ELEC2760 - Exercise Session #1: Linear Cryptanalysis

Exercise #0.

Data representation. We chose to switch data representation to perform the permutation layer. Tools for basis change are provided in `utils.py`. We provide two alternative versions dedicated to the conversion from binary to hexadecimal/decimal representation.

- `dec2bits`: takes a vector of n values in $[0, 2^s - 1]$ and returns the corresponding binary representation that is a vector of $s \cdot n$ binary symbols.
- `bits2dec`: takes a vector of n binary values and returns the corresponding hexadecimal representation that is a vector of n/s symbols in $[0, 2^s - 1]$.

Test the functions using some handmade examples (data is represented as row vectors). For the present blockcipher we will study in this exercise session, we will manipulate 4-bit words such that $s = 4$.

Exercise #1. The first task in this exercise session is to implement the cipher against which we will perform cryptanalysis. For simplicity, we chose to use a toy cipher based on a real (standardized) lightweight cipher, namely PRESENT. 4 rounds of this toy cipher are represented at the end of the document. It has message and key lengths of 32 bits, with an SPN (Substitution-Permutation Network) structure that iterates non-linear layers (i.e. S-boxes) and linear ones (i.e. bit-permutations), interleaved with key additions (i.e. bitwise XORs). No key schedule is performed: the key used is the same (i.e. the master key) in each round. You may have noticed that an additional key addition is performed at the end of the cipher. You should be able to explain why at the end of Exercise 3.

We provide `player.npz` and `sbox.npz` files that contain the tables used to perform both the substitution and permutation layers (and their inverse). The j-th coordinate of `Sbox` contains the output of the Sbox for input j. Hence to apply Sbox to a variable x (decimal representation), you should perform `sbox[`x`]`. Similarly, `player[`j`]` contains the index of the input bit corresponding to the j-th output bit of the permutation layer. Hence to apply the permutation layer to a 32-bit string b, you should perform b`[player]`. Your goal is to implement the toy cipher encryption and decryption using these tables. Your code should pass the tests in the main of `e1.py`. To implement the encryption and decryption function, you should fill the `encrypt` and `decrypt` functions which takes as input 8 nibbles for both plaintext/ciphertext and key.

Exercise #2. The first task before attacking the toy cipher using linear cryptanalysis is to find a good linear approximation. This is the goal of this exercise.

1. Starting with one S-box, compute and print the biases table of the PRESENT S-box. To do so, fill the file `e2.py`.
2. Chain two S-boxes to form a linear approximation over 2 rounds of the toy cipher.
3. Using the *piling-up lemma* and the biases table, estimate the bias of the approximation.
4. By enciphering random plaintexts with a fixed key, evaluate the empirical bias of the approximation. Why using a fixed key? How many samples are required to obtain a precise estimate (compare to the formula of Lecture 3)? To do so, fill the file `e3.py`. Use another approximation than the one proposed in `e3.py`.

Exercise #3. Using one linear approximation, it is possible to recover few key bits using a `Divide & Conquer` attack. You should limit yourselves to a small number of key bits to recover (at most 8). Try to perform the steps below to attack the cipher (with a fixed key). To do this exercise, fill the file `e4.py`.

1. Choose a 2-round linear approximation to be used in the cryptanalysis.
2. Encrypt *enough* plaintexts to perform the attack (what does it mean?).
3. Implement partial decryption: given a key guess and ciphertext, invert the last round.
4. For each guess, compute the empirical bias for the approximation using the samples.
5. Plot the empirical biases: does the correct key guess pop up?
6. *Optional.* Use different approximations to recover the full key.
7. *Optional.* Use the same approximations to recover the key hidden plaintext/ciphertext pairs in `pts_cts_pairs.npz`.

Exercise #4.

When hull effect appears. In the previous exercise, we chose to use 2-round linear approximations as a starting point. But it turns out that we could have attacked more rounds. If you are interested in attacking more rounds, you should do Exercise 2 again but now considering a 3-round approximations. What do you observe for question 4?

4 bis. Find a 3-round approximation for which the input and output masks only contain one bit equal to 1 (e.g. consider S-box $S1$). Compute it bias using the piling-up lemma.

4 ter. Estimate the bias of your approximation with random plaintexts and **one fixed key**. Does something go wrong? Then try a couple of different keys. What do you observe?

4 qua. Try to find all linear approximations starting with the same input mask and ending with the same output mask[1]. Estimate their biases using the piling-up lemma. Compare the key masks obtained for these three linear trails and evaluate the corresponding parity bits for the keys you used in 4 ter. Can you explain what happened?

Speeding-up partial deciphering. For a number N of samples and k of bit to guess, the complexity of the naive algorithm for attacking the cipher is $\mathcal{O}(N2^k)$: each sample is deciphered for all key guesses. In a classical cryptanalysis scenario, an attack against a cipher targets the largest number of rounds that can be attacked. In such a scenario, the required number of samples N is usually much larger than the number of bit of the key guess k. Therefore, it is possible to reduce the complexity of the deciphering down to $\mathcal{O}(N + 2^{2k})$. Can you find how to obtain this gain (hint: never compute the same thing twice and remember that $N \gg k$). Then implement it for attacking 3 rounds of the cipher and recovering 4 bits of the key (in this setting the improvement should be significant).

Using the Walsh transform. The naive algorithm for computing biases table for an n-bit S-box is $\mathcal{O}(2^{3n})$: for each input mask and each output mask, evaluate the bias using the 2^n possible inputs. This is tractable when n remains small (e.g. 4 for PRESENT) but some ciphers have bigger S-boxes. For instance AES already has 8-bit S-boxes that is the complexity of computing the biases table becomes 2^{24} (there exist even larger S-boxes). We propose to investigate the use of Walsh transform to decrease the complexity to $\mathcal{O}(n2^{2n})$.

Let T be an array of 2^n elements, its Walsh transform $\tilde{T}$ is defined as:

$$\tilde{T}(\alpha) \triangleq \sum_{X \in \mathbb{F}_2^n} (-1)^{\alpha \bullet X} T(X).$$

This transform can be performed in $\mathcal{O}(n2^n)$ using a butterfly algorithm (as for the Fourier transform). You can use the package `simpy` to perform this in Python.

How could you use this algorithm to build the biases table in $\mathcal{O}(n2^{2n})$?

[1] A linear approximation is a pair of input/output masks. Two different paths corresponding to the same pair of input/output masks are usually named *linear trails*, or *linear characteristics*.

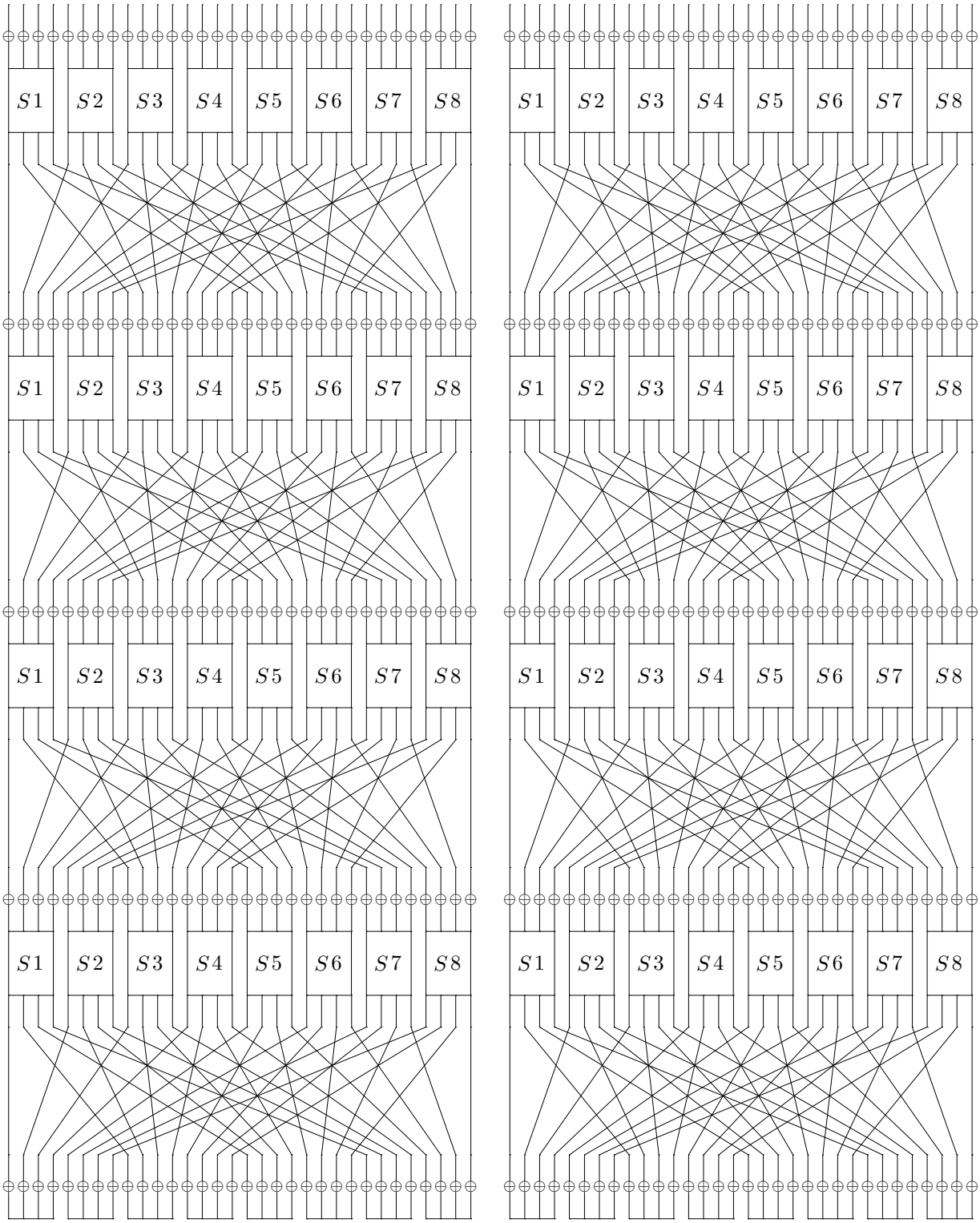
Using a Branch & Bound algorithm to find approximations. This additional question requires more programming skills than others. We provide a short description of an algorithm that searches for the best linear approximation bias for a given number of rounds. Try to program it and if possible perform searches for different numbers of rounds.

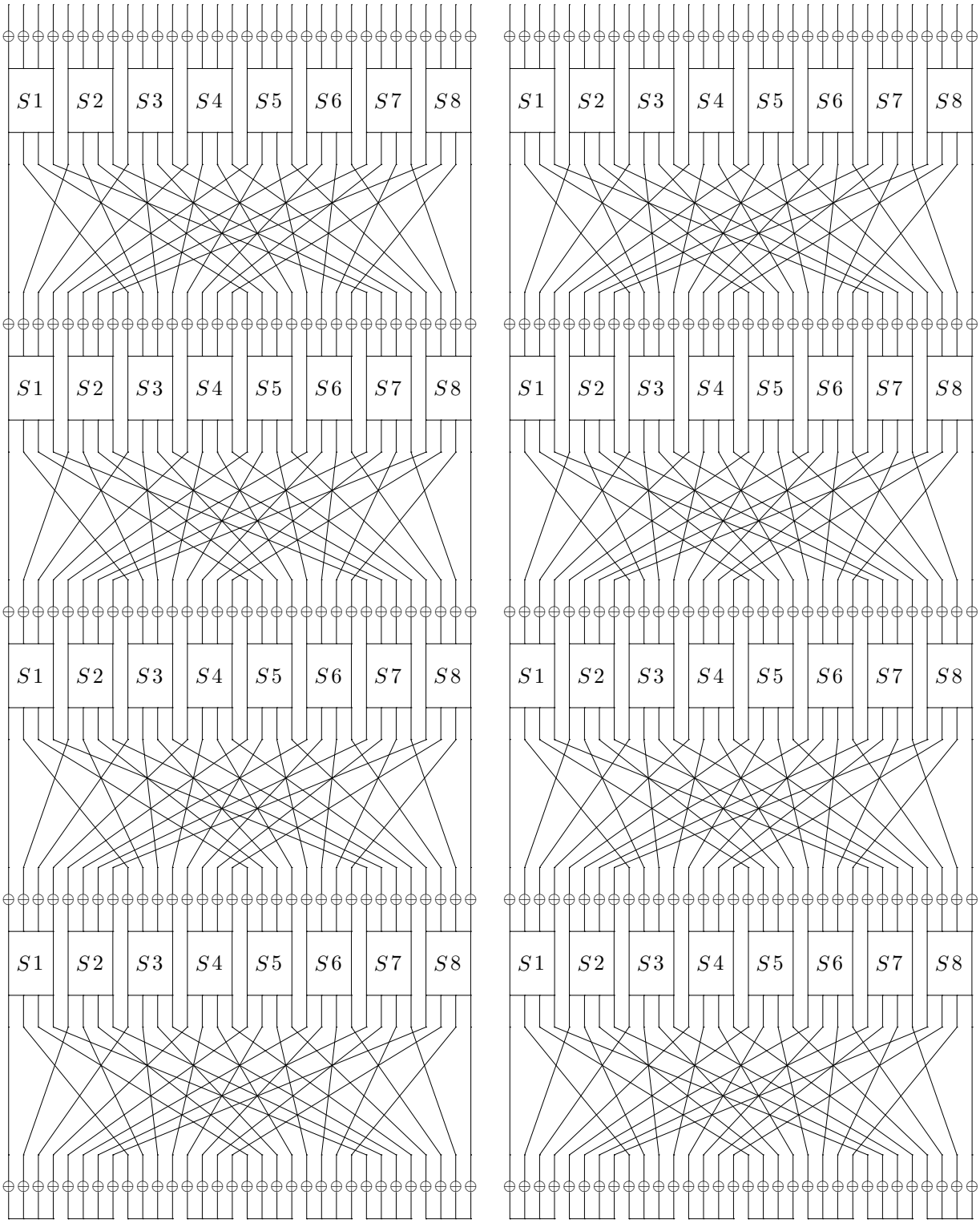
Require: A number r of rounds, best biases for smaller number of rounds $B_1, \ldots, B_{r-1}$.
Ensure: Best r-round linear trail bias $\bar{B}_r$.

$\bar{B}_r \leftarrow 0$
for all α_1 **do**
  $\varepsilon_1 \leftarrow \max_{\beta_1} \epsilon(\alpha_1, \beta_1)$;
  if $2 \cdot p_1 \cdot B_{r-1} \geq \bar{B}_r$ **then**
    Call procedure `Round-2`
  end if
end for
return $\bar{B}_r$

`Round-`i
$\alpha_i \leftarrow$ `Player`(β_{i-1})
for all β_i **do**
  $\varepsilon_i \leftarrow \epsilon(\alpha_i, \beta_i)$
  if $2^{i-1} \cdot \varepsilon_1 \cdots \varepsilon_i \cdot B_{r-i} \geq \bar{B}_r$ **then**
    Call procedure `Round-`$(i+1)$
  end if
end for
return

`Round-`n
$\alpha_r \leftarrow$ `Player`(β_{r-1})
$p_r \leftarrow \max_{\beta_r} \epsilon(\alpha_i, \beta_r)$
if $2^{r-1} \cdot \varepsilon_1 \cdots \varepsilon_r \geq \bar{B}_r$ **then**
  $\bar{B}_r \leftarrow 2^{r-1} \cdot \varepsilon_1 \cdots \varepsilon_r$
end if
return


S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8


S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8
S1 S2 S3 S4 S5 S6 S7 S8