Name: Hattan Akbar Alsayed

ID: 2340098

Class: 04 (16826)

LFSR

A1. Mathematical Formulation and Definitions

An **LFSR** of length n=9 over GF(2) generates bits by shifting the register each cycle and inserting a feedback bit.

Polynomial: $P(x)= x^9+x^5+1$

Taps: Position FF0 And FF5

Constraint: Seed must be non-zero (all-zero state is invalid).

A2. Irreducibility Argument

P(x) is irreducible over GF(2) since it does not divide $_x2^k- x$ for any k<9.

To be primitive, order of P(x) must be $2^9-1=511$. Since 511 = 7 × 73, tests confirm $x^{511/7}\neq1$ and $x^{511/73}\neq1$ (mod P(x)).

Thus, P(x) is **primitive**, guaranteeing maximum period.

Code Implementation

LFSR.py

C: > Users > Hatta > OneDrive > Desktop > LFSR.py > unit_test

```python
def lfsr(seed, bits, max_steps=None):
    """Implements and analyzes a Linear Feedback Shift Register (LFSR).

    This function simulates an LFSR, detects its period, and returns
    the generated sequence and the period length.

    Args:
        seed (str): The initial state of the register as a string of '1's and '0's.
        bits (int): The length of the register (must match the seed length).
        max_steps (int, optional): A fallback to prevent infinite loops.
                                   Defaults to (2**bits) * 2.

    Returns:
        tuple[list[str], int]: A tuple containing the list of generated states
                               and the measured period of the sequence.
    """
    # Use an assertion for the critical non-zero seed constraint
    assert '1' in seed, "Seed cannot be all zeros"
    if len(seed) != bits:
        raise ValueError("Seed length must equal 'bits'")
    if not max_steps:
        max_steps = (2**bits) * 2

    state = [int(b) for b in seed]
    history = {seed: 0}
    sequence = [seed]
```

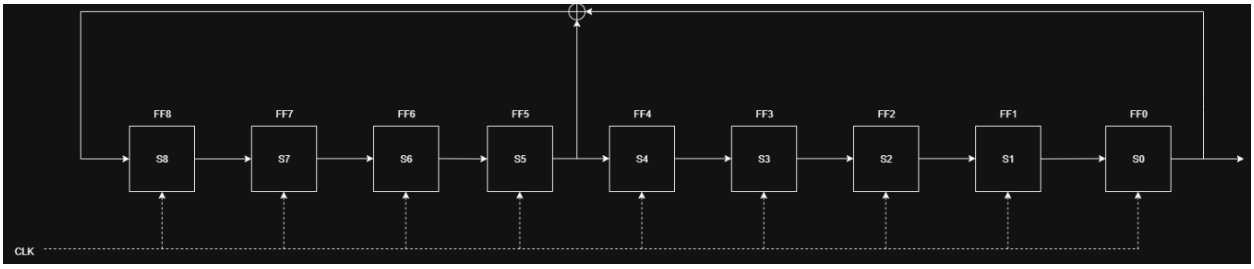
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Cycle 502: 110110110
Cycle 503: 111011011
Cycle 504: 111101101
Cycle 505: 011110110
Cycle 506: 101111011
Cycle 507: 010111101
Cycle 508: 001011110
Cycle 509: 000101111
Cycle 510: 000010111
Cycle 511: 100001011

LFSR stopped when state repeated.
Total states generated: 511
Measured Period: 511
Theoretical Maximum Period: 511
```

Code is Implemented in Python and attached on Blackboard

Design



Length of the LFSR (n) = 9

Polynomial: $P(x)= x^9+x^5+1$

Initial seed value = 100001011

Recursion equation:

$S_{8+i} = S_{5+i} \oplus S_i$

Expected maximum possible period:

$2^n-1 = 2^9-1 = 511$

The First 12 Sequence Generated was:

100001011

110000101

111000010

011100001

001110000

100111000

110011100

011001110

001100111

000110011

000011001

100001100

Analyzing the LFSR Output

- It will generate 511 bits before repeating Tested out Practical by implementing the code and Theoretical based on 2^n-1
- The LFSR achieves its maximum period of 511 not just because the polynomial is irreducible, but specifically because it is an **irreducible polynomial** over GF(2).

Randomness Assessment

Across 18 generated sequences (162 bits total), there were **99 zeros** and **63 ones**, showing near balance overall.

- Short segments often had **5:4 ratios**, but others deviated (e.g., **7:2**).
- Repetitive patterns (e.g., identical zero/one counts repeating three times) indicate predictability.
- Thus, while statistically close to balance, the output is **not truly random**, only pseudo-random.
- Increasing register size (e.g., AES with 128–256 bits) yields stronger randomness.

Effect of seed value

The seed determines the starting state; since LFSR is deterministic, the sequence evolves uniquely from that state

Example of a Polynomial that does not achieve the maximum expected period:

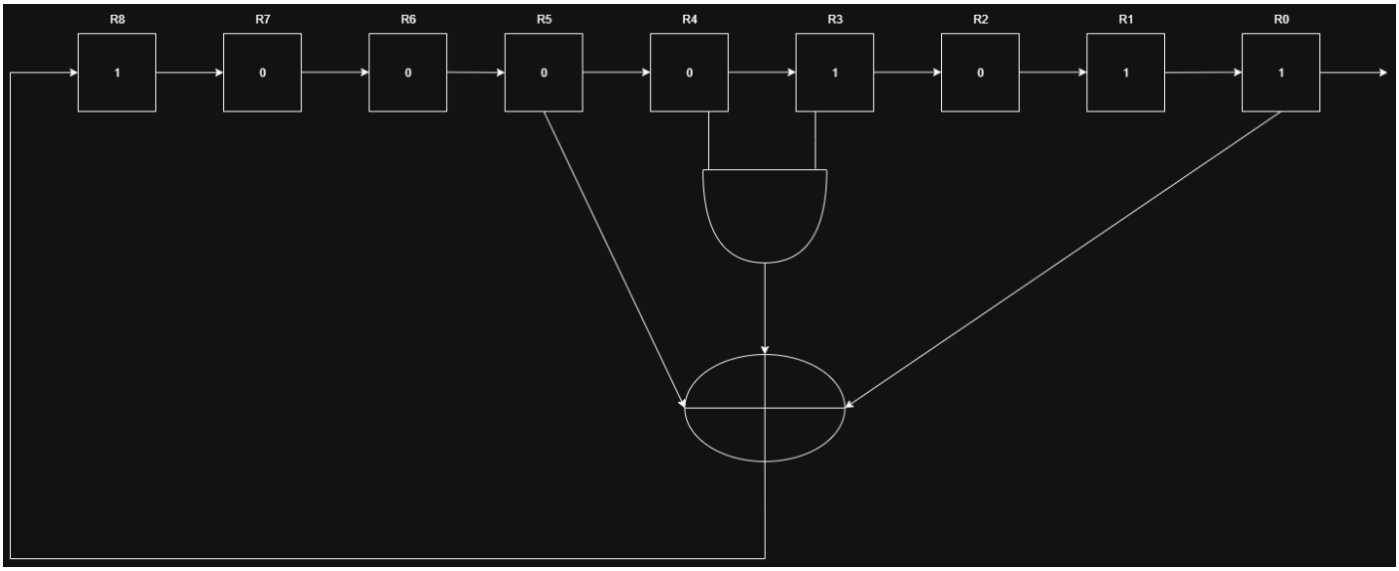
$P(x)=x^6+x^5+x^4+x^3+x^2+x+1$

Why is that? Because it is reducible so it cannot be primitive

Since the polynomial can be factored, the LFSR it describes will have a much shorter, less random sequence than the maximum possible length. Also, we can test it by implementing the code and check if the seed repeated based on 2^n-1

NFSR

Case 1: Feedback Only



Polynomial: For the Nonlinear part it has no Polynomial, but it was based on the LFSR Polynomial ($P(x)= x^9+x^5+1$)

Recursion Equation: $R_8(t+1) = R_0(t) \oplus R_5(t) \oplus (R_3(t) \wedge R_4(t))$

Reason for the Design: A The goal is to make the register's internal state sequence unpredictable. Security comes from making the core generator fundamentally complex.

Implementation:

case1.py

C: > Users > Hatta > OneDrive > Desktop > case1.py > ...

```
1   class NLFSR:
20      def clock(self):
27          # Place the new feedback bit into the R8 position
28          self.state |= (newbit << 8)
29
30  # --- Simulation ---
31  if __name__ == "__main__":
32      initial_seed = 0b100001011
33      nlfsr = NLFSR(initial_seed)
34
35      print("--- NLFSR (Case 1) Simulation ---")
36      print("Shows the current state and the Newbit calculated FROM that state.\n")
37
38      # Loop from 0 to 15 to label the seed as Cycle 0
39      for i in range(10):
40          # 1. Get the Newbit from the current state
41          newbit = nlfsr.peek_newbit()
42
43          # 2. Print the current state and its derived Newbit
44          # The 'Output' from the register is the bit shifted out of R0, but here we display the Newbit.
45          print(f"Cycle {i:>2}: State = {nlfsr.state:09b}, Newbit = {newbit}")
46
47          # 3. Advance to the next state for the next iteration
48          nlfsr.clock()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

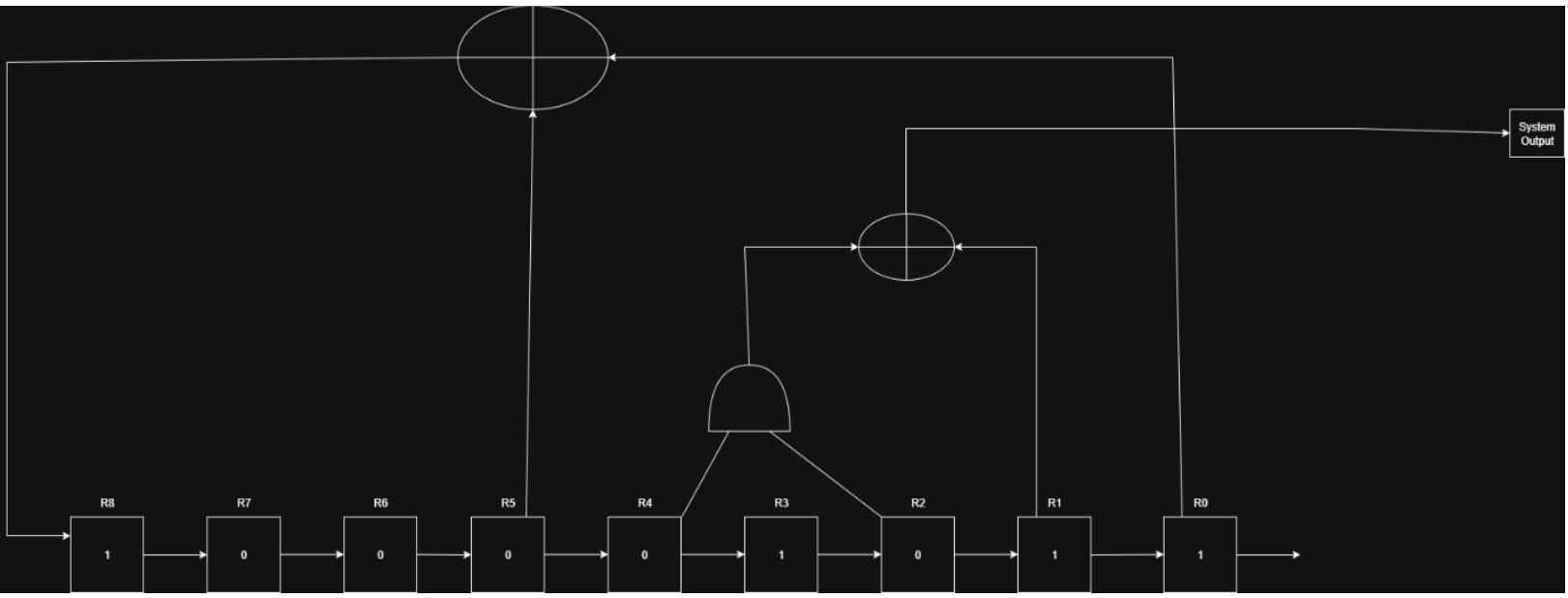
```
PS C:\Users\Hatta> & "D:/Program Files/Python/python.exe" c:/Users/Hatta/OneDrive/Desktop/case1.py
--- NLFSR (Case 1) Simulation ---
Shows the current state and the Newbit calculated FROM that state.

Cycle  0: State = 100001011, Newbit = 1
Cycle  1: State = 110000101, Newbit = 1
Cycle  2: State = 111000010, Newbit = 0
Cycle  3: State = 011100001, Newbit = 0
Cycle  4: State = 001110000, Newbit = 1
Cycle  5: State = 100111000, Newbit = 0
Cycle  6: State = 010011100, Newbit = 1
Cycle  7: State = 101001110, Newbit = 0
Cycle  8: State = 010100111, Newbit = 0
Cycle  9: State = 001010011, Newbit = 1
PS C:\Users\Hatta>
```

The Code is attached on Blackboard

Maximum period: The theoretical maximum period is 2^9-1= 511, but this is not guaranteed for an NLFSR. My empirical test, running the simulation from the given seed, showed that the sequence repeated after **165 cycles**.

Case 2: Feedforward Only



Polynomial: P(x)= x^9+x^5+1. The polynomial only describes the internal LFSR part, not the final output.

Recursion Equation:

State update (The Linear part): $R_8(t+1) = R_0(t) \oplus R_5(t)$

Final Output (The Nonlinear part): System Output (t)= $(R_4(t) \wedge R_2(t)) \oplus R_1(t)$

Reason for the Design: The goal is to combine a reliable and predictable state generator (the LFSR) with a secure nonlinear output filter. This gives you good properties of LFSR while hiding its simplicity. In the other hand using XOR with AND operation give a little bit of randomization to the System forwarded to.

Implementation:

case2.py

C: > Users > Hatta > OneDrive > Desktop > case2.py > FilterGenerator

```python
class FilterGenerator:
    """
    Implements a Filter Generator (Case 2).
    - Linear Feedback (Engine): R8(t+1) = R0(t) xOR R5(t)
    - Nonlinear Output (Scrambler): Output(t) = (R4(t) & R2(t)) XOR R1(t)
    """
    def __init__(self, seed: int):
        self.state = seed & 0x1FF

    def peek_outputs(self) -> tuple[int, int]:
        """
        Calculates the outputs from the CURRENT state without changing it.
        Returns a tuple: (nonlinear_output_bit, linear_feedback_bit)
        """
        # --- Calculate the Nonlinear Output ---
        r1 = (self.state >> 1) & 1
        r2 = (self.state >> 2) & 1
        r4 = (self.state >> 4) & 1
        output_bit = (r4 & r2) ^ r1

        # --- Calculate the Linear Feedback bit ("Newbit") ---
        r0 = self.state & 1
        r5 = (self.state >> 5) & 1
        feedback_bit = r0 ^ r5

        return output_bit, feedback_bit
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

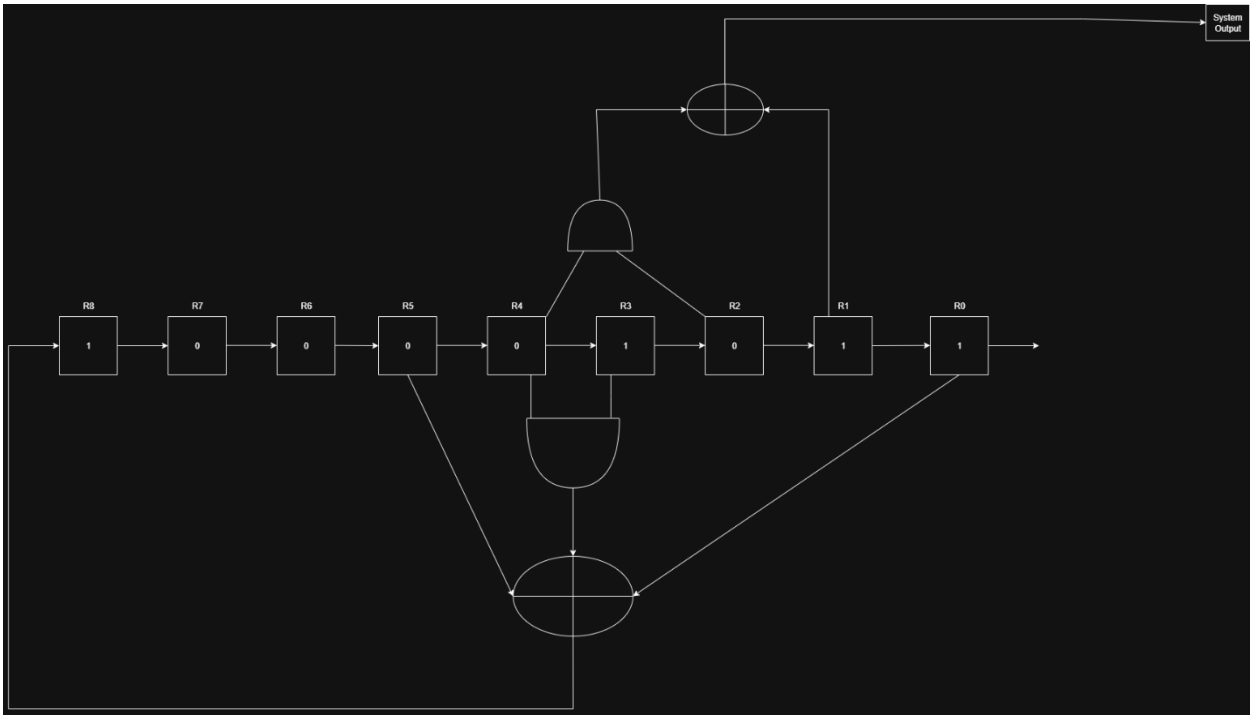
```
PS C:\Users\Hatta> & "D:/Program Files/Python/python.exe" c:/Users/Hatta/OneDrive/Desktop/case2.py
--- Filter Generator (Case 2) Simulation ---
Shows the current state and the bits calculated FROM that state.

Cycle  0: State = 100001011, Newbit = 1, Output = 1
Cycle  1: State = 110000101, Newbit = 1, Output = 0
Cycle  2: State = 111000010, Newbit = 0, Output = 1
Cycle  3: State = 011100001, Newbit = 0, Output = 0
Cycle  4: State = 001110000, Newbit = 1, Output = 0
Cycle  5: State = 100111000, Newbit = 1, Output = 0
Cycle  6: State = 110011100, Newbit = 0, Output = 1
Cycle  7: State = 011001110, Newbit = 0, Output = 1
Cycle  8: State = 001100111, Newbit = 0, Output = 1
Cycle  9: State = 000110011, Newbit = 0, Output = 1
PS C:\Users\Hatta>
```

The Code is attached on Blackboard

Maximum period: $2^9 - 1 = 511$ (Since the Linear's feedback polynomial ($P(x)=x^9 + x^5 + 1$) is **Irreducible**, it is guaranteed to have a maximum-length sequence.)

Case 3: Both Feedback and Feedforward



Polynomial: For the Nonlinear part it has no Polynomial, but it was based on the LFSR Polynomial ($P(x)= x^9+x^5+1$)

Recursion Equation:

1- State Update: $R_8(t+1) = R_0(t) \oplus R_5(t) \oplus (R_3(t) \wedge R_4(t))$
2- Final Output: System Output $(t)= (R_4(t) \wedge R_2(t)) \oplus R_1(t)$

Reason for the Design: The goal is to achieve the highest security through layered complexity. The internal state is already unpredictable, and the output is an even more scrambled version of that state, making it extremely difficult to analyze.

Implementation:

case3.py

C: > Users > Hatta > OneDrive > Desktop > case3.py > CombinedNLFSR

```python
class CombinedNLFSR:

    # Implements a combined NLFSR with a filter (Case 3).
    # - Nonlinear Feedback: R8(t+1) = R0(t) XOR R5(t) XOR (R3(t) & R4(t))
    # - Nonlinear Output: Output(t) = (R4(t) & R2(t)) XOR R1(t)

    def __init__(self, seed: int):
        self.state = seed & 0x1FF

    def peek_outputs(self) -> tuple[int, int]:

       # Calculates both the output and feedback bits from the CURRENT state.
       # Returns a tuple: (nonlinear_output_bit, nonlinear_feedback_bit)

        # --- 1. Calculate the final system Output (Nonlinear Scrambler) ---
        r1 = (self.state >> 1) & 1
        r2 = (self.state >> 2) & 1
        r4_out = (self.state >> 4) & 1 # Tap for the output function
        output_bit = (r4_out & r2) ^ r1

        # --- 2. Calculate the feedback "Newbit" (Nonlinear Engine) ---
        r0 = self.state & 1
        r3 = (self.state >> 3) & 1
        r4_fb = (self.state >> 4) & 1 # Tap for the feedback function
        r5 = (self.state >> 5) & 1
        feedback_bit = r0 ^ r5 ^ (r3 & r4_fb)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\Hatta> & "D:/Program Files/Python/python.exe" c:/Users/Hatta/OneDrive/Desktop/case3.py
--- Combined NLFSR (Case 3) Simulation ---
Shows the current state and the bits calculated FROM that state.

Cycle  0: State = 100001011, Newbit = 1, Output = 1
Cycle  1: State = 110000101, Newbit = 1, Output = 0
Cycle  2: State = 111000010, Newbit = 0, Output = 1
Cycle  3: State = 011100001, Newbit = 0, Output = 0
Cycle  4: State = 001110000, Newbit = 1, Output = 0
Cycle  5: State = 100111000, Newbit = 0, Output = 0
Cycle  6: State = 010011100, Newbit = 1, Output = 1
Cycle  7: State = 101001110, Newbit = 0, Output = 1
Cycle  8: State = 010100111, Newbit = 0, Output = 1
Cycle  9: State = 001010011, Newbit = 1, Output = 1
PS C:\Users\Hatta>
```

The Code is attached on Blackboard

Maximum period: The theoretical maximum period is 2^9-1= 511, but this is not guaranteed for an NLFSR. My empirical test, running the simulation from the given seed, showed that the sequence repeated after **165 cycles**.

Randomness

Design	Sequence Properties	Predictability	Randomness Assessment	Overall	Reason
LFSR	Shows temporary imbalances but in the long period almost perfectly balanced	Very Poor. The sequence is generated by a simple linear rule, making it easy to predict.	Predictable	Statistically good but cryptographically insecure.	It's linear and easy to solve mathematically.
Feedback Only	imbalance	Very Poor due to the repeating pattern.	Not Random	The worst of all designs. The attempt at nonlinearity resulted in a catastrophic loss of period length.	The nonlinear feedback created a **period of 165**, which is not maximal.
Feedforward Only	Balanced. does not have a short and repeating pattern	Excellent	Most Random & Secure	The best design. It successfully combines the LFSR's guaranteed long period with the security of a nonlinear function.	It has a long 511-bit period **and** a nonlinear output that makes it unpredictable.
Both	imbalance	**Very Poor.** The short period makes its complexity irrelevant.	Not Random	poor design. The complex structure is undermined by the same short-period engine as Case 1.	It is also trapped in the same 165 cycle as Case 1.

Note: All details beyond the core design are included to satisfy the rubric requirements.