Lab 4 – Random Number Generators
Due Date: 11/17/2025 11:59 pm
Upload Submissions to Canvas

Description:

In security engineering, the key to many cryptographic algorithms like encryption, authentication and digital signatures rely on the quality of randomness on a fundamental level. If random numbers in cryptographic operations exhibit predictability and bias, an attacker can compromise the entire system, regardless of how mathematically sound the rest of the protocol is.
This lab introduces two core approaches to generating randomness: pseudorandom number generation (PRNG) and true random number generation (TRNG). By creating and analyzing these two systems, you will get a better understanding of how randomness is produced, where it can fail and how to evaluate its statistical quality.

Learning Outcomes:

By the end of this lab you should be able to:

- Implement and reason about a simple LFSR pseudorandom generator (seed, taps, period).
- Build a jitter-based TRNG using a floating GPIO pin and time measurements.
- Understand bias in raw TRNG bits and apply the von Neumann debiaser to produce unbiased bits (Extra Credit).
- Measure and compare entropy, autocorrelation and other basic statistical properties of bit streams

Lab Setup:

- Raspberry Pi 5 with Python 3
    - Like previous labs, run the following commands:
    - sudo apt-get update
    - sudo apt-get upgrade
    - pip3 install lgpio matplotlib numpy
- Connect the Buzzer to the Raspberry Pi's GPIO18 (Don't forget to ground the Buzzer: GPIO18 (pin 12)->Buzzer->GND (pin 6)); Use previously provided buzzer.py to ensure it works properly.
- Connect an LED to the Raspberry Pi's GPIO27 (Ground the LED through a resistor between 100-500 Ω: GPIO27 (pin 13)->LED->[Ω]->GND (pin 6)).
- Connect a loose wire to GPIO17 (pin 11) to induce jitter for TRNG.

Starter Code Setup:

You are provided with three starter skeleton Python files in your Lab 4 folder:

- lfsr_prng.py – Implement a 6-bit LFSR PRNG (bits, taps) with optional dynamic seeding, LED and buzzer feedback
- trng.py – Implement a TRNG that samples a floating input and uses time.time_ns() deltas to extract bits with LED and Buzzer Feedback
- analysis.py – Implement analysis tests to compute entropy, autocorrelation, run statistics, and plot histograms. Uses LED and Buzzer feedback for displaying RNG with higher entropy.
- Extra credit:
    - For trng.py: Implement a von Neumann debiaser instead of taking delta & 1 directly.
    - For analysis.py: Implement additional statistical tests.

Each script will show plots and print metrics to console.

Part 1 – LFSR_PRNG.py TODOs (20 points)

Implement the lfsr(seed, taps, n_bits, n_values) function such that:

- It uses the given seed (an integer), taps (tuple of tap indices), and n_bits register size.
- Produces a list of output bits of length n_values (each element should be 0 or 1). Use the LFSR feedback formula with XOR taps. Hint: take the LSB as the output bit each iteration.
- Avoid the all-zero state (if seed==0, adjust to non-zero).

The rest of the file includes the main function, LED/beep helpers and a plotting scaffold; do not change those.

Deliverable: lfsr() returns the list of random bits.

Part 2 – TRNG.py TODOS (20 + 5EC points)

Implement the trng(bits=NUM_BITS) collection loop function such that:

- It waits for the floating input to be high; use time.time_ns timestamps.
- Computes delta as the difference between current time and previous time.
- Extracts a raw bit from delta (e.g., delta & 1) and appends it to the output list.
- Calls the provided blink_led() helper function to show each bit visually.

Like Part 1, do not change anything else in the file.

Deliverable: trng() returns the list of random bits.

Extra Credit (5 points): Implement a von_neumann(bits) function that converts the raw bitstream into a debiased bitstream.

Part 3 – Analysis.py TODOs (20 + 5EC points)

Implement the entropy(data) function such that:

- It computes the Shannon Entropy per bit.
- H=−p0log2p0−p1log2p1 (handle edge cases where p=0). (Will write fancy equation style later)

Implement the autocorrelation(bits) function such that:

- It computes lag-1 autocorrelation.
- Use numpy or implement directly.

Extra credit (5 points):

- Monobit frequency test: count 0s and 1s; compute proportion and z-score.
- Runs test: check frequency and number of runs of 0s and 1s.
- Autocorrelation for multiple lags: Similar to lag 1 correlation but between 0 and 10.

Part 4 – Lab Report (10 points)

Short report that answers:

- LFSR: chosen taps, seed, period observed.
- TRNG: entropy estimate for raw bits and debiased bits (if implemented).
- Comparison summary: What tests did you run? What did you observe? Which source had higher entropy? Do you think your analysis matches up with your expectations? Why or why not?
- Add all the plots that you collected from the python files to the report at their relevant sections.
- Mention any troubleshooting notes and places where you felt stuck in this lab. This will be used to improve future labs.

What to submit

1) lfsr_prng.py (with TODOs filled): 20 points
2) trng.py (with TODOs filled): 20 + 5 EC points
3) analysis.py (with tests implemented): 20 + 5 EC points
4) Lab report: 10 points
5) Live demo: 30 points