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: Any a with an i as defined in the question is a witness that m is not prime. Define $b := a^{(m-1)/2^i}$. b is square root of $1 \pmod m$ because:

$$b^2 = (a^{(m-1)/2^i})^2 = a^{(m-1)/2^{i-1}} \equiv 1 \pmod m$$

But $b \not\equiv \pm 1 \pmod m$, therefore the polynomial $(x-1)^2$ has more than 2 roots in $\mathbb{Z}_m$, which implies that $\mathbb{Z}_m$ is not a field (otherwise it would be a contradiction to the fundamental theorem of algebra). Therefore m is not a power of a prime number, and specifically, it's not a prime number.

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 1, 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

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

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

Listing 3: Q3 Code

```
def Q3():
    p = random_prime(2^46, proof=True, lbound=2^45)
    q = random_prime(2^48, proof=True, lbound=2^47)
    m = p * q
    print 'p:', p
    print 'q:', q
    print 'm:', m
    cs = [1, 212321, 35432, 0, -1]
    xs = [1, 32151, 7]
    max_iterations = 5 * int(m^0.25)
    for e in [2, 1]:
        print '*' * 20
        print 'Running for f=x^%d+c' % e
        for c in cs:
            for x0 in xs:
                f = lambda x: (x^e + c) % m
                print '- x0=%06s, c=%06s:' % (x0, c),
                factor, i = rho(f, x0, m, max_iterations)
                if factor == p:
                    text = 'p'
                elif factor == q:
                    text = 'q'
                else:
                    text = 'ran out of time'
                relative_iterations = float(i) / p^0.5
                print 'factor = %s (after %s iterations=%s * sqrt(p))' % \
                        (text, i, relative_iterations)

def rho(f, x0, m, max_iterations):
```

```python
    x = x0
    y = x0
    g = 1
    for i in xrange(max_iterations):
        x = f(x)
        y = f(f(y))
        g = gcd(m, y - x)
        if g > 2 and g < m:
            break
    else:
        return 1, max_iterations
    return g, i + 1

if __name__ == '__main__':
    Q3()
```

Listing 4: Q3 Output

```
p: 57021442427041
q: 168499908198593
m: 9608107814307764528141353313
*******************
Running for f=x^2+c
- x0=     1, c=     1: factor = q (after 3022422 iterations
    =0.400254296232311 * sqrt(p))
- x0= 32151, c=     1: factor = p (after 1779380 iterations
    =0.235640320785731 * sqrt(p))
- x0=     7, c=     1: factor = p (after 7117520 iterations
    =0.942561283142924 * sqrt(p))
- x0=     1, c=212321: factor = q (after 2124493 iterations
    =0.281343058833436 * sqrt(p))
- x0= 32151, c=212321: factor = p (after 3183580 iterations
    =0.421596180943383 * sqrt(p))
- x0=     7, c=212321: factor = p (after 6019221 iterations
    =0.797115381380148 * sqrt(p))
- x0=     1, c= 35432: factor = p (after 3216320 iterations
    =0.425931884448270 * sqrt(p))
- x0= 32151, c= 35432: factor = q (after 4655137 iterations
    =0.616472016085111 * sqrt(p))
- x0=     7, c= 35432: factor = p (after 4824480 iterations
    =0.638897826672404 * sqrt(p))
- x0=     1, c=     0: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0= 32151, c=     0: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=     7, c=     0: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=     1, c=    -1: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0= 32151, c=    -1: factor = p (after 9469164 iterations
    =1.25398556943019 * sqrt(p))
```

```
- x0=      7, c=     -1: factor = q (after 5980593 iterations
    =0.791999939871695 * sqrt(p))
*******************
Running for f=x^1+c
- x0=      1, c=      1: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0= 32151, c=      1: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      7, c=      1: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      1, c=212321: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0= 32151, c=212321: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      7, c=212321: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      1, c= 35432: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0= 32151, c= 35432: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      7, c= 35432: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      1, c=      0: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0= 32151, c=      0: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      7, c=      0: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      1, c=     -1: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0= 32151, c=     -1: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
- x0=      7, c=     -1: factor = ran out of time (after 49502765
    iterations=6.55556847013041 * sqrt(p))
```

3.a

Based on the results of this small scale experiment we can recommend not to choose $c = 0$ or $x_0 = 1, c = -1$ as the algorithm would fail to factor a big number as in our case (in reasonable time). Other than that, for all the other values we tried, the time it took to factor m was about $[0.3, 3] \cdot \sqrt{p}$ iterations, which is the expected average case for this algorithm. Without further investigation we cannot recommend particular values which would give good results for any values of m.

3.b

With the linear function $f(x) = x + c$ non of the program executions terminated (in the time frame of $5m^{\frac{1}{4}}$ iterations). This is because the function f is indeed not random at all. Observe that $f^k(x) = x + k \cdot c$. The algorithm ends when:

$$
\begin{aligned}
& gcd(m, y - x) \neq 1 \\
\iff & gcd(m, f^{2k}(x_0) - f^k(x_0)) \neq 1 \\
\iff & gcd(m, x_0 + 2kc - x_0 - kc) \neq 1 \\
\iff & gcd(m, kc) \neq 1
\end{aligned}
$$

Therefore it will halt only after $min\{p, q\}$ steps (unless $c \mid m$, which is unlikely).

4

Listing 5: Q4 Code

```
import string
TEXT = 'THESE VIOLENT DELIGHTS HAVE VIOLENT ENDS'

def Q4():
    p = get_prime(min_digits=82)
    q = get_prime(min_digits=77)
    N = p * q
    phi = (p-1) * (q-1)
    while True:
        e = randint(2, phi - 1)
        if gcd(e, phi) == 1:
            break
    d = inverse_mod(e, phi)
    print 'N:', N
    print 'p:', p
    print '  ', 10^82
    print 'q:', q
    print '  ', 10^72
    print 'e:', e
    print 'd:', d
    print 'p-1:', factor(p-1)
    print 'q-1:', factor(q-1)
    print
    print 'Message         :', TEXT
    encoded = encode(TEXT)
    print 'Encoded message:', encoded
    encrypted = encrypt(encoded, N, e)
    print 'Encrypted      :', encrypted
    decrypted = decrypt(encrypted, N, d)
    print 'Decrypted      :', decrypted
    decoded = decode(decrypted)
    print 'Decoded         :', TEXT

def get_prime(min_digits):
```

```
    while True:
        r = random_prime(10^(min_digits+1),
                         proof=True,
                         lbound=10^(min_digits))
        s = 2 * r + 1
        if is_prime(s):
            return s

def encode(s):
    encoded = 0
    for c in s:
        if c == ' ':
            n = 0
        elif c in string.ascii_uppercase:
            n = ord(c) - ord('A') + 1
        else:
            raise ValueError('Unexpected char')
        encoded += n
        encoded *= 100
    encoded //= 100
    return encoded

def decode(number):
    chars = []
    while number > 0:
        n = number % 100
        if n == 0:
            chars.append(' ')
        elif 1 <= n <= 26:
            chars.append(chr(n - 1 + ord('A')))
        else:
            raise ValueError('Unexpected number')
        number //= 100
    return ''.join(chars)


def encrypt(message, N, e):
    return int(pow(message, e, N))

def decrypt(cipher, N, d):
    return int(pow(cipher, d, N))

if __name__ == '__main__':
    Q4()
```

Listing 6: Q4 Output

```
N:  51527542493862786303577465885471668548570396233720476495170277840423812
80905630205152564389943779044693022801247757232332025764038181429002042804
9240861879737809
p:  30718380893785475171199444438743005772348145749926030804306936327721256
660504967967
    10000000000000000000000000000000000000000000000000000000000000000000000
000000000000
q:  16774172659694814042495173722936995652294507327634958574531075569930797
96218127
    10000000000000000000000000000000000000000000000000000000000000000000000
00
e:  19288678490240985222124141066365571492349497588991474234250948705183089
89163627913086030523204100406716339794850751959363884306248817301813209447
6071978848497353
d:  26261901487479055314548622968128005203023631558531980563573243400081398
79144886513911331155009863180672085632900508319546043091327155014753280859
7681211927132209
p-1: 2 * 15359190446892737585599722219371502886174072874963015402153468163
860628330252483983
q-1: 2 * 83870863298474070212475868614684978261472536638174792872655377849
6539898109063

Message          : THESE VIOLENT DELIGHTS HAVE VIOLENT ENDS
Encoded message: 2008051905002209151205142000040512090708201900080122050022
209151205142000005140419
Encrypted        : 38049395411248730657737551168183646558568374868492034905
58073367607316289348277654570067097992487693941756934915603229447662661282
6675250639296659107673040165451
Decrypted        : 20080519050022091512051420000405120907082019000801220500
2209151205142000005140419
Decoded          : THESE VIOLENT DELIGHTS HAVE VIOLENT ENDS
```

Explanation: TODO

4.a

As hinted at the course forum (http://tau-crypto-f16.wikidot.com/forum/t-2034665/some-ex04-questions-for-q2-q4-and-q5) the way we chose the primes is as follows:

1. Get a random prime number r of the requested size,
2. Calculate $s = 2r + 1$
3. Check if s is also a prime:
   If it is - return it.
   Otherwise go the step 1.

In other words: p, q were drawn randomly from the set $\left\{x \in \text{Primes} \mid \frac{x-1}{2} \in \text{Primes}\right\}$, with the restriction on minimum size of the primes.

4.b

Required results are in Listing 6: Q4 Output.

5

We construct a polytime algorithm A' that on input p, g, g^x does the following:

- Draw $x \in \mathbb{Z}^*{}_p$ uniformly.
- Execute A on p, g, g^{x+y} (note that $g^{x+y} = g^x g^y$), set z to be the result.
- If $g^z = g^{x+y}$ return $z - y$, else if this is the 700'th time return 0, else go to the first step.

First note that this algorithm is polynomial as it executes A at most 700 times, and A is polynomial. For each iteration the probability of landing within the subset of $x's$ for which A finds and inverse is $\frac{1}{1000}$ as the sum of a uniform random variable and a constant is uniform. Hence with probability $\frac{1}{1000}$ we obtain the correct z in the last step, note that

$$g^{z-y} = g^z g^{-y} = g^{x+y} g^{-y} = g^x$$

Thus $z - y$ is a solution to the DL problem. The last step in A fails only if $x + y$ is not inside the set for which A solves the DL problem, this probability is $\frac{1}{1000}$ because $x + y$ distributes uniformly over $\mathbb{Z}^*{}_p$.

Because A' makes 700 tries before returning with a false result, the probability that A' fails is the probability that A fails at each attempt which is $(1 - \frac{1}{1000})^{700} < \frac{1}{2}$. Thus A' is an algorithm as requested.

6

6.a

We construct a the decryption function:

$$Dec(c_1, c_2) = \begin{cases} 1 & \text{if } c_1^x = c_2 \\ 0 & \text{else} \end{cases}$$

If $b = 0$, then $c_2 = h^y = g^{xy} = c_1^x$, thus if $b = 0$ Dec returns the correct result.
If $b = 1$ then given z there is exactly one value of y for which $g^y = g^{zx}$ since g is a multiplicative generator, if $g^y = g^{zx}$ then $y = zx \pmod{p-1}$, the only case in which we decrypt a 1 to 0 is if $y = xz$ which happens with probability at most $\frac{1}{p-1}$. Thus correct and efficient decryption is possible except for a negligible probability.

6.b

Assume that this encryption scheme is not ϵCPA secure, then there is a polynomial adversary A that wins the adversarial indistinguishability test with probability $> \frac{1}{2} + \epsilon$. We construct a polynomial time adversary A' that shows DDH is not hard: Given input (g^x, g^y, g^z) our algorithm does the following:

- Supply A with (p, g, g^x).
- Get the two messages from A assume WLOG A replays with $m_0 = 0$, $m_1 = 1$ (if this is not the case we can construct an algorithm B that is based on A and wins with the same probability, since if the messages are in a different order B can change the order and if both messages have the same value A can only guess which bit was chosen as both will be encrypted to the same value and B can supply us with two messages and also guess and win with the same probability).
- Supply A with (g^y, g^z), if A returns 1 return 0, else return 1.

We shall now show that A' distinguishes (g^x, g^y, g^z) from (g^x, g^y, g^{xy}):

$$\begin{aligned}
&\Pr_{x,y \leftarrow U_{\mathbb{Z}^*_p}, z=xy}(A'(g^x, g^y, g^z) = 1) - \Pr_{x,y,z \leftarrow U_{\mathbb{Z}^*_p}}(A'(g^x, g^y, g^z) = 1) \\
&= \Pr(A'(g^x, g^y, g^z) = 1 | x, y \leftarrow U_{\mathbb{Z}^*_p}, z = xy) - \Pr(A'(g^x, g^y, g^z) = 1 | x, y, z \leftarrow U_{\mathbb{Z}^*_p}) \\
&= \Pr(A \text{ wins } | b = 1) - \Pr(A \text{ loses } | b = 0) \\
&= 2[\Pr(A \text{ wins } \cap b = 1) - \Pr(A \text{ loses } \cap b = 0)] \\
&= 2[\Pr(A \text{ wins } \cap b = 1) - \Pr(b = 0) + \Pr(A \text{ wins } \cap b = 0)] \\
&= 2[\Pr(A \text{ wins }) - \Pr(b = 0)] \\
&\geq 2[\frac{1}{2} + \epsilon - \frac{1}{2}] = 2\epsilon
\end{aligned}$$

Note that in our calculation we refer to the probability that x, y, z are drawn uniformly or $z = xy$ (this is b as defined in the adversarial indistinguishability test) , each case has probability $\frac{1}{2}$ as we are in a distinguisher setup and thus are supplied with a sample from each distribution with equal probability (otherwise the streams are distinguishable by always saying that the current input originated from the stream with higher probability to be sampled). Thus A' is a distinguisher as required.

7

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

$$\begin{aligned}
&\frac{1}{2} + \varepsilon \\
(\varepsilon\text{-CPA secure}) \geq\ &\Pr\left[A_1 \text{ wins}\right] \\
(\text{by definition}) =\ &\Pr\left[A_1(c^i) = b\right] \\
(\text{total probability}) =\ &\frac{1}{2}\sum_{d\in\{0,1\}} \Pr\left[A_1(c^i) = d \mid b = d\right] \\
(\text{total probability}) =\ &\frac{1}{2t}\sum_{d\in\{0,1\}}\sum_{k=1}^{t} \Pr\left[A_1(c^i) = d \mid b = d \wedge i = k\right] \\
(\text{by definition}) =\ &\frac{1}{2t}\sum_{d\in\{0,1\}}\sum_{k=1}^{t} \Pr\left[A_1(E_{pk}(m^k_d)) = d \mid b = d \wedge i = k\right] \\
(\text{by definition}) =\ &\frac{1}{2t}\sum_{d\in\{0,1\}}\sum_{k=1}^{t} \Pr\left[A_{mult}\left(E_{pk}(m^1_0, \ldots, m^{k-1}_0, m^k_d, m^{k+1}_1, \ldots, m^t_1)\right) = d\right] \\
(\text{sum reorder}) =\ &\frac{1}{2t}\Bigg(\overbrace{\sum_{d\in\{0,1\}} \Pr\left[A_{mult}\left(E_{pk}(m^1_d, \ldots, m^t_d)\right) = d\right]}^{=2\cdot\Pr[A_{mult} \text{ wins}]} \\
&+ \sum_{k=1}^{t-1}\overbrace{\sum_{d\in\{0,1\}} \Pr\left[A_{mult}\left(E_{pk}(m^1_0, \ldots, m^k_0, m^{k+1}_1, \ldots, m^t_1)\right) = d\right]}^{=1}\Bigg) \\
(\text{simplification}) =\ &\frac{1}{2t}\left(2 \cdot \Pr\left[A_{mult} \text{ wins}\right] + t - 1\right) \\
(\text{simplification}) =\ &\frac{1}{t}\Pr\left[A_{mult} \text{ wins}\right] + \frac{1}{2} - \frac{1}{2t}
\end{aligned}$$

Therefore:

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

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 $[0, m-1]$:

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

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:

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

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.