Solution Set #3

CMSC/Math 456
Instructor: Daniel Gottesman

Due on Gradscope, Thursday, Sep. 29, noon (before start of class)

Overview: The goal of this problem is to break a poorly designed stream cipher. You will write Python scripts to attack the stream cipher in three different contexts.

For this solution set, I will not give code for attacking, but discuss a few different approaches that can be made to work, although in some cases not with full reliability or at all lengths.

The stream cipher: The stream cipher is somewhat similar to RC4. The state passed between calls of the stream cipher consists of list of 256 bytes (i.e., 256 numbers, each between 0 and 255), plus one additional number between 0 and 255, which can be viewed as a pointer into the list. (This is a difference from RC4, which has 2 pointers.). Another difference is that this stream cipher takes an initial value, which must be in the form of a list of 256 bytes.

The Init and Next functions are substantially simpler than RC4. Init takes initial value IV (a list of 256 bytes) and key k (a list of some number of bytes) as input and does the following:

1. state = IV
2. for j running up to the length of k, let k_j be the jth byte of k, and update state:
    - new state[i] = old state[$i + k_j \bmod 256$]
3. $i = 0$

Init outputs (i, state).

The Next algorithm takes internal state (i, state) as input and does the following:

1. $x = \text{state}[i]$
2. $i = i + \text{state}[(i+1) \bmod 256]$

It outputs the byte x as the output of the stream and the updated internal state (i, state).

The Enc function provided to you takes as input IV (which is supposed to be chosen randomly at the time of encryption), a key k, and a message m (which is given as a list of bytes) and outputs ciphertext c, which begins with the IV and then follows with $m_j + x_j \bmod 256$ for all j, where m_j is the jth message byte and x_j is the jth byte output from the stream cipher with the IV and k. Dec just inverts this given key k and ciphertext c.

The functions described above allow the key for the stream cipher can be any length, but you may assume that the keys are 16 bytes long for these problems.

Problem #1. Attack via the Pseudorandom Generator (20 points)

Define a function "PRG_attack(IV, x)" which takes as input IV (an initial value which is a list of 256 bytes) and x (a list of bytes of variable length). Your function should return 0 if x is an output of the stream cipher with that IV and some seed and 1 if x was generated by a random process.

Note: Do not return the key value, only if x was random or pseudorandom. If you have an attack which determines the key, that is fine; but it is also acceptable to have an attack which gives the right answer without learning the key.

Your attack will be tested on a number of instances with x being 10 bytes long, 100 bytes long, and 1000 bytes long. You can get partial credit if your attack works for some of these lengths but not others.

Solution: There were a number of ways to attack the pseudorandom generator. The best ones take advantage of the fact that each key byte rotates the IV string by k_j positions, meaning that no matter the length of the key, the IV string is rotated overall by $k = \sum_j k_j$ positions. Thus, all you need to know to fully break this system is k.

Here are some possible approaches:

1. The most straightforward approach is to search over the 256 possible values of k to see what the output stream is in each case using your knowledge of the IV. If one of those output streams matches the string x, return 0. Otherwise return 1. This approach is somewhat slow, but can be sped up significantly by only checking for matches on enough bytes of x to be confident that it is not an accident. For instance, if you check 10 bytes of x, the probability of a random string matching a single pseudorandom candidate (for a particular guess of k) is 256^{-10}; since you have to check 256 candidates, the total probability of an accidental match is 256^{-9}, which is still extremely small.

2. A somewhat cleverer and faster approach (albeit slightly more difficult to code) is to notice that the first output byte of the stream is state[0], which was originally IV[256-k]. Therefore, if you can find the first output byte in the IV, that gives you a candidate for k. The thing to be careful of is that that same value might appear more than once in the IV, in which case you will have multiple candidates for k and you need to check which one is correct as above. If none of them match or the first output byte is not in the IV, then return 1.

3. If you notice that the state list after Init is a permutation over IV values but not that it is always a shift, you can realize that the output stream can only ever contain values that appear in the IV. The odds of the IV repeating something are quite high (we are way over the birthday paradox size), so there will also (with very high probability) be some values that don't appear in the IV at all. If they appear in x, then x must be random; return 1. Otherwise, return 0.

   This approach will work with very high probability when x is long. When x is only 10 bytes, the odds are still quite good that it will work (about a 1% chance of failure per trial).

4. Another approach is to notice that the pointer i jumps around but state doesn't change under Next. This means that eventually i will repeat its position, at which point the stream will start to repeat itself. This always happens after at most 256 steps, but usually will happen earlier. Therefore, to distinguish random x from pseudorandom x, look for a point where the output stream begins to repeat itself. (Note: it might start repeating from some point in the middle, not repeating from the beginning.) If it repeats, return 0, and if not, return 1.

   This approach is certain to work (up to the extremely small chance of a random string that happens to repeat) for x with length 1000, very likely to work for x with length 100, and very *unlikely* to work for x with length 10.

Problem #2. Attack with a Pair of Possible Messages (20 points)

Define *two* functions "EAV_choose(length)" and "EAV_attack (m0, m1, c)." Together these will implement the EAV security game we discussed in class.

"EAV_choose(length)" takes as input a length, which could in principle be any positive integer, but in practice in this problem set ranges from 10 to 1000. It should return a pair of two messages (m0,m1) which are each lists of bytes with a number of entries equal to the length. You should choose a pair (m0,m1) which you will find easy to distinguish.

"EAV_attack (m0, m1, c)" takes as input two messages (each a list of bytes) and a ciphertext (a longer list of bytes generated via the Enc function provided). You may assume that m0 and m1 are the messages you chose via "EAV_choose"; they are provided to make sure your attack has access to them. "EAV_attack" should return 0 if c is an encryption of m0 for some key and 1 if c is an encryption of m1 for some key.

Your attack will be tested on a number of instances with the length being 10 bytes long, 100 bytes long, and 1000 bytes long. You can get partial credit if your attack works for some of these lengths but not others.

Solution: For this problem, any solution to problem 3 will also work using any pair of messages you choose. There are also a few solutions that don't generalize easily to problem 3. The bottom line, though, is that any solution will build on one of the weaknesses identified for problem 1. You first separate out the IV as the first 256 bytes of the ciphertext and then attack the remainder of the ciphertext.

1. If you choose m_0 and m_1 to both have 0 for the first byte, then you can use approach 2 from problem 1 to find a small list of candidates for k. A search over k will then rapidly reveal which message you were given.

2. Alternatively, you can use messages m_0 as all 0's and m_1 as random. Then any approach to problem 1 will work since the m_1 ciphertext is of the form (IV, random string). Choosing m_1 as non-random but still different from m_0 in most places will generally work here as well, although if it is always the same or repeats frequently, approach 4 will fail to distinguish the m_0 and m_1 ciphertexts, since both will have repeats.

Problem #3. Attack with a List of Possible Messages (20 points)

Define a function "decrypt(m_list, c)" which takes as input a list of possible messages (each of which is itself a list of bytes) and a ciphertext (a list of bytes) which is an encryption of one of the messages using the stream cipher. The function should return an index into the list of messages to specify which message encrypts to give the ciphertext. The list of messages could in principle be any length, but the autograder will test your program with a list of 20 possible messages. The length of each message in the list will be the same, and again that length could in principle have any value.

Your attack will be tested on multiple instances with the length of messages being 10 bytes long, 100 bytes long, and 1000 bytes long. You can get partial credit if your attack works for some of these lengths but not others.

Solution: The basic strategy here will be to try a solution to problem 1 stepping through the possible messages.

1. Strategy 1 for problem 1 works well. Simply step through possible values of k and for each one, decode a message from the ciphertext based on that value of k, and compare with each of the possible messages. If it matches, return that message and halt. If none of the possible messages match, step to a different value of k. The actual messages used in the test were random, so if you sped things up by looking at only the first 10 bytes (or whatever) of the message, that would almost certainly be sufficient to identify the correct message. However, if the messages were non-random, they might differ in only a few places, and you would need to identify and check those locations in order to distinguish the messages.

2. Strategy 2 for problem 1 is a little harder to implement here, but it can be done as well. In particular, we can separate out the IV and a list of bytes which have the form $x_i + y_i \bmod 256$, where x_i is the ith output of the stream and y_i is the ith byte of the message. If we step through the possible messages, we can get a sequence of possible values for y_i. Subtracting these off, we get a candidate for the pseudorandom stream. Using the approach from strategy 2 for problem 1, we can then identify a small set of possible values of k and test all of them against the stream candidate. If one matches, return the current message value and halt. Otherwise, continue to the next message.

3. Strategies 3 and 4 from problem 1 can be used here much like strategy 2: Step through candidate message values and subtract them off to get a candidate stream, which is then evaluated as in problem 1. When the possible messages are random, this works as well as it did in problem 1 (which is to say, better for longer lengths), but when some are non-random, it can potentially fail. For instance, if two messages agree in most but not all places, strategy 3 will have trouble; its ability to distinguish these two messages depends on the number of differences rather than the length of the message. If multiple messages have short-period repeats in them, that will cause problems for strategy 4 since the candidate streams produced by those messages may all have repeats if one of them is correct.