21:59 Sun Feb 21 99%

cryptography eng

Home Insert Draw View

CW

Friday, February 19, 2021 00:33

Ziyang Yang

Yiming Zhang

1. a. Compute $p(C=2)$:

$$p(C=2) = \sum_{k \mid c \in e_k(K)} p(K=k) \cdot p(P=d_k(2))$$

$$= p(K=k_1) \cdot p(P=b) + p(K=k_2) \cdot p(P=d) + p(K=k_3) \cdot p(P=c) + p(K=k_4) \cdot p(P=a) + p(K=k_5) \cdot p(P=a)$$

$$= 0.2 \cdot 0.19 + 0.2 \cdot 0.21 + 0.2 \cdot 0.22 + 0.2 \cdot 0.18 + 0.2 \cdot 0.18.$$

$$= 0.2 \cdot (0.19 + 0.21 + 0.22 + 0.18 + 0.18)$$

$$\boxed{= 0.196.}$$

b. Compute $p(C=4 \mid P=e)$

$$p(C=4 \mid P=e) = \sum_{k \mid m = d_k(c)} P(K=k) = P(K=k_3) \boxed{= 0.2.}$$

c. Compute $p(P=c \mid c=3)$

$$p(C=3) = p(K=k_1) \cdot p(P=d) + p(K=k_2) \cdot p(P=e) + p(K=k_3) \cdot p(P=b) + p(K=k_4) \cdot p(P=e) + p(K=k_5) \cdot p(P=c)$$

$$= 0.2 \cdot (0.21 + 0.2 + 0.19 + 0.2 + 0.22)$$

$$= 0.2 \cdot 1.02$$

$$= 0.204.$$

$$p(P=c \mid c=3) = \frac{p(P=c) \cdot p(C=3 \mid P=c)}{p(c=3)} = \frac{0.22 \cdot 0.2}{0.204} \boxed{= 0.216}$$

d. For Perfect Secure: $p(P=m \mid c=c) = p(P=m)$.

We already Compute that $p(P=c \mid c=3) = 0.216$ and $p(C=3) = 0.204$, which is not fit the Perfect Secure Requirement.

Also, For a Double Check,

In this case, $|P| = |C| = |K|$, we can apply S1 & S2 to Jusify the answer.

- S_1 is True, Due to $P(K=k_i)$ equals to each other.
- S_2 is False. In the Table, when $P=e$, $c=3$, there are 2 possible keys to generate that. which means it is not unique key in K with $e_k(m)=c$.

Thus, This crypto-system is $\boxed{\text{NOT}}$ perfectly Secure.

2: ECDSA Generation:

map points P on elliptic curve to integer interval $[0, q-1]$.

$f(P)$ = "x-coordinate of point P" mod q

1: $h = \text{RIPEMD-160}[\text{SHA-256}(m)]$

2: choose random u, with $0 < u < q$.

3: $r = f(u * G)$ (repeat step 2 if $r=0$)

4: $s = (h + k \cdot r) \cdot u^{-1} \bmod q$. (repeat step 2 if $s=0$)

ECDSA Verification:

1: $h = \text{RIPEMD-160}[\text{SHA-256}(m)]$

2: $a = h \cdot s^{-1} \bmod q$

3: $b = r \cdot s^{-1} \bmod q$

4: $v = f(a * G + b * K)$

5: accept if $v = r$.

For Question (a):

Under the assumption that $m' = m$,

$$v = f(a * G + b * K)$$
$$= f[h \cdot s^{-1} * G + r \cdot s^{-1} * (k * G)]$$
$$= f[s^{-1}(h + k \cdot r) * G]$$

From the Generation step 4: and 3

$$v = f[u * G] = r$$

The verification is successful!

2. For Question B:

Because the m_1 and m_2 are provided, the attacker could compute the hash value of m_1, m_2. $\boxed{< \text{assume } h_1 \neq h_2 >}$

$\boxed{h_1 = h(m_1),\quad h_2 = h(m_2).}$

The attacker also has the signatures of m_1, m_2, and the

$\boxed{(r_1, s_1) \text{ for } m_1,\quad (r_2, s_2) \text{ for } m_2}$

The attacker could also get public parameters of the Elliptic Curve

From the ECDSA Generation Step ③, $r = f(u * G)$

Using the same u, So $r_1 = r_2$, attacker could get:

$$\boxed{\begin{cases} s_1 = (h_1 + k \cdot r) \cdot u^{-1} \bmod q \\ s_2 = (h_2 + k \cdot r) \cdot u^{-1} \bmod q \end{cases}} \Rightarrow \boxed{u = \frac{h_1 - h_2}{s_1 - s_2} \bmod q.}$$

At once attacker get u, the k could be calculated:

$$s_1 = u^{-1}(h_1 + k \cdot r) \bmod q$$
$$\Rightarrow k = r^{-1}(s_1 u - h_1) \bmod q.$$

For the possibility of collision:

① Because m_1 and m_2 are two different message, and the hash value is calculated by two hash: $h = \text{RIPEMD-16}[\text{SHA-256}(m)]$.

② The second hash is shorter than the first, the result is distributed closer to random.

③ And there is a SHA-256 between RIPEMD-16 and ECDSA, which makes it very impossible to find address collision.

④: So it is very impossible for the attacker to find the collision

3. question a:

under the assumption: y_0 with $0 < y_0 < p$, statisfies $y^2 = n \bmod p$

So: $(p-y_0)^2 \bmod p = (\boxed{p^2 - 2py_0} + y_0^2) \bmod p$

$= y_0^2 \bmod p$

$= n \bmod p$

$(p-y_0)^2 = n \bmod p$

Thus. $(p-y_0)$ is the second solution of this equation.

question b:

①: because p is a prime, p must be an odd number or 2.

if $p=2$, there are only two points on the curve, $(1,0)$ and $(0,1)$. It is meaningless for Bitcoin security. So $p > 2$.

because $0 < y_0 < p$,

we get: $\boxed{-p < -y_0 < 0 \Rightarrow 0 < p - y_0 < p}$

② Assume $y_0 = p - y_0$, we get $y_0 = p/2$, (y_0 is an integer)

But we have known that the p is prime and $p > 2$.

So the assumption $y_0 = p - y_0$ is false, which means $y_0 \neq p - y_0$.

Question c):

we have known that: $\begin{cases} y_0 \neq p - y_0 \\ 0 < y_0 < p, \\ 0 < p - y_0 < p. \end{cases} \Rightarrow \begin{cases} y_0^2 = n \mod p \\ (p-y_0)^2 = n \mod p \end{cases}$

So, there are two different solutions, and the two solutions' value are both in $(0, p)$, the two solutions are $(y_0, p-y_0)$

When we have known one solution Y, but we don't know if the Y is the smallest solution, At this time we could calculate the $(p-Y)$.

Then, let $z = \text{minimum}(Y, p-Y)$, which means z equals with the smallest value in Y and $p-Y$

In this way, we could get the unique smallest solution value z.

Let $y_0 = z$, y_0 is the smallest solution.

Question d): Generally, the point (x_0, y_0) is the public key,

And the point (x_0, y_0) statisfies: $\boxed{y_0^2 = x_0^3 + 7 \mod p}$

In more details: this key format is: $\underset{\text{prefix}}{04}\ \underbrace{x_0}_{32\text{ bits}}\ \underbrace{y_0}_{32\text{ bits}}$, which is known as uncompressed format.

From what I think, we can only store x_0, to achieve the compressed format.

①: Because y_0, $(p-y_0)$ are two different solution of $y_0^2 = x_0^3 + 7 \mod p$

we could get the minimum solution between y_0, $(p-y_0)$, let $z_0 = \min(y_0, p-y_0)$

the prefix $= \begin{cases} 03 \text{ when } z_0 \text{ is odd} \\ 02 \text{ when } z_0 \text{ is even} \end{cases}$

In the end, we get $<\underline{\text{prefix}}\ \underbrace{x_0}_{32\text{ bits}}>$ is the compressed public key in only half size

②: For decompressing, the x_0 could be extracted from the compressed public key.

With the x_0, we could get two different solutions $(z_0, p-z_0)$, from $z_0^2 = x_0^3 + 7 \mod p$

Let $y_0 = \min(z_0, p-z_0)$,

or, according to the prefix, $\begin{cases} \text{if prefix} = 03, & y_0 \text{ is the odd one from } (z_0, p-z_0) \\ \text{if prefix} = 02, & y_0 \text{ is the even one from } (z_0, p-z_0) \end{cases}$

[Because p is prime and $p>2$,
p = an odd number + an even number]

In this way, we could decompress to get the original public key (x_0, y_0).

Question - 4:

a. With input k**, *generateRSAPrime(int k)*** function returns big prime number *x* which is random in the range of [*2**(k - 1) , 2**k - 1*]. Also, the value of *((x - 1) mod e)* cannot be *0*.

The function has at most *100*k* times to try to find the ***x*** that is meet the previous conditions. The reason of *(x % 5 != 1)* is because return of the function is for creating p and q for the RSA algorithm. We want to make sure *gcd(e, (p - 1)(q - 1))* not equals to e itself. Which means *((x - 1) mod e) cannot be 0* → *(x mod e)* cannot be 1.

There are 2 asserts in the function.

1. The first one checks k's value is in the range of [*1024, 4096*]. This make sure the k we use is big enough to against the brute-force attacks and not too big.
2. The second one makes sure the finding x loop loops at most *100*k* times. So that if suitable *x* is not been found in limited time, the function will stop at a certain time.

b. ***generateRSAKey(int k)*** returns two prime number *p*, *q*, their product *p*q* (in another representation is N), and the private key *d* it created in the function.

There are 3 asserts in this function:

1. Make sure k is in the range we want, big enough for create brute-force attack free p and q. Also value of *p*q* is in the range of [*2**2048, 2**8192*].
2. Make sure *p* is not equal to *q*. If *p* is equal to *q*, they will not be coprimes which will be easy for attackers to compute the private key.
3. Make sure the greatest common divisor of *5* and *(p-1)*(q-1)* is 1, which means *e* and *t* are coprimes.

c. Because k is big enough, k is secure for generating r. Compute N from p and q (we calculate in ***generateRSAKey***) is easy, but by only known *N* it is extremely had to calculate its 2 prime factors.

d. In the ***encryptRandomKeyWithRSA(N)***, *k = floorint[log2(N)]*, which means:

$$\begin{aligned}
&\log_2 N - 1 < k \leqslant \log_2 N \\
&\tfrac{N}{2} < 2^k \leqslant N \\
&\tfrac{N}{2} - 1 < 2^k - 1 \leqslant N - 1
\end{aligned}$$

The huge range of r could be shown as:

$$\begin{aligned}
&0 \leqslant r \leqslant 2^k - 1 \\
&r \leqslant N - 1
\end{aligned}$$

Thus, k is secure enough to generate a random element r in the plaintext space.

K is derived as the double hash of r, which is selected randomly from the range [*0,N*]. Because hash function is collision free, it can avoid the brute-force attack. Instead of avoiding attackers get *r* directly, the double hash can also avoid length extension happens.

e. The assert in the function is to make sure *c* is in the correct range [*0,N).* This assert is checking if the c is what we want in RSA space.

From the notes p.53, there are two important fact:

$$m^{e \cdot d} = m \bmod N$$
$$e \cdot d \equiv 1 \bmod (p-1)(q-1)$$

And then:

$$\begin{aligned} i &= c^d \bmod N \\ &= (r^e)^d \bmod N \\ &= r^{e \cdot d} \bmod N \\ &= r \cdot \left(r^{(p-1)(q-1)}\right)^s \bmod N \\ &= r \bmod N \end{aligned}$$

The function utilizes the double SHA-256 hash to the result of *(i = r mod N).* In this way, with the correct *N, d, c,* K could be recovered successfully.

f. By using Alice and Bob as share session key example:

1. With a k *(2048 ⩽ k ⩽ 8192),* Alice uses ***generateRSAKey(int k)*** to generate *p, q, N (N = p * q), d*. In this case, the ***e*** is 5. To create big prime numbers *p* and *q*, Alice needs to call ***generateRSAPrime(k/2)*** inside the function. Finally she will send (*5, N*) to Bob.

   During this sending process, the attacker can only obtain the (*5, N*).

2. Bob receives (*5, N*) and calls ***encryptRandomKeyWithRSA(N)*** by using *N* as an input. The function will create the session key they want, *K*, and key generator *c***.** Bob sends *c* to Alice.

   During this sending process, the attacker can get the c.

3. Then Alice receives c and uses ***decryptRandomKeyWithRSA(N,d,c)***. She can calculate *K* by using values she already knows (*N, d, c*).

4. Session key share securely finishes.

The attacker will only have chance to get *5, N* and encrypted *c*, which is impossible to decrypt *K* by using these information.