CS 406 : Cryptography and Network Security Spring 2024

Lab 1 — One-Time Pad and Variations

Instructor: Manoj Prabhakaran *Lab TA:* Nilabha Saha

Challenge 1: Two-Time Pad

The one-time pad, as you have learnt, is perfectly secure. However, its perfect secrecy depends on the key being used exactly once. If the same key is reused, the resulting "two-time pad" is no longer secure as XORing the two ciphertexts gives you the XOR of the plaintexts, which is a lot of information about the two messages. In fact, if the messages are in natural language, it may even allow you to extract both the messages fully given their XOR. In this challenge, you'll do exactly that.

`ciphertext1.enc` and `ciphertext2.enc` contain two one-time pad encrypted ciphertexts encrypted with the same key. One of them is a "flag" used in a Capture The Flag competition (hint: it starts with *cs406{*) and the other one is a normal English sentence. Use the redundancy of the English language to figure out the entire flag. Feel free to look at `encrypt.py` to understand how to use the `strxor` function to XOR two byte objects in python.

Historical Note: *During WWII, the USSR made the mistake of using several one-time pad keys twice and the US intelligence agencies used this blunder to their advantage to decrypt the Soviet secret messages. Look up Venona Project if you're interested in the history!*

Challenge 2: One-Time Pad over Groups

Recall that a finite group is a finite set $\mathbb{G}$ along with a binary operation $\cdot : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$ such that the binary operation $\cdot$ satisfies the following conditions:

1. (Associativity) $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ for any $x, y, z \in \mathbb{G}$.
2. (Identity) There exists an element $e \in \mathbb{G}$ such that $x \cdot e = e \cdot x = x$ for any $x \in \mathbb{G}$. e is called the identity element of $\mathbb{G}$.
3. (Inverse) For every $x \in \mathbb{G}$, there exists an element $y \in \mathbb{G}$ such that $x \cdot y = y \cdot x = e$. The element y is called the inverse of x.

For instance, the set $\{0, 1\}$ with the binary operation of XOR ($\oplus$) forms a finite group. The identity element is 0, the inverse of 0 is 0 and the inverse of 1 is 1. Another example of a finite group is $\mathbb{Z}_n$, the set of all integers modulo n for some natural number n. Given a finite group $\mathbb{G}$ and a positive integer m, the group $\mathbb{G}^m$ consisting of m-tuples of elements from $\mathbb{G}$ is also a group, with the group operation being co-ordinate wise application of the group operation of $\mathbb{G}$.

As discussed in class, the one-time pad can actually be used for messages in any finite group. When using the group $\mathbb{G}^m$, this requires that key is in the form a string of m "characters" each of which is uniformly sampled from $\mathbb{G}$, independent of each other.

In this challenge, you will see a (faulty) implementation of the one-time pad over the group $\mathbb{Z}_{128}^m$ (for some m). Each message character is converted to its 7-bit ASCII representation and treated as a group element of $\mathbb{Z}_{128}$. You do not know the key used in the encryption, but you suspect that it is not uniformly random. Look at the encryption script in `encryptor.py` and figure out a way to retrieve the original message.

Challenge 3: Faulty One-Time Pad Distinguishing Attack

True one-time pad encryption, as you have studied, is perfectly secure, that is, there is no way for any (potentially unbounded) adversary to distinguish between the encryption of string under one-time pad and a string sampled uniformly at random (of the appropriate length) with a non-zero advantage.

Bob, however, felt that the byte `0x00` in the key could potentially lead to security issues because it does not change the message character at all during the encryption. So instead of sampling each key-byte from the interval `[0x00, 0xff]`, he decided to instead sample the key-byte uniformly from the interval `[0x01, 0xff]`. Show that Bob's encryption scheme is no longer perfectly secure by distinguishing between the encryption of a string you provide and the encryption of a random message. (This is slightly different from IND-onetime security as defined in class, but it is also equivalent to perfect secrecy.)

You'll enter the hex-representation of your string (hint: the only thing that will matter is that your string is sufficiently long). Then you'll be presented with two strings: one which is a valid encryption of your string, and another which is the encryption of a random string, using a fresh random key (with the above modification). You need to identify which of the two is the encryption of your string. If you succeed to distinguish between the two 100 times in a row, you'll receive the flag!

Note that you will need to communicate with a server for this challenge. The template script provided to you handles the server communication for you using the pwntools library. You are only required to fill in the attack logic.

Challenge 4: Fixing the Fault

Suppose you are given a long key generated using the method from the previous challenge (uniformly sampling each key byte from `[0x01, 0xff]`). Could you still repurpose this key somehow to make a perfectly secure encryption scheme?

One way to do so is as follows: Say we denote the plaintext by $m_1, m_2, \ldots, m_n$ where each m_i is a single byte. Considering each byte as a number in base-256, we can think of the message as a number $m = m_1 * 256^{n-1} + m_2 * 256^{n-2} + \cdots + m_{n-1} * 256 + m_n$ in decimal. Converting this number to base-255, say we get the base-255 representation as $p_1 p_2 \ldots p_{n'}$, that is, $m = p_1 * 255^{n'-1} + p_2 * 255^{n'-2} + \cdots + p_{n'-1} * 255 + p_{n'}$, where $p_i \in [0, 254]$ for each $i \in [1, n']$ (where n' is fixed ahead of time – what should it be?). Define the ciphertext as follows: say that for $i \in [1, n']$, we have $c_i = (p_i + k_i - 1) \pmod{255}$ where k_i is the i^{th} byte of the key interpreted as an integer in `[1,255]`. Interpret $c = c_1 c_2 \ldots c_{n'}$ as a number in base-255 and convert it back to base-256 to get a byte sequence which forms the ciphertext. Can you argue that this method of encryption is perfectly secure?

Alice and Bob have now implemented this scheme and have encrypted the flag using it. Your task is to write code to be able to decrypt the flag given the keystream. You are given the ciphertext in `ciphertext.enc` and the keystream in `keyfile`.