Crypto - HW 4

Hagai Ben Yehuda, ID num: 305237000
Jonathan Bauch, ID num: 204761233

1

Listing 1: Q1 Code

```
import random
from collections import Counter

m = 90256390764228001
Zm = Integers(m)

def Q1():
    a_values = [Zm(random.randint(2, m-1)) for _ in range(100)]
    gcds = [gcd(a, m) for a in a_values]
    max_is = [max_i(a) for a in a_values]
    i_counts = Counter(max_is)
    print "number of a's s.t. gcd(a,m)!=1:", sum(1 for g in gcds if g != 1)
    print "number of a's with max i=[5,..,1,None]:"
    for i in [5,4,3,2,1,None]:
        print '#i=%s: %s' % (i, i_counts[i])

def max_i(a):
    vals = [a^((m-1)/(2^i)) for i in range(0, 5+1)]
    triplets = zip(vals[:5], vals[1:], range(1,5+1))
    for prev, current, i in reversed(triplets):
        if current != Zm(1) and current != Zm(-1) and prev == Zm(1):
            return i
    return None

if __name__ == '__main__':
    Q1()
```

Listing 2: Q1 Output

```
number of a's s.t. gcd(a,m)!=1: 0
number of a's with max i=[5,..,1,None]:
#i=5: 35
#i=4: 54
#i=3: 0
#i=2: 0
#i=1: 0
#i=None: 11
```

Explanation: TODO

2

We construct a randomized algorithm A' that operates on input $m = pq$ as follows:

1. Draw $y \in Z_m^*$ uniformly (we do that by drawing from $\{1, ..., m-1\}$ and making sure $\gcd(y, m)$ is zero, if it isn't we can factor m using y). .
2. Execute A on input $y^2 \pmod{m}$ and set x to be its result (note that since $y^2 \pmod{m}$ is a quadratic residue we will get a number and not "go catch a Stellagama stellio").
3. If $x = \pm y \pmod{m}$ and this step was executed less then c times (c being a constant positive integer that will affect the probability of success) go to step one if this step was executed c times, return 0.
4. Calculate $w = xy^{-1} \pmod{m}$.
5. Set $k = w + 1 \pmod{m}$
6. Set $z = \frac{k}{2}$.
7. set $q = gcd(z, m)$ and return $(q, \frac{m}{q})$

We shall now prove that A' runs in $O(t(n))$ and finds a factorization for m with probability $1 - \frac{1}{2^c}$. First note that m executes steps 1 through 3 at most c time (from the restriction in step 3) and each steps takes $O(t(n))$ steps. In addition for each execution A' passes step 3 with probability $\frac{1}{2}$, that is because y^2 has four roots in Z_m^*, and only two of them are $\pm y$, since y was chosen uniformly, the probability that the root that A returns for y^2 is $\pm y$ is $\frac{2}{4} = \frac{1}{2}$. Now we prove that if A' passes step 3 it returns a correct factorization.

From the CRT we can write $x = wy$ with

$$w = a_1(q^{-1} \pmod{p})q + a_2(p^{-1} \pmod{q})p$$

and $a_i \in \{\pm 1\}$ (this is because as stated in the lecture, if x is a root of y then it can be written as ly with l being a root of 1 in m), since we chose x such that $x \neq \pm y$, we know that $a_1 \neq a_2$.
Assume without loss of generality that $a_1 = 1$ and $a_2 = -1$, thus we have (w is from step 4)

$$w = (q^{-1} \pmod{p})q - (p^{-1} \pmod{q})p$$

Note that from fermat's little theorem we have

$$(q^{-1} \pmod{p}) = q^{p-2} + cp$$

$$(p^{-1} \pmod{q}) = p^{q-2} + rq$$

Thus

$$w = q^{p-1} - p^{q-1} \pmod{pq}$$

Note that

$$q^{p-1} + p^{q-1} = 1 \pmod{p}$$

and

$$q^{p-1} + p^{q-1} = 1 \pmod{q}$$

Hence

$$q^{p-1} + p^{q-1} \pmod{pq}$$

Thus

$$w + 1 = q^{p-1} - p^{q-1} + 1 = q^{p-1} - p^{q-1} + q^{p-1} + p^{q-1} = 2q^{p-1} \pmod{pq}$$

Therefore when we calculate z in step 6 we obtain q^{p-1} and obviously $\gcd(q^{p-1}, pq) = q$, and thus we indeed recover q and p in step 7 as required, since we got to step 4 in $O(t(n))$ steps and 4 through 7 also take $O(t(n))$ steps, A' is an algorithm as request. Randomization is required in our algorithm as we must get a root that is differs from the root we know not only by sign. Since we have no knowledge of what root A will return, and since we cant find another root by ourselves, we must hope A returns a different root, by choosing y randomly many times, the probability we will indeed find a root that is different not only by sign approaches 1.

3

4

5

We will construct B as follows: given number $z = g^y$

0. if z == 1, return $y = 0$
1. choose random $r \in [1, p-1]$
2. calculate $z^r = g^{yr}$, r^{-1} (using iterated squaring, and xgcd)
3. run $A(z^r)$
   - if it succeeded and returned yr, then return y by multiplying by r^{-1}
   - otherwise, return to step 1
   - if failed 1000 times, quit

Correctness: As shown in a previous exercise, if r is uniformly distributed on $[1, p-1]$ and $y \in [1, p-1]$ then ry is uniformly distributed on that range. And indeed $y \in [1, p-1]$ because in step 0 we rule out $y = 0$. Therefore there is a $\frac{1}{1000}$ probability that yr is in the exponents that A can successfully find. Therefore there is a $\frac{1}{1000}$ that step 3 succeeds. We then get:

$$\Pr[B \text{ succeeds}] = 1 - \left(\frac{999}{1000}\right)^{1000} \approx 0.63 > 0.5$$

Run time: step 0 is $O(1)$, choosing r is $O(n)$ when $n = \log_2(p)$. Calculating z^r by repeated squaring is $O(n^2)$, and xgcd is $O(n)$. A is poly-time in n, therefore it runs in time $O(n^k)$ for some $k \geq 1$. then we get that a single iteration of B takes $O(n^2 + n^k)$. Since there is a constant number of iterations, the total runtime is $O(n^2 + n^k) = O\left(max\left\{n^2, n^k\right\}\right)$.

6

6.a

Decryption: Given $\langle c_1, c_2 \rangle$:

- Compute c_1^x
- If $c_1^x = c_2$ return 0,
- Otherwise return 1.

Run-Time: it is clear that this method is efficient, as it requires one exponentiation (which can be done efficiently by iterated squaring).
Correctness:

- If the encrypted message is $b = 0$, then $c_1^x = (g^y)^x = (g^y)^x = h^y = c_2$. Therefore the decryption will succeed with probability 1.

- If the encrypted message is $b = 1$, then $c_1^x = g^{xy}$, $c_2 = g^z$. Since y is random and independent from z, then so is yx. This mean that $\Pr[yx \equiv z \pmod{p}] = \frac{1}{p}$. Therefore $\Pr[c_1^x \neq c_2] = 1 - \frac{1}{p}$, meaning that the decryption succeeds with some negligible error probability.

6.b

7

Let $i \leftarrow U_t$ be the random index A_1 chooses. $b \leftarrow U_{0,1}$. $c^i = E_{pk}(m_b^i)$. Denote $A(x)$ the answer of an adversary A, given a cipher x. We have:

$$\begin{aligned}
&\frac{1}{2} + \varepsilon \\
(\varepsilon\text{-CPA secure}) \geq{} &\Pr\left[A_1 \text{ wins}\right] \\
(\text{by definition}) ={} &\Pr\left[A_1(c^i) = b\right] \\
(\text{total probability}) ={} &\frac{1}{2} \sum_{d \in \{0,1\}} \Pr\left[A_1(c^i) = d \mid b = d\right] \\
(\text{total probability}) ={} &\frac{1}{2t} \sum_{d \in \{0,1\}} \sum_{k=1}^{t} \Pr\left[A_1(c^i) = d \mid b = d \wedge i = k\right] \\
(\text{by definition}) ={} &\frac{1}{2t} \sum_{d \in \{0,1\}} \sum_{k=1}^{t} \Pr\left[A_1(E_{pk}(m_d^k)) = d \mid b = d \wedge i = k\right] \\
(\text{by definition}) ={} &\frac{1}{2t} \sum_{d \in \{0,1\}} \sum_{k=1}^{t} \Pr\left[A_{mult}\left(E_{pk}(m_0^1, \ldots, m_0^{k-1}, m_d^k, m_1^{k+1}, \ldots, m_1^t)\right) = d\right] \\
(\text{sum reorder}) ={} &\frac{1}{2t} \Bigg(\overbrace{\sum_{d \in \{0,1\}} \Pr\left[A_{mult}\left(E_{pk}(m_d^1, \ldots, m_d^t)\right) = d\right]}^{=2\cdot\Pr[A_{mult} \text{ wins}]} \\
&+ \sum_{k=1}^{t-1} \overbrace{\sum_{d \in \{0,1\}} \Pr\left[A_{mult}\left(E_{pk}(m_0^1, \ldots, m_0^k, m_1^{k+1}, \ldots, m_1^t)\right) = d\right]}^{=1} \Bigg) \\
(\text{simplification}) ={} &\frac{1}{2t} \Big(2 \cdot \Pr\left[A_{mult} \text{ wins}\right] + t - 1 \Big) \\
(\text{simplification}) ={} &\frac{1}{t} \Pr\left[A_{mult} \text{ wins}\right] + \frac{1}{2} - \frac{1}{2t}
\end{aligned}$$

Therefore:

$$\Pr\left[A_{mult} \text{ wins}\right] \leq t\left(\frac{1}{2} + \varepsilon - \frac{1}{2} + \frac{1}{2t}\right) = \frac{1}{2} + t \cdot \varepsilon = \frac{1}{2} + \varepsilon_t \qquad \square$$

8

8.a

First, note that if p, q are n bit numbers, then $m = pq$ has at most $2n = O(n)$ bits. Computing a^{2^t} using iterated squaring involves t steps of (modular) squaring a number in the range $[0, m-1]$:

$$a_0 = a, a_1 = a_0^2 = a^2, a_2 = a_1^2 = a^{2^2}, a_3 = a_2^2 = a^{2^3}, \dots, a_t = a_{t-1}^2 = a^{2^t}$$

Therefore the number of modular multiplications of $O(n)$ bit numbers is exactly t.

8.b

Knowing the factorization of m allows us to compute $\phi(m) = (p-1)(q-1)$. Then, by Euler's theorem we know that $a^{2^t} \equiv a^{2^t \mod \phi(m)} \pmod{m}$. Therefore to compute $a^{2^t} \pmod{m}$ we need to calculate:

$$a_0 = a, a_1 = a_0^2 = a^2, \dots, a_k = a_{k-1}^2 = a^{2^k}$$

where k is the number of bits of $2^t \mod \phi(m)$. afterwards we multiply the elements according to the binary representation of $\phi(m)$. $\phi(m)$ can have no more than $2n$ bits, therefore we need to perform at most $k + k = 2k \leq 4n$ modular multiplications. Note that if $2^t < \phi(m)$ then we resort to the first method and perform exactly t multiplication.
So to summarize, we perform no more than $\min\{t, 4n\}$ multiplications.