Homework 2

1 Problem 1 (5 points)

Classify the following functions as polynomial, inverse polynomial, super-polynomial, or negligible. You do not need to rigorously prove your answer, but you should give a convincing justification.

(a) $f_a(\lambda) = \lambda^{(10^{10})}$

(b) $f_b(\lambda) = 2^{\log(1/\lambda)\log(\lambda)}$

(c) $f_c(\lambda) = \lambda^{-(\lambda^{1/\lambda})}$

(d) $f_d(\lambda) = \log(\lambda!)$

(e) $f_e(\lambda) = \lambda^{\log^*(\lambda)}$, where $\log^*(\lambda)$ represents the *iterated logarithm* https://en.wikipedia.org/wiki/Iterated_logarithm

2 Problem 2 (15 points)

Suppose $(\mathsf{Enc}, \mathsf{Dec})$ is an encryption scheme that has ciphertext indistinguishability, with key space $\mathcal{K}_\lambda$, message space $\mathcal{M}_\lambda$, and ciphertext space $\mathcal{C}_\lambda$. Assume $\mathcal{K}_\lambda = \mathcal{M}_\lambda = \{0,1\}^\lambda$.

Which of the following encrpytion algorithms are guaranteed to represent correct encryption schemes with ciphertext indistinguishability? There are several cases:

- The scheme is guaranteed to have ciphertext indistinguishability, no matter what $(\mathsf{Enc}, \mathsf{Dec})$ does, as long as $(\mathsf{Enc}, \mathsf{Dec})$ has ciphertext indistinguishability. In this case, prove the ciphertext indistinguishability of the derived scheme.
- The scheme is always insecure, not matter what $(\mathsf{Enc}, \mathsf{Dec})$ does. In this case, show an attack that works no matter what.
- The scheme may be secure for some choices of $(\mathsf{Enc}, \mathsf{Dec})$, but insecure for others. In this case, both (1) give an example of a secure $(\mathsf{Enc}, \mathsf{Dec})$ such that the derived scheme is insecure (and provide an attack), and also (2) provide an

example of a secure $(\mathsf{Enc}, \mathsf{Dec})$ that yields a secure scheme, and prove security. Contrived examples are allowed, and if needed you may use a secure PRG as a building block for the examples.

- For some schemes, it may be the case that it is not a correct encryption scheme. In this case explain why. But it also makes sense to talk about the security of the scheme even if the scheme isn't correct. Therefore, in this case, also decide which of the three cases above applies.

You do not need to explain how to decrypt.

(a) $\mathsf{Enc}_a(k, m) = \mathsf{Enc}\,(k, (m, r))$. Here, the message space for Enc_c is $\{0,1\}^{\lambda/2}$, r is a random $\lambda/2$-bit string, and (m, r) is the concatenation of m and r.

(b) $\mathsf{Enc}_b(k, m) = \mathsf{Enc}(k, m) \oplus \mathsf{Enc}(k, 0^n)$.

(c) $\mathsf{Enc}_c(k, m) = \big(\mathsf{Enc}(k, m), \mathsf{Enc}(k, m \oplus 1^\lambda)\big)$. That is encrypt under Enc, and then encrypt the bitwise complement under m.

3 Problem 3 (10 points)

Let PRG be a pseudorandom generator. Consider the following attempt at building a stateless many-use encryption scheme. $\mathsf{Enc}(k, m)$ chooses a random string IV of length λ (here, λ is the length of the key), and then runs $x \leftarrow \mathsf{PRG}(IV, k)$ (that is, run PRG on the string obtained by concatenating IV and k). Finally, it computes $c \leftarrow x \oplus m$. The ciphertext is the pair (IV, c). $\mathsf{Dec}(k, (IV, c))$ uses IV and k to compute x, and XORs x and c to recover m.

Devise an example of a PRG PRG such that the above encryption scheme using PRG is insecure even for a single message. That is, PRG should satisfy the definition of a secure PRG, but Enc should not satisfy ciphertext indistinguishability for any small ϵ.

You may assume as a building block a secure pseudorandom generator PRG', which you can use to build your PRG. Your construction must work (that is, yield an insecure encryption scheme) for any PRG', as long as PRG' is a secure PRG; do not assume any particular structure on PRG'. Remember to prove the security of PRG assuming the security of PRG' using a reduction.

4 Problem 4 (20 points)

Give efficient attacks on each of the following candidate PRG constructions.

You should not need a computer to solve any of these problems. You can use a computer if it helps, but you should be able to demonstrate your attack works without resorting to any code. So for example, if the second byte of output is biased towards 0, you should be able to explain why this is the case by using the description of the function (so something like "If event A happens, then the second byte is 0. For a random initial state, event A happens with probability 2/256"). Empirical evidence is not sufficient (so "I ran 1,000,000 tests and this is what I found" is not an acceptable answer).

Since the following constructions are concrete functions, we cannot use asymptotics to say whether an attack works or not. Instead, your should run in a "reasonable" time (say, at most seconds) and should have a "noticeable" advantage (say, at least 2^{-20}). If you have found the correct attack, it should be very clear that it satisfies these constraints.

(a) PRG_a is a LFSR on 256-bit inputs. The feedback is the XOR of the 2nd, 27th, 168th, and 220th bits of the state. In order to introduce non-linearity, the output is the AND of the last two bits of state.

(b) PRG_b is a LFSR on 256-bit inputs, except that only steps that are multiples of 100 give an output. For all the steps in between, the last bit of state is just discarded without being outputted. The idea is that the initial state will no longer be the first 256 bits of output.

(c) PRG_c is defined as follows.

- The state is a permutation on 256 elements and two counters i, j, just like RC4. The permutation is stored as an array S.
- The input is a 128-bit seed, just like RC4, and the state is initialized based on the seed as in RC4
- To produce the next output and update the state, do the following:
  - Let $i = i + 1 \bmod 256$. Let $j = j + S[i] \bmod 256$
  - Rotate the array S to the right by 17 positions. That is, let $S[\ell] = S[\ell - 17 \bmod 256]$ for all ℓ.
  - The new state is the new values for i, j, and S. The output is

$$S[S[i] + S[j] \bmod 256]$$

Note that for this problem, since this function is different than RC4, you cannot assume anything based on known weaknesses of RC4. Instead, any anomalies you find you must prove yourself.

(d) PRG_d is defined as follows.

- The state is a permutation on 256 elements. The permutation is stored as an array S.
- The input is a 128-bit seed, just like RC4, and the state is initialized based on the seed as in RC4 (notice that initializing RC4 also outputs two counters that are set to 0, we will discard these)
- To produce the next output and update the state, do the following:
  * The next output byte is $S[0]$
  * To update the state, do the following. Compute

$$i = S[S[0] + S[S[0]] \bmod 256]$$

    If $i = 0$, shift S to the left by 1 (so $S[0]$ becomes $S[1]$, etc). Otherwise, swap the contents of $S[0]$ and $S[i]$.

Again, since this function is different than RC4, you cannot assume anything based on known weaknesses of RC4. Instead, any anomalies you find you must prove yourself.