Modern Cryptology - Homework 2

Yosef Goren

May 24, 2022

Question 1

1.1

G' is PRG.
Proof:
Let G be a PRG (with strech $l(n)$). Denote $S_n = \{1 \dots n\}, R = \{1 \dots l(n)\}$. Let σ be a permutation over R.

Assume to contradict that $G : S \longrightarrow R' =< G, \sigma >$ is not PRG.
From the definition of PRG, there must be some distinguisher D' with advantage over outputs of G' which are more than $neg(n)$.
Namely:

$$(*); \exists D : |\Pr_{s \leftarrow S_n}[D'(G'(s)) = 1] - \Pr_{r \leftarrow R_n}[D'(r) = 1]| > neg(n)$$

We will now use D' to construct a distinguisher D:

$$(**); D(r) = D'(\sigma(r)) \Rightarrow D'(r) = D(\sigma^{-1}(r))$$

Note that when we apply a permutation over R to a string of characters, the meaning is that we produce a new string by reordering the original string according to the indices generated by the permutation.

Given $(*)$ and $(**)$, we can see:

$$|\Pr_{s \leftarrow S_n}[D'(G'(s)) = 1] - \Pr_{r \leftarrow R_n}[D'(r) = 1]| > neg(n)$$

$$\Rightarrow |\Pr_{s \leftarrow S_n}[D(\sigma^{-1}(G'(s))) = 1] - \Pr_{r \leftarrow R_n}[D(\sigma(r)) = 1]| > neg(n)$$

$$\Rightarrow |\Pr_{s \leftarrow S_n}[D(\sigma(\sigma^{-1}(G(s)))) = 1] - \Pr_{r \leftarrow R_n}[D(r) = 1]| > neg(n)$$

$$\Rightarrow |\Pr_{s \leftarrow S_n}[D(G(s)) = 1] - \Pr_{r \leftarrow R_n}[D(r) = 1]| > neg(n)$$

This is a direct contediction to G being PRG.
So there is no distinguisher for G', and the strech condition applies ofcourse since the permutation does not change the length of output; meaning G' is PRG.

1.2

G' is not PRG.
Proof:
Take for example the distinguisher D': $D'(s) = \{s_L = s_R : 1, 0\}$, meaning that the distinguisher checks if the left half of the input is the same as the right part and if so - returns 1.
Now we can see:

$$|\Pr_{s\leftarrow S_n}[D'(G'(s)) = 1] - \Pr_{r\leftarrow R_n}[D'(r) = 1]|$$

$$= |\Pr_{s\leftarrow S_n}[D'(G(s)G(s)) = 1] - \frac{1}{2^{l(n)}}|$$

$$= |1 - \frac{1}{2^{l(n)}}| \approx 1 > neg(n)$$

Indeed this means G' cannot be PRG.

1.3

G' is PRG.
Proof:
Assume a distinguisher D' for G'.
Define D:

$$D(s) = D'(s; r)$$

where r is sampled from R.
Since G is PRG, there must be no (distinguishable) statistical difference between $r \leftarrow R$ and $G(s) : s \leftarrow S$. This means that for a computationaly limited algorithm such as D', there cannot be a statistical difference between $D'(G(s); G(s')) : s, s' \leftarrow S$ and $D'(G(s); r) : s \leftarrow S, r \leftarrow R$.
Hence:

$$neg(n) < |\Pr_{s\leftarrow S_{2n}}[D'(G'(s)) = 1] - \Pr_{r\leftarrow R_{2n}}[D'(r) = 1]|$$

$$= |\Pr_{s,s'\leftarrow S_n}[D'(G(s)G(s')) = 1] - \Pr_{r\leftarrow R_n}[D'(r) = 1]|$$

$$= |\Pr_{s\leftarrow S_n, r\leftarrow R_n}[D'(G(s); r) = 1] - \Pr_{r\leftarrow R_n}[D'(r) = 1]|$$

$$= |\Pr_{s\leftarrow S_n}[D(G(s)) = 1] - \Pr_{r\leftarrow R_n}[D(r) = 1]|$$

But this contredicts G being PGR. So G' must be PRG too (again the strech condition is trivialy shown).

Question 2

The algorithm:

```
1  def Log(a):
2    n = len(a)
3    for _ in range(Poly(n)**2):
4      j = random(1, p(n))
5      b = a*(g(n)**(-j))
6      z = A(b)
7      if b == g(n)**z:
8        return (z+j)%p(n)
```

Claim (Correctness):

$$\Pr_{x\leftarrow\mathbb{Z}^*_{p_n}}[Log(g_n^x)=x]=o(1)$$

Proof:
Let $n\in\mathbb{N}$.
Denote the set of randomizations (at line 4) with: $J=\{j_i\}_{i=1}^{Poly(n)}$
Also denote the space of all possible sets of randomizations: $R=(\mathbb{Z}^*_{p_n})^{Poly(n)^2}$.

$$\Pr_{x\leftarrow\mathbb{Z}^*_{p_n},J\leftarrow R}[Log(g_n^x)=x]=1-\Pr_{x\leftarrow\mathbb{Z}^*_{p_n},J\leftarrow R}[Log(g_n^x)\neq x]$$

$$\overset{(*)}{=}1-\Pr_{x\leftarrow\mathbb{Z}^*_{p_n},J\leftarrow R}[\bigwedge_{i=1}^{Poly(n)^2}A(g_n^{x-j_i})\neq x]$$

$$=1-\prod_{i=1}^{Poly(n)^2}\Pr_{x\leftarrow\mathbb{Z}^*_{p_n},j\leftarrow\mathbb{Z}^*_{p_n}}[A(g_n^{x-j})\neq x-j]$$

$$=1-\Pr_{x\leftarrow\mathbb{Z}^*_{p_n},j\leftarrow\mathbb{Z}^*_{p_n}}[A(g_n^{x-j})\neq x-j]^{Poly(n)^2}$$

$$=1-\Pr_{z\leftarrow\mathbb{Z}^*_{p_n}}[A(g_n^z)\neq z]^{Poly(n)^2}=1-(1-\Pr_{z\leftarrow\mathbb{Z}^*_{p_n}}[A(g_n^z)=z])^{Poly(n)^2}$$

$$=1-(1-\frac{1}{Poly(n)})^{Poly(n)^2}\overset{(**)}{\underset{n\to\infty}{\longrightarrow}}1$$

Showing $(*)$:
In each iteration of our algorithm, we sample some $j\leftarrow\mathbb{Z}^*_{p_n}$. The iterations where our algorithm will return (and we will show why in the cases it does return - the returned value is correct) are the cases where: $g^{A(a\cdot g^{-j})}=a$; In order for the algorithm to fail, each iteration must fail, or in other words:

$$\bigwedge_{i=1}^{Poly(n)^2}A(g_n^{x-j_i})\neq x$$

The values returned by the algorithm are indeed correct, because it only returns $z+j$ in cases where $b=g_n^z$, and we can see that:

$$b=g_n^z\Rightarrow a\cdot g_n^{-j}=g_n^z\Rightarrow g_n^{x-j}=g_n^z$$

$$\Rightarrow x - j \underset{mod(p_n)}{=} z \Rightarrow x \underset{mod(p_n)}{=} z + j$$

Showing $(**)$:

$$Poly(n) \underset{n\to\infty}{\longrightarrow} \infty \Rightarrow (1 - \frac{1}{Poly(n)})^{Poly(n)} \underset{n\to\infty}{\longrightarrow} \frac{1}{e}$$

$$\Rightarrow ((1 - \frac{1}{Poly(n)})^{Poly(n)})^{Poly(n)} \underset{n\to\infty}{\longrightarrow} 0 \Rightarrow (1 - \frac{1}{Poly(n)^2})^{Poly(n)} \underset{n\to\infty}{\longrightarrow} 0$$

$$\Rightarrow 1 - (1 - \frac{1}{Poly(n)^2})^{Poly(n)} \underset{n\to\infty}{\longrightarrow} 1$$

Note that if $Poly(n)$ can be a constant and thus will not approach infinity, we can deal with such cases by simply defining $Poly'(n) = \max(Poly(n), n)$, and using $Poly'(n)$ in our algorith.

□

Question 3

3.1

The problem with the scheme proposed is that Bob can just fake his result to get whatever he wants: Say that Alice and Bob agreed that a result of zero means Alice gets the dog and a result of one means Bob gets the dog, and we also assume that Bob wants the dog. Bob can get the bit a from Alice, and send her $b = a \oplus 1$.
Now both Bob and Alice will see:

$$result = b \oplus a = a \oplus 1 \oplus a = 1$$

Which means Bob always gets the dog.

3.2

Alice and Bob can use a coin flipping scheme by deciding that a result of zero means that Alice gets the dog, and otherwise - Bob does.

We could build a coint flipping scheme with the following two algorithms for Alice (denoted with A) and Bob (denoted with B):

```
A(n):
b = random
r = random
send commit(b, 1^n;r)
recv c'
```

```
send b,r
recv b',r'
if commit(b',1^n;r') != c', return 1
return xor(b,b')

B(n):
b' = random
r' = random
recv c
send commit(b', 1^n;r')
recv b,r
send b',r'
if commit(b,1^n;r) != c, return 0
return xor(b,b')
```

Note:
Here *send* and *recv* are "blocking" or "synchronous" - meaning the algorithm will not advance to the next statment before verifying that communication has completed successfully.

Correctness:
If both follow the protocol, both b and b' are sampled randomly, and also, both parties do not go into the "if" statment and thus return $b \oplus b'$; with these two uniformly sampled, their xor will also be unfiormly sampled, which means both parties return some value randomly sampled from $\{0, 1\}$.

Alice Soundness:
If alice follows the protocol, she will only send her bit after she sees a commit from her opponent. So her opponent cannot know the value of b before sending her some commit c. This means the opponent cannnot wait until he knows b to decide the value of b' (knowing b would mean breaking the "binding" property of the commitment scheme).
If so, there cannot be any dependence between the variables b and b' - meaning that regardless of the b' distribution: $b \oplus b'$ is distributed evenly between zero and one.
So we can conclude that in cases where the opponent decides to send a legal commit message, Alice will return zero with probability $\frac{1}{2}$.

Furthermore, in cases where the opponent does not send a legal commit;message pair - Alice finds this is the case and returns zero.

To sum up, there are two cases; if a legal commit is sent: $\Pr[A(n) = 1] = \frac{1}{2}$.
Otherwise: $\Pr[A(n) = 1] = 0$.
Regardless of the distribution between these two cases, the total probability for $A(n) = 1$, is no more than $\frac{1}{2}$.

Bob Correctness:
Mirror proof. Indeed Bob also requires knowing Alice's commit message before sending b' in the same way Alice waits for Bob's commit before sending b - and in addition, Bob also checks the correctness of the commit message sent to him in the same way.

3.3

Indeed both can exploit an opponent following his/her part of the protocol to cause the other to sample their own token - by simply sending an incorrect commit message. For example, if Bob sends an incorrect commit message, Alice will check that at the end - and thus return 0. If Bob is also sampling 0 himself, both sides have "agreed" that the dog should go to Alice.

Question 4

4.1

Trivial aglorithm:

```
def ForgeTag(reference_msg, reference_tag, new_msg):
result = ""
for i in range(len(reference_tag)):
  result += random([0, 1])
return result
```

Correctness:
Denote correct tag for *new_msg* as *new_tag*. Since the length of *new_tag* is t and the algorithm uniformly samples from all binary strings of length t; there is a $(\frac{1}{2})^t$ chance that the string randomized by the algorithm will be *new_tag*. Meaning this is the probability of the algorithm being correct.

4.2

Pairwise Independence:
Let $x_1, x_2 \in \mathbb{F} : x_1 \neq x_2$. Let $y_1, y_2 \in \mathbb{F}$.

$$= \Pr_{h \leftarrow H}[h(x_1) = y_1 \wedge h(x_2) = y_2] = \Pr_{a,b \leftarrow \mathbb{F}}[ax_1 + y_1 = b \wedge ax_2 + y_2 = b]$$

$$= \sum_{B \in \mathbb{F}} \Pr_{a,b \leftarrow \mathbb{F}}[ax_1 + y_1 = b \wedge ax_2 + y_2 = b \mid b = B] \Pr_{b \leftarrow \mathbb{F}}[b = B]$$

$$= \sum_{B \in \mathbb{F}} \Pr_{a,b \leftarrow \mathbb{F}}[ax_1 + y_1 = B \wedge ax_2 + y_2 = B] \frac{1}{|\mathbb{F}|}$$

$$= \frac{1}{|\mathbb{F}|} \sum_{B \in \mathbb{F}} \Pr_{a,b \leftarrow \mathbb{F}}[ax_1 + y_1 = ax_2 + y_2 = B]$$

$$= \frac{1}{|\mathbb{F}|} \Pr_{a,b \leftarrow \mathbb{F}}[ax_1 + y_1 = ax_2 + y_2] = \frac{1}{|\mathbb{F}|} \Pr_{a,b \leftarrow \mathbb{F}}[a(x_1 - x_2) = y_2 - y_1]$$
$$\overset{(*)}{=} \frac{1}{|\mathbb{F}|} \Pr_{a,b \leftarrow \mathbb{F}}[a = (y_2 - y_1)((x_1 - x_2))^{-1}] = \frac{1}{|\mathbb{F}|^2} = \frac{1}{2^{2m}} = 2^{-2m}$$
$$\square$$

$$(*) : x_1 \neq x_2 \Rightarrow x_1 - x_2 \neq 0 \Rightarrow (x_1 - x_2)^{-1} \in \mathbb{F}$$

The general case for the range field:
To make use of this same hash family (and proof of independence), in a setting as to transfer items from $\mathbb{F}_1$ to $\mathbb{F}_2$; define:

$$H = \{h_{a,b} \mid a'b \in \mathbb{F}_2\}; h_{a,b}(x) = a \cdot T(x) + b$$

Where $T(\cdot)$ is some transformation $T : \mathbb{F}_1 \longrightarrow \mathbb{F}_2$, such that $T(0_{\mathbb{F}_1}) = 0_{\mathbb{F}_2}$ and $T(x + 1_{\mathbb{F}_1}) = T(x) + T(1_{\mathbb{F}_1})$. Under these assumtions, the provided proof from before is correct in the general case.
Also note how in the prior proof, the determining factor for the final probability was the size of the field from which a, b are sampled, i.e. $\mathbb{F}_2$ in the general case. Meaning the probability is indeed 2^{-2m}, where $m = |\mathbb{F}_2|$.

4.3

Define the following MAC algorithm:

```
def h(a, b, x):
  """This is the hash function parameterized by 'a,b' from the H
    family of pairwise independent functions (from section 4.2).
    The output field size is the same as that of 'x' and 'y'."""
  ...

def MakeTag(key, msg):
  mid = len(key)//2
  kL = key[:mid]
  kR = key[mid:]
  return h(kL, kR, msg)
```

Correctness:
Given that the key is selected randomly, (and the opponent does not have access to it), Both sides of the key are also selected randomly - meaning that the algorithm has simply sampled as random hash function from H. As stated in section 4.2; this means that the behaviour of this randomly selected hash from H must be the same as that of a completely random function over the same space. If indeed our tagging scheme behaves in the same way as a random function over the same sapce, it means that any PPT will not be able to deduce anything from just one output of our tagging scheme (with a spesific key). Thus any opponent algorithm must not do any better than either guessing an output; which would yeild that algorithm a chance of success equal to $\frac{1}{N}$ where N is the size of the output domain. In our case, $N = 2^t$, meaning no opponent can do better than 2^{-t}.