CS 411 - Homework 2

Ege Demirci - 28287

Sabanci University, Fall 2023-2024, Cryptography

1. Question - 1

1.1. Part a:

Firstly, I received the number $n = 542$ from the server. To determine the number of elements in the group Z_n^*, I used Euler's totient function, $\phi(n)$, which computes the number of integers less than n that are relatively prime to n. The reason I calculated $\phi(542)$ is that in group theory, the size of the group Z_n^* is exactly the number of integers from 1 to $n-1$ that do not share any common factors with n except for 1. After running my code, I found out there are 270 elements in the group and sent this number to the server using the provided `checkQ1a` function, which confirmed my answer was correct.

1.2. Part b:

Next, I had to find a generator for the group Z_{542}^*. A generator is an element which, when raised to different powers, produces every other element in the group. My script tested various elements to see if they could produce all the elements in the group by exponentiation. The element 15 turned out to be a generator because raising it to different powers modulo 542 cycled through all the elements of the group. After identifying 15 as the generator, `checkQ1b` function, which confirmed my answer was correct.

1.3. Part c:

Finally, the server asked for a generator of a subgroup of Z_{542}^* with an order $t = 27$. To approach this, I looked for an element within Z_{542}^* that, when raised to the power of 27, gives a result of 1 modulo 542. This element needed to be such that its powers generate exactly 27 distinct elements before repeating, matching the order of the subgroup. I ran my script to check elements against this condition and discovered that the number 385 was such a generator. This was because 385 raised to any power less than 27 did not yield 1 modulo 542, satisfying the properties of a subgroup generator. I submitted 385 using the `checkQ1c` function, and the server confirmed it was correct.

2. Question - 2

Since we know that $n = p \cdot q$, we can calculate $\phi(n)$ as $(p-1) \cdot (q-1)$. We can also obtain the number d by using the `modinv` function and the parameters ϕ_n and e. Finally, I used the 3-parameter `pow` function to find the decrypted message. This last function resulted in the m which the question asks, and after converting it to bits and decoding using UTF-8, I got the following message: "I think I have 793 unread e-mails. Is that a lot?"

3. Question - 3

Firstly, the numerical key is transformed into a byte object, a required format for cryptographic functions in the Salsa20 algorithm. Secondly, the nonce is extracted from the third ciphertext; this is because all ciphertexts use the same nonce and key. In the process, I tried all 3 ciphertexts with

the help of the provided IPython notebook, and I could only break ciphertext3 and extract the nonce from there. Thirdly, an attempt to decrypt each ciphertext is made by iterating from the first byte onwards to find the correct starting point of the ciphertext, compensating for any potential corruption in the transmission. This iterative process is necessary since we do not know how many bytes of the nonce are missing. The try-except block handles any errors during decryption attempts, ensuring that the loop continues to try different starting points until successful decryption occurs. Once the correct nonce starting point is found, the Salsa20 cipher instance decrypts the ciphertext and reveals the plaintext.

Here are the outputs:

- The first principle is that you must not fool yourself, and you are the easiest person to fool.

- Somewhere, something incredible is waiting to be known.

- An expert is a person who has made all the mistakes that can be made in a very narrow field.

4. Question - 4

When solving the congruences, I check if b is divisible by the GCD of a and n. If it's not, there's no solution. If it is, I divide a, b, and n by the GCD to simplify the equation. After that, I find a particular solution using the modular inverse of a multiplied by b, modulo n. For equations with multiple solutions, I find all of them by computing the solution set based on the GCD of a and n, as each solution will be spaced evenly apart by n modulo the product of n and the GCD of a and n. My script iterates through all the given equations and prints out all possible solutions for x in each congruence.

Here are the outputs:

```
The solutions of 790561357610948121359486508174511392048190453149805781203471x ≡ 789213546531316846789795646513847987986321321489798756453122 are: [111563634314800439832213513866100835794512614711477009341482 6]

The solutions of 789651315469879651321564984635213654984153213216584984653138x ≡ 798796513213549846121654984652134168796513216854984321354987 are: No solution exists.

The solutions of 654652132165498465231321654946513216854984652132165849651312x ≡ 987965132135498749652131684984653216587986515149879613516844 are: [184045108563697882707983051431202214996694119114301061438590 0, 4573017168579321153146925263568627765759266852067838063135011]

The solutions of 798442746309714903987853299207137826650460450190001016593820x ≡ 263077027284763417836483408268884721142505761791336585685868 are: [120574576795431477647425259344685590574672051332591719355582, 16920414540719870513978988950415999365363127960851309070161 43, 326350833134854262514837253073851428249795354083767009467670 4, 483497520862509819889884616643542862845959428559020928233726 5]
```

Figure 1. The output.

5. Question - 5

In my Python script, I evaluated the maximum period of sequences generated by three different binary connection polynomials using a Linear Feedback Shift Register (LFSR). To achieve this, I used two main functions which are already provided: `LFSR`, which simulates the LFSR operation, and `FindPeriod`, which calculates the period of a sequence. I first defined the connection polynomials $p_1(x)$, $p_2(x)$, and $p_3(x)$ in binary form, with each array `C_A`, `C_B`, and `C_C` representing the coefficients of the respective polynomials. For each polynomial, I generated a random initial state and used the `LFSR` function to create a keystream of length 256. To determine if the generated sequence achieved the maximum period, I compared the period found by `FindPeriod` against the theoretical maximum $2^L - 1$, where L is the length of the LFSR. According to my output, polynomial $p_1(x)$ and $p_3(x)$ generated sequences with the maximum period, while $p_2(x)$ did not, as indicated by the boolean assessments at the end of my script.

Here is the output:

Testing Polynomial A (p_1): Initial state: [0, 1, 0, 1, 0, 0, 1] Period for $p_1(x)$: 127

Testing Polynomial B (p_2): Initial state: [1, 0, 1, 0, 1, 0] Period for $p_2(x)$: 21

Testing Polynomial C (p_3): Initial state: [0, 1, 1, 0, 0] Period for $p_3(x)$: 31

Assessment:

A generates maximum period: True

B generates maximum period: False

C generates maximum period: True

6. Question - 6

In my analysis, I assessed the predictability of sequences generated by different random number generators by calculating their linear complexity. I defined a function for this, that uses the Berlekamp-Massey algorithm to determine the linear complexity (L) of a given binary sequence. This algorithm is represented by the function 'BM' in my code, which was already given. I compared the calculated linear complexity against an expected threshold, calculated as half the length of the sequence plus $\frac{2}{9}$. This was given in the lecture slides, so I directly used it. If the linear complexity of a sequence was greater than this expected value, the sequence is "not predictable"; otherwise, it is "predictable". Using this function, I analyzed three sequences, 'x1', 'x2', and 'x3', and printed out whether each was predictable based on its linear complexity relative to the expected complexity.

Here is the output:

x1: predictable

x2: not predictable

x3: predictable

7. Question - 7

This question was bit tricky. Firstly, I focused on the end of the ciphertext which was given in the hint. I expected to find a known plaintext pattern corresponding to the instructor's name, Erkay Savas. This name was first converted into binary using the provided ASCII2bin function, which creates a binary representation of ASCII text. I then isolated the corresponding segment from the end of the ciphertext and performed a bitwise XOR operation with the binary representation of the known text. The result of this XOR operation should theoretically be the final part of the LFSR-generated keystream if the assumption about the encryption method was correct. Next, I applied the Berlekamp-Massey algorithm to the derived keystream segment to determine the shortest LFSR that could produce such a sequence. This algorithm returned the length of the LFSR and the connection polynomial. The connection polynomial here basically represents the taps of the LFSR, which are the points where the output of certain stages of the shift register is fed back and combined with the input. With the length and the connection polynomial known, I set up an LFSR using the obtained configuration and ran it to regenerate the keystream. To ensure that I could recreate the entire keystream used for the original encryption, I used the connection polynomial to seed the LFSR with the obtained end segment of the keystream, assuming that the LFSR is in the same state at that point as it was during the encryption process. The assumption comes from the fact that LFSR generates bits sequentially. Therefore, if we have a segment of the keystream, we should be able to backtrack to figure out what the internal state of the LFSR must have been at the time that segment was generated, as long as we have as many bits of the keystream as the length of the LFSR. This state is what the LFSR would have contained just before it began to produce that segment of the keystream. If we can accurately determine the state of the LFSR at a given point in the keystream, we can then run the LFSR forward from that state to produce the subsequent bits of the keystream. This is what we're doing in our decryption process: we're running the LFSR backward from the state we found, which we're assuming to be the correct state at the end of the ciphertext. In other words, if I configure an LFSR with this state and run, it will produce the preceding bits of the keystream. Here we actually create an LFSR that works in reverse. I then generated the keystream for the entire length of the ciphertext and

performed a bitwise XOR between this keystream and the ciphertext. Since stream ciphers work by XORing the plaintext with a keystream to produce ciphertext (and vice versa), this operation should reverse the encryption and yield the original plaintext message. The binary result of this XOR operation was then converted back to ASCII text using the bin2ASCII function, revealing the decrypted message:

Dear Student,

Outstanding job on tackling this challenging problem!

Congratulations!

Best, Erkay Savas