In this document, we provide an overview of the encryption algorithm, with a particular focus on the padding scheme, used for the challenge. For each variable, we write within parentesis and in a monospaced font its corresponding name in the server code.

Encryption. Assume we want to pad and encrypt a plaintext p (`ptxt`) which is byte-aligned. Let k (`RSA_KEYLEN`) be the bit-length of the RSA modulus (in our case $k = 1024$). Let $\ell_R = 256$ (`R_LEN`) and let $\ell_p = k - \ell_R - 8$ (`P_LEN`).

1. Pad the plaintext p by prepending `0x00` bytes and a single final `0x01` byte until the result reaches a length of ℓ_p bits. We refer to the value $00 \dots 01 \,||\, p$ as p_{padded} (`ptxt_padded`).
2. Sample a random value R (`rand`) of bit-length equal to ℓ_R bits.
3. Set $S \leftarrow \mathsf{SHAKE256}(R, \ell_p)$.[1]
4. Set $p_{\text{masked}} \leftarrow S \oplus p_{\text{padded}}$, where $\oplus$ is the XOR operation between bit strings.
5. Set $M \leftarrow R \,||\, p_{\text{masked}}$. Note that, by construction, M will be one byte shorter than the RSA modulus (since its length is $\ell_R + \ell_p = k - 8$). In other words, extending M to a k-bit string requires appending an initial `0x00` byte.
6. We denote by m the big-endian integer representation of M. RSA-encrypt m to obtain the final ciphertext $c = m^e \mod N$.

Decryption and Padding Errors. Decryption reverses the padding and encryption algorithms. Additionally, the decryption algorithm carries out two integrity checks:

1. Check whether there is an initial `0x00` in M
2. Check whether p_{padded} starts with any number of (possibly zero) `0x00` bytes, followed by a `0x01` byte.

You will discover that, despite having randomized padding, we can leak information about the padded plaintext *if, during decryption, a failure during the first check can be distinguished from a failure in the second check*. In the wild, such an information leakage may arise if the developers are not careful enough or through side-channels.

First understand how you can detect which error has been triggered (if any). Then, assume that the first check passes (i.e. M *does* have an initial `0x00`). What does this tell you about m (recall that m is the value obtained by interpreting M as a big-endian integer)? In particular, can you find an upper bound for m that is tighter than $m < N$ (where N is the RSA modulus)?

[1] $\mathsf{SHAKE256}$ is a hash function with a variable output length based on SHA3: R is the input to the hash function and ℓ_p determines the output length

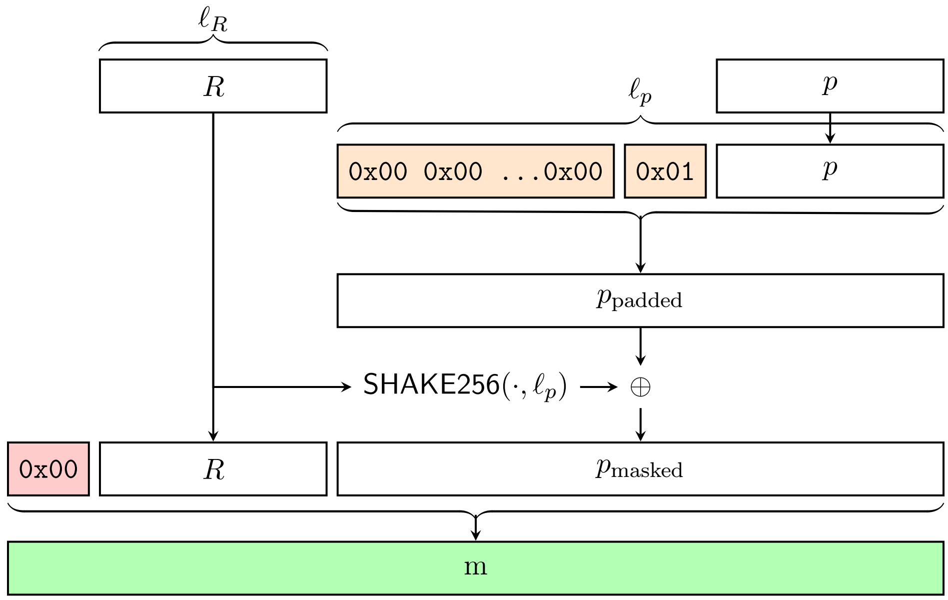

Figure 1: The padding algorithm taking input p (of length at most $\ell_p - 8$), sampling a random string R of length ℓ_R, and outputting a k-bit string m, which is then represented as a big-endian integer and encrypted with RSA. In red and orange, the bytes that are checked during unpadding. In green, m, the value of which you must recover the bit-length.

The Task. The server will encrypt a random message p. Your objective is to leak the bit-length of m, that is the minimum amount of bits required to represent m in big-endian byte order. In Python terms: what is the value of `m.bit_length()` for an integer m. In other words, find i such that $2^{i-1} \leq m < 2^i$ when m is represented as a big-endian integer. In order to win, you must be able to guess the correct value of i for 256 times in a row.

Good luck!