element is g^{ab}, solving DDH.

2.2 (b)

To prove the contrapositive, if we have an algorithm that can solve DLP, then given g^a, g^b, we can compute the values of a, b, multiply them together, then compute g^{ab} efficiently as it is just a simple exponentiation, solving CDH.

2.3 (c)

According to wikipedia (https://en.wikipedia.org/wiki/Legendre_symbol), the value of a Legendre symbol x is 1 when it can be represented as an even power of a generator g of the group $x = g^i$ if i is even (half of all possible values), so approximately half of the values of the Legendre symbol in some prime order group is 1 / is square modulo p (SQP). We can use this property to give us an edge when guessing to solve this.

If g^a, g^b are both SQP, then g^{ab} must also be a SQP since the power ab can be written as a power of power of twos. In the cases where both g^a, g^b are SQP but the proposed third element of the tuple is not, we can always correctly guess that this is not a valid triple. In all other cases we guess randomly with a $1/2$ probability of being correct. The chances of each element of a tuple being a SQP is independently $1/2$, so the chance of the first two elements being SQP but the third not being is $1/2 \times 1/2 \times 1/2 = 1/8$, and in every one of these cases we will guess correct. The probability of our adversary guessing correctly is $1/2 + 1/8 = 5/8$.

3 3

3.1 (a)

$$
\begin{aligned}
r_1 &= g^{s_0} \bmod p = 21^{123} \bmod 167 = 126 \\
s_1 &= g^{r_1} \bmod p = 21^{126} \bmod 167 = 57 \\
t_1 &= h^{r_1} \bmod p = 6^{126} \bmod 167 = 7 \\
r_2 &= g^{s_1} \bmod p = 21^{57} \bmod 167 = 47 \\
s_2 &= g^{r_2} \bmod p = 21^{47} \bmod 167 = 124 \\
t_2 &= h^{r_2} \bmod p = 6^{47} \bmod 167 = 54 \\
r_3 &= g^{s_2} \bmod p = 21^{124} \bmod 167 = 141 \\
s_3 &= g^{r_3} \bmod p = 21^{141} \bmod 167 = 152 \\
t_3 &= h^{r_3} \bmod p = 6^{141} \bmod 167 = 150
\end{aligned}
$$

3.2 (b)

$$
\begin{aligned}
r_i &= g^{s_{i-1}} \bmod p \\
s_i &= g^{r_i} \bmod p \\
t_i &= h^{r_i} \bmod p \\
h &= g^u \bmod p
\end{aligned}
$$

$$
\begin{aligned}
t_1 &= h^{r_1} \bmod p \\
&= (g^u)^{r_1} \bmod p \\
\log_g t_1 &= u r_1 \bmod p \\
r_1 &= \frac{1}{u} \log_g t_1 \bmod p
\end{aligned}
$$

Given that we have r_1 now, we can now compute $s_1 = g^{r_1} \bmod p$. Given s_1, we can then compute $r_2 = g^{s_1} \bmod p$, $t_2 = h^{r_2} \bmod p$, and on.

3.3 (c)

We can add a secure one-way hash function on the output of the PRG to make it impossible to reverse the PRG by taking a log: $t_i = H(h^{r_i} \bmod p)$.

4 4

4.1 (a)

Left does an inversion before an addition and right does an addition before an inversion.

The very first operation we need to do when adding two points is an inversion, to compute $\delta = \dfrac{y_2 - y_1}{x_2 - x_1}$, since additions/subtractions don't count. After that is computed only then can we do

the multiplications to compute the new points so the left graph is the result of computing $P + Q$ and the right graph is $P + P$, in which you can reorder operations to do a multiplication before an inversion.

4.2 (b)

We compute a key by performing a sclaar multiplciation of a point with a key, we can do this using the double-adn-add algorithm (https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication). For zero bits we double, for one bits we double then add. Reading the power consumption graphs for $P + P$ and $P + Q$, the sequence is:

(double add) (double) (double) (double add) double
in bits:
1 0 0 1 0

4.3 (c)

We will compute $9P = 8P + P = 2(2(2P)) + P$.

4.4 (d)

A simple modification would be to always compute both the double and the add, but only use the result of the add if the value is a one bit.

5 5

5.1 (a)

$$
\begin{aligned}
c_2 &= s'b + e'' + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
&= s'(As + e) + e'' + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
&= s'As + s'e + e'' + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
c_1 s &= (s'A + e')s \bmod q \\
&= s'As + e's \bmod q \\
c_2 - c_1 s &= (s'As + s'e + e'' + \left\lfloor \frac{q}{2} \right\rfloor m) - (s'As + e's) \bmod q \\
&= s'e + e'' - e's + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
&= (s'e + e'' - e's) + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
c2 - c1s &= \left\lfloor \frac{q}{2} \right\rfloor m \bmod q
\end{aligned}
$$

Assuming error is small, then $m \approx \frac{c_2 - c_1 s}{\lfloor \frac{q}{2} \rfloor} \bmod 2$.

Checkign the values in numpy:

```python
import numpy as np

n = 4
q = 3329

c1 = np.array([220, 1211, 2479, 1812])
c2 = 1799
s = np.array([-1, 1, 1, 0])

tmp = (c2 - np.dot(c1, s)) % q
m_hat = abs(tmp / (q // 2))

print(f"m = {m_hat}")
# m = 0.9963942307692307
```

Which matches our expected value of $m = 1$.

5.2 (b)

if $e = 0$, then $b = As$, which we can easily invert to get the secret key $s = A^{-1}b$.

5.3 (c)

if $e' = 0$, then

$$
\begin{aligned}
c_1 &= s'A \bmod q \\
s' &= A^{-1}c_1 \bmod q \\
c_2 &= s'b + e'' + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
c_2 - s'b &= e'' + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
\frac{c_2 - s'b}{\left\lfloor \frac{q}{2} \right\rfloor} &\approx m \bmod 2
\end{aligned}
$$

e' can be drawn from a small range of small values, and we know that q is much larger, so and attacker could recover a value of m close to its real value.

5.4 (d)

if $b = As + s = (A + I)s$, then assuming $A + I$ is invertible, we could sovle for s as $s = (A + I)^{-1}b$.

5.5 (e)

As in (c):

$$
\begin{aligned}
c_1 &= s'A + s' \bmod q \\
c_1 &= (A + I)s' \bmod q \\
s' &= (A + I)^{-1}c_1 \bmod q \\
c_2 &= s'b + e'' + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
c_2 - s'b &= e'' + \left\lfloor \frac{q}{2} \right\rfloor m \bmod q \\
\frac{c_2 - s'b}{\left\lfloor \frac{q}{2} \right\rfloor} &\approx m \bmod 2
\end{aligned}
$$

Given that $A + I$ is invertible and e' can be drawn from a small range of small values, and we know that q is much larger, so and attacker could recover a value of m close to its real value.