AI in Cryptography – project/laboratory 3

Topic: Searching for "Good" 8×8 S-boxes with AI / Evolutionary Methods

Goal of the Exercise:

You will start from **random 8×8 S-boxes** (permutations on 256 values) and use **AI / ML / evolutionary / other heuristic methods** to search for **better S-boxes**.

"Better" here means, primarily:

- higher **minimum component nonlinearity** (target: **112**, which is the theoretical maximum for balanced 8-bit components of a permutation),
- and, as secondary goals: good **differential uniformity (DU)**,
  good **SAC/BIC** behaviour, high **algebraic degree**, and no **fixed points** or **opposite fixed points**.

You cannot beat the theoretical maximum nonlinearity for 8×8 permutation S-boxes (112), but you can:

- reach **112** more often than random search,
- simultaneously improve other cryptographic criteria.

You are free to choose the optimisation method (e.g., evolutionary algorithm, RL-like search, neural network, cellular automata), but you must **define it clearly** and compare it to basic baselines.

Objects and what we are searching over

- An **8×8 S-box** is a mapping
  S: {0,1}^8 → {0,1}^8
  usually required to be a **permutation** on the set ({0,1, ...,255}). You can represent it as an array of 256 distinct integers from 0 to 255.
- We search over the **space of permutations** of 256 elements (the set of all possible 8×8 S-boxes).

Your job is to design a search procedure that starts from random permutations and **iteratively improves** them with respect to a **fitness function** built from the metrics below.

Core definitions (informal, but precise enough to implement)

1. XOR

- Bitwise XOR is addition modulo 2 performed **bit-by-bit**.

- For scalars or bytes, you can think of `a XOR b` as: each output bit is 1 if the input bits differ, 0 if they are equal.

2. Component Boolean functions of an S-box (intuitive description)

An 8×8 S-box takes an 8-bit input and returns an 8-bit output.
You can think of it as being built from **8 separate Boolean functions**, one for each output bit.

- Let the S-box be a permutation table with 256 entries:
  for each input value $x \in \{0, \ldots, 255\}$,

  you have one output value S(x) (also 0…255).

- If you write each output value S(x) in binary (8 bits), you get:

  S(x)=(b7(x), b6(x), …, b1(x), b0(x)),

  where each bj(x) is a single bit (0 or 1) – the **j-th output bit** for input x.

Now you can define **8 Boolean functions**:

- The **first Boolean function** f0 is:
  f0(x)=b0(x) → its truth table is the least significant bit of every S(x), taken in order for all x=0,1, …,255.
- The **second Boolean function** f1 is:
  f1(x)=b1(x) → its truth table is the second bit of every S(x).
- …
- The **eighth Boolean function** f7 is:
  f7(x)=b7(x) → its truth table is the most significant bit of every S(x).

In other words:

To get the truth table of the j-th Boolean function, take the j-th bit of **each output value** of the S-box (for all 256 inputs), and line them up in order. That sequence of 0/1 values *is* the truth table of that Boolean function.

These 8 Boolean functions together fully describe the S-box, since combining their bits back gives you the 8-bit output for every input. For each of these functions you can then compute **nonlinearity**, **algebraic degree**, etc., and the S-box quality is usually summarised by the **worst (minimum) value** among its 8 components (or over all nonzero linear combinations of them, depending on the convention).

3. Nonlinearity (NL) of a Boolean function

- Intuition: nonlinearity measures **how far** a Boolean function is from **all affine (linear + constant) functions**.
- Formal definition (you do *not* need to implement the brute-force version):
  Nonlinearity of (f) is the **minimum Hamming distance** between (f) and any affine function.
- Efficient way via Walsh–Hadamard spectrum (recommended):

1. Build the truth table of (f) as a vector of length (2^n) (here (n=8)).
2. Map bits to ((-1)^{f(x)}) values (so 0 → +1, 1 → –1).
3. Compute the **Walsh–Hadamard transform** (FWHT) of this vector.
4. Let (M) be the maximum absolute value in the Walsh spectrum.
5. For (n=8),
   NL(f) = 2^{n-1} – 1/2 M = 128 – M/2

- **S-box nonlinearity** (what we optimise):
  Take the **minimum** nonlinearity over all nonzero masks (v in {1, …,255}) of the component functions (f_v).
  This is called the **minimum component nonlinearity**. Your target is **112**.

4. Differential Uniformity (DU)

- We study differences (Δx) at the input and resulting differences (Δy) at the output.
- For each nonzero input difference (Δx in {1, …,255}):
  1. For all (x in {0, …,255}), compute
     (Δy = S(x) ⊕ S(x ⊕ Δ x)) (XOR).
  2. Count how many times each (Δy) occurs (build a histogram for this Δx).
- The **differential uniformity DU** of (S) is the **maximum** count over all (Δx not equal 0) and all Δy.
- Interpretation: smaller DU means better resistance to differential attacks; for 8×8 S-boxes, DU = 4 is considered very good.

5. Strict Avalanche Criterion (SAC)

We consider a hash-like mapping; here you can apply the idea directly to the S-box (for each input bit and each output bit).

- For each input bit position (i) (0..7) and many random inputs (x):
  1. Compute (y = S(x)).
  2. Flip only bit (i) of the input: (x' = x ⊕ e_i) (vector with 1 at bit i, 0 elsewhere).
  3. Compute (y' = S(x')).
  4. Record which output bits changed: (Δ = y ⊕ y').
- Ideally, for a good S-box, these probabilities are close to **0.5**.

6. Bit Independence Criterion (BIC)

- After flipping an input bit (i), you look at **which output bits change**.
- For each pair of output bits ((j, k)), and across many trials, you have two binary sequences (Δy_j) and (Δy_k) (1 if bit flipped, 0 otherwise).
- BIC asks whether these sequences are **independent**.

Algorithmically:

1. For a fixed input bit (i), repeat the SAC experiment many times and record (\Δ y_j) for each output bit.
2. For each pair ((j,k)), compute a correlation measure between their flip sequences (e.g., Pearson correlation or the φ coefficient).

3. For a good S-box, these correlations should be **close to zero**.

7. Fixed points and opposite fixed points

- A **fixed point** is an input (x) such that (S(x) = x).
- An **opposite fixed point** is an input (x) such that (S(x) = x ⊕ 0xFF) (bitwise complement).
- Count how many such points exist. For cryptographic S-boxes we usually prefer **zero**.

8. Algebraic degree

- Each component function can be written as a polynomial over GF(2) in the input bits (Algebraic Normal Form, ANF).
- The **degree** of f is the maximum number of variables in any monomial with nonzero coefficient in its ANF.
- The **algebraic degree of S** is the maximum degree over all components.

(You don't need the explicit polynomial; you can compute the degree from the truth table via a Möbius transform.)

Your task

1. **Represent** S-boxes as permutations of 0…255 and implement checks that:
   - the S-box is indeed a permutation (no duplicates, all values 0...255 present).
2. **Implement metrics** for a given S-box:
   - minimum component nonlinearity (over all nonzero masks (v)),
   - differential uniformity (DU),
   - SAC summary,
   - BIC summary (e.g., distribution of correlations between output bits, optional),
   - number of fixed points and opposite fixed points,
   - algebraic degree.
3. **Baseline study**:
   - Generate a sample of random S-boxes (e.g., 100–300).
   - For each, measure the metrics above.
   - Plot/describe the **distribution** of: minimum NL, DU, degree, SAC deviations, fixed points.
4. **Define a fitness function** that combines the metrics:
   - Prioritise **minimum nonlinearity** (e.g., lexicographically or with the largest weight).
   - Penalise high DU (especially values > 4).
   - Reward higher algebraic degree.
   - Penalise fixed and opposite fixed points.
   - Penalise deviations of SAC from 0.5.

   Explain your design: why these terms, why these weights or priority rules.

5. **Choose a search method** (at least one of):
   - evolutionary algorithm / genetic algorithm (mutation on permutations, optional crossover),
   - reinforcement-learning-style search (policy for choosing modifications),

- surrogate-model-guided search (ML model approximating fitness),
    - cellular automata or other heuristic scheme.

    Clearly describe:

    - the representation (how candidates are stored),
    - variation operators (how you modify an S-box),
    - selection strategy (how you keep or replace candidates).
6. **Run experiments**:
    - Start from random S-boxes and apply your method for a fixed evaluation budget (e.g., total number of fitness evaluations).
    - Track the **best-so-far metrics** as a function of evaluation count or generations.
    - Compare against **baselines**:
        - random search (simply sampling new random permutations),
        - simple hill-climbing (only accept changes that improve fitness).
7. **Analyse results**:
    - Show convergence plots (minimum NL, DU, fitness vs. evaluations).
    - Present the best S-box you found (or top few) and its metrics.
    - Compare to the baseline distributions:
        - Did you achieve minimum NL = 112?
        - Did you achieve DU $\leq$ 4?
        - Did you reduce fixed/opposite points to 0?
        - How does SAC/BIC look compared to random S-boxes?
8. **Write a short report** (3–5 pages):
    - Problem description in your own words.
    - Definitions of the metrics you used.
    - Description of your fitness design and search method.
    - Baseline vs. your method: plots, tables, commentary.
    - Discussion: trade-offs (e.g., when NL improves, does DU worsen?), difficulties, and what you would try next with more time.

What you should understand at the end

- Why **nonlinearity** is a key metric and why **112** is a theoretical maximum for 8×8 permutation S-boxes.
- How **differential uniformity**, **SAC**, **BIC**, **algebraic degree**, and **fixed points** contribute to the cryptographic "quality" of an S-box.
- How AI / heuristic search can help explore a huge combinatorial space and systematically find S-boxes that outperform naive random choices.