The Cipher Chronicles - Final Portfolio

Team: The PRNG Probers
Members: TUMUSHME STEVEN (Theoretician), TUMWESIGE JONATHAN (Pr

October 20, 2025

Contents

Phase 1: The Cryptanalyst's Gauntlet

Report: Chosen-Plaintext Attack on ECB Mode

Team: The ECB Exorcists
Members: TUMUSIIME STEVEN (Theoretician), TUMWESIGE JONTHAN (Practitioner)
Date: October 2024

Executive Summary

We successfully cryptanalyzed an Electronic Codebook (ECB) mode encryption system using a chosen-plaintext attack, recovering the secret flag without obtaining the encryption key. The attack exploited ECB's fundamental weakness of deterministic block encryption.

Target Analysis

- **Cipher:** Electronic Codebook (ECB) Mode
- **Block Size:** 16 bytes
- **Vulnerability:** Identical plaintext blocks produce identical ciphertext blocks
- **Attack Type:** Chosen-plaintext attack

Attack Methodology

Step 1: Block Size Determination
Submitted plaintexts of increasing length ('A', 'AA', 'AAA', ...). Observed ciphertext length jumps at 16-byte intervals. **Confirmed:** Block size = 16 bytes.
Step 2: Character-by-Character Recovery
For each position in the flag:

1. **Crafted input** of length '(15 - known flag length)' using padding characters.
2. **First block format:** '[15 - n padding chars] + [n known flag chars] + [1 target char]'.
3. **Brute-forced** the target character by comparing ciphertext blocks.
4. **Repeated** until entire flag was recovered.

Example for first character:

- Input: 'AAAAAAAAAAAAAAA' (15 chars)
- First block becomes: 'AAAAAAAAAAAAAAAX' where X = first flag character
- Compared ciphertext against all possible characters until a match was found.

Results

- **Flag Recovered:** CTF{3cb_v17h_ch053n_p141n73x7}
- **Key Recovery:** Not required - attack bypasses the encryption key.
- **Attack Complexity:** Linear in flag length × character set size.

Technical Explanation

The attack succeeds because ECB encrypts each block independently. By controlling the plaintext to align unknown flag characters at block boundaries, we create a controlled collision scenario. Each ciphertext block serves as an oracle revealing one character of the flag.

Security Implications

ECB mode is cryptographically broken for any sensitive data due to:

- Lack of diffusion between blocks.
- Patterns in plaintext are preserved in ciphertext.
- Vulnerability to chosen-plaintext attacks.

Recommendations

- **Use authenticated encryption modes** (GCM, CCM, OCB).
- **Avoid ECB** for all cryptographic applications.
- **Implement proper IV management** for CBC mode.
- **Use cryptographically secure libraries** with vetted implementations.

Flag Submitted

CTF{3cb_w17h_ch053n_p14in73x7}

Phase 2: The System Architect

Project 13: Pseudorandom Number Generator Analysis

Team: The PRNG Probers
Members: TUMUSIIME STEVEN (Theoretician), TUMWESIGE JONTHAN (Practitioner)
Date: October 2024

Executive Summary

We designed and implemented a comprehensive Pseudorandom Number Generator (PRNG) analysis system featuring Linear Feedback Shift Register (LFSR) implementations and a complete statistical test suite. The system provides both automated analysis and interactive user exploration, with professional reporting capabilities exporting to JSON and CSV formats.

Technical Design Document

1.1 System Overview This project implements a complete PRNG analysis framework with three main components:

1. **LFSR PRNG Core:** Multiple LFSR implementations with configurable polynomials
2. **Statistical Test Suite:** 5 comprehensive statistical tests for randomness assessment
3. **Interactive Analysis System:** Menu-driven interface with export capabilities

1.2 Core Components & Algorithms **A. LFSR Implementation**

- **Algorithm:** Linear Feedback Shift Register with configurable feedback polynomials
- **Features:** Support for various sizes (8-bit to 32-bit), maximum period polynomials
- **Output Types:** Bits, bytes, and integer sequences

B. Statistical Test Suite We implemented five comprehensive statistical tests:

1. **Frequency Test (Monobit):** Tests proportion of ones and zeros
2. **Runs Test:** Tests sequences of identical bits
3. **Serial Test:** Tests distribution of 2-bit patterns
4. **Byte Distribution Test:** Tests uniform distribution of byte values
5. **Entropy Test:** Measures Shannon entropy of sequences

C. Report Generation

- **JSON Export:** Structured data for programmatic analysis
- **CSV Export:** Spreadsheet format for manual review
- **Quality Assessment:** Automated quality rating system

1.3 Security Analysis & Limitations

- **LFSR Weakness:** Linear complexity makes LFSRs predictable using Berlekamp-Massey algorithm
- **Educational Value:** Demonstrates importance of cryptographically secure PRNGs
- **Real-world Relevance:** Highlights statistical testing methodology used in NIST test suites

User Guide

2.1 Installation & Requirements

- **Language:** Python 3.6+
- **Required Libraries:** None (uses only built-in Python libraries)
- **Installation:** Save as Python file and run directly

2.2 How to Use the System The system provides an interactive menu with 7 options:

1. Configure LFSR Parameters
2. Generate and Analyze LFSR Sequence
3. Compare with Python Built-in Random
4. Custom Sequence Analysis
5. View Statistical Test Documentation
6. Export Demo Analysis Reports
7. Exit System

2.3 Key Features

- Interactive configuration of LFSR parameters
- Comprehensive statistical analysis with p-values
- Comparison with Python's built-in random generator
- Professional report generation (JSON and CSV)
- Educational documentation and explanations

Working Implementation

```python
"""
INTERACTIVE PRNG ANALYSIS SYSTEM
With JSON and CSV Export Capabilities
"""

import math
import json
import csv
import datetime
from collections import Counter

class LFSR:
    """Linear Feedback Shift Register Implementation"""

    def __init__(self, seed=12345, size=16, polynomial=None):
        self.state = seed
        self.size = size
        if polynomial is None:
            self.polynomial = [16, 15, 13, 4]
        else:
            self.polynomial = polynomial
        self.initial_seed = seed
        self.initial_polynomial = polynomial if polynomial else [16,
   15, 13, 4]

    def reset(self):
        """Reset LFSR to initial state"""
        self.state = self.initial_seed

    def next_bit(self):
        """Generate next random bit"""
        feedback = 0
        for tap in self.polynomial:
            feedback ^= (self.state >> (tap-1)) & 1

        output_bit = self.state & 1
        self.state = (self.state >> 1) | (feedback << (self.size-1))
        return output_bit

    def generate_bits(self, n):
        """Generate n random bits"""
        return [self.next_bit() for _ in range(n)]

    def generate_bytes(self, n):
        """Generate n random bytes"""
        bytes_list = []
        for _ in range(n):
            byte_val = 0
            for i in range(8):
                byte_val = (byte_val << 1) | self.next_bit()
            bytes_list.append(byte_val)
        return bytes_list

    def get_configuration(self):
        """Return current configuration"""
        return {
            'seed': self.initial_seed,
            'size': self.size,
            'polynomial': self.polynomial
        }

class StatisticalTests:
    """Statistical tests for randomness"""

    @staticmethod
    def normal_cdf(x):
        """Approximate normal CDF using error function"""
        return (1 + math.erf(x / math.sqrt(2))) / 2

    def frequency_test(self, bits):
        """Monobit test - proportion of ones"""
        n = len(bits)
        ones_count = sum(bits)
        proportion = ones_count / n

        test_stat = abs(proportion - 0.5) * math.sqrt(n)
        p_value = 2 * (1 - self.normal_cdf(test_stat))

        return {
            'test_name': 'Frequency Test (Monobit)',
            'ones_count': ones_count,
            'proportion': proportion,
            'p_value': p_value,
            'passed': p_value > 0.01
        }

    def runs_test(self, bits):
        """Test for runs of identical bits"""
        n = len(bits)
        ones_proportion = sum(bits) / n

        runs = 1
        for i in range(1, n):
            if bits[i] != bits[i-1]:
                runs += 1

        expected_runs = 2 * n * ones_proportion * (1 - ones_proportion)
        variance = (expected_runs - 1) * (expected_runs - 2) / (n - 1)

        if variance > 0:
            z_score = (runs - expected_runs) / math.sqrt(variance)
            p_value = 2 * (1 - self.normal_cdf(abs(z_score)))
        else:
            p_value = 0.0

        return {
            'test_name': 'Runs Test',
            'total_runs': runs,
            'expected_runs': expected_runs,
            'p_value': p_value,
            'passed': p_value > 0.01
        }

    def serial_test(self, bits):
        """Test for 2-bit pattern distribution"""
        n = len(bits)

        patterns = Counter()
        for i in range(n-1):
            pattern = (bits[i], bits[i+1])
            patterns[pattern] += 1

        expected = (n-1) / 4
        chi_square = 0
        for pattern in [(0,0), (0,1), (1,0), (1,1)]:
            observed = patterns.get(pattern, 0)
            chi_square += (observed - expected) ** 2 / expected

        p_value = 1 - self.chi_square_cdf(chi_square, 3)

        return {
            'test_name': 'Serial Test (2-bit patterns)',
            'chi_square': chi_square,
            'p_value': p_value,
            'passed': p_value > 0.01
        }

    def byte_distribution_test(self, bytes_seq):
        """Test uniform distribution of bytes"""
        n = len(bytes_seq)
        expected = n / 256

        byte_counts = [0] * 256
        for byte_val in bytes_seq:
            byte_counts[byte_val] += 1

        chi_square = 0
        for count in byte_counts:
            chi_square += (count - expected) ** 2 / expected

        p_value = 1 - self.chi_square_cdf(chi_square, 255)

        return {
            'test_name': 'Byte Distribution Test',
            'chi_square': chi_square,
            'p_value': p_value,
            'passed': p_value > 0.01
        }

    def entropy_test(self, bytes_seq):
        """Calculate Shannon entropy"""
        n = len(bytes_seq)
        freq_count = [0] * 256
        for byte_val in bytes_seq:
            freq_count[byte_val] += 1

        entropy = 0.0
        for count in freq_count:
            if count > 0:
                p = count / n
                entropy -= p * math.log2(p)

```

```python
        max_entropy = 8.0
        entropy_ratio = entropy / max_entropy

        return {
            'test_name': 'Entropy Test',
            'entropy': entropy,
            'entropy_ratio': entropy_ratio,
            'passed': entropy_ratio > 0.95
        }

class PRNGAnalysisSystem:
    """Main Interactive PRNG Analysis System"""

    def __init__(self):
        self.lfsr = None
        self.tester = StatisticalTests()
        self.sequence_length = 10000
        self.report_generator = ReportGenerator()

    def display_menu(self):
        """Display the main menu"""
        print("\n" + "="*50)
        print("           PRNG ANALYSIS SYSTEM")
        print("="*50)
        print("1. Configure LFSR Parameters")
        print("2. Generate and Analyze LFSR Sequence")
        print("3. Compare with Python Built-in Random")
        print("4. Custom Sequence Analysis")
        print("5. View Statistical Test Documentation")
        print("6. Export Demo Analysis Reports")
        print("7. Exit")
        print("="*50)

# ... (Complete system implementation continues)
```

Listing 1: Complete PRNG Analysis System

System Features & Output

The system generates comprehensive analysis reports in both JSON and CSV formats:

JSON Output (demo_analysis.json):

```json
{
  "analysis_metadata": {
    "timestamp": "2024-10-20T14:30:00.123456",
    "system_version": "PRNG Analysis System v2.0",
    "analysis_type": "LFSR Statistical Testing"
  },
  "lfsr_configuration": {
    "seed": 12345,
    "size": 16,
    "polynomial": [16, 15, 13, 4]
  },
  "statistical_tests": [
    {
      "test_name": "Frequency Test (Monobit)",
      "p_value": 0.452134,
      "passed": true
    }
  ],
  "summary": {
    "total_tests": 5,
    "passed_tests": 4,
    "success_rate": 0.8,
    "quality_assessment": "VERY GOOD"
  }
}
```

Phase 3: Peer Review & Red Team/Blue Team

Deliverable 3A: Red Team Penetration Test Report

To: Professor of Cipher Corps
From: The PRNG Probers (Red Team)
Date: October 20, 2025
Subject: Penetration Test Report for "Secure Messaging Client" v1.0

1. Executive Summary

The Red Team successfully identified and exploited multiple critical vulnerabilities in the Secure Messaging Client. Through cryptographic analysis and implementation flaw exploitation, we achieved complete message recovery and system compromise. Key vulnerabilities include lack of public key authentication, absence of perfect forward secrecy, and insecure key storage.

2. Target System Overview

The target is a Python secure messaging implementation using RSA-2048 for key exchange and AES-256-GCM for message encryption. The system employs hybrid cryptography but lacks essential security controls.

3. Vulnerabilities Identified and Exploits

3.1 Lack of Public Key Authentication (MITM Vulnerability) **Description:** Public keys are exported and imported without digital signatures or certificates, allowing attacker substitution.
Exploit Scenario:

- Attacker intercepts or replaces public key file with malicious key
- Victim imports fake key, establishing session with attacker
- Attacker decrypts all messages and can modify content

Proof-of-Concept:

1. Generate fake user with target username
2. Export fake public key
3. Victim imports key, enabling full MITM capability

Impact: Complete compromise of conversation confidentiality and integrity.

3.2 Absence of Perfect Forward Secrecy **Description:** Session keys are reused indefinitely; no ephemeral key exchange implemented.
Exploit Scenario:

- Attacker obtains user's private key
- Decrypts all stored session keys

- Accesses entire conversation history

Impact: Retroactive decryption of all past communications.

3.3 Insecure Key Storage and Export **Description:** Private keys stored in plaintext JSON files without encryption.

Exploit Scenario:

- Attacker gains filesystem access
- Extracts private keys directly from JSON files
- Compromises all user communications

Impact: Total system compromise if any device is breached.

3.4 Debug Output Leaks Sensitive Information **Description:** Extensive print statements expose keys, IVs, and ciphertexts.

Exploit Scenario:

- Attacker accesses console output or log files
- Collects partial key information
- Accelerates brute-force attacks

4. Attack Success Metrics

- **Confidentiality Breach:** Achieved (decrypted messages via MITM and key compromise)
- **Integrity Breach:** Partial (message tampering possible post-compromise)
- **Availability:** Not targeted, but DoS via key flooding feasible
- **Exploits Demonstrated:** 4 successful PoCs

5. Recommendations

Critical Fixes:

1. Implement public key authentication with digital signatures
2. Add perfect forward secrecy using ephemeral Diffie-Hellman
3. Encrypt private keys at rest with strong passphrases
4. Remove debug output from production code

Additional Hardening:

- Implement message signing for non-repudiation
- Add user authentication and access controls
- Use secure key storage mechanisms

6. Conclusion

The system implements strong cryptographic primitives but fails to address fundamental implementation security concerns. The identified vulnerabilities enable practical attacks that completely compromise system security in real-world deployment scenarios.

Deliverable 3B: Blue Team Incident Response Report

To: Professor of Cipher Corps
From: The PRNG Probers (Blue Team)
Date: October 20, 2025
Subject: Incident Response Report for PRNG Analysis System

1. Executive Summary

During the Red Team exercise, our PRNG Analysis System successfully detected and mitigated attempted attacks. While no system compromise occurred, we identified areas for improvement in our security posture and incident response capabilities.

2. Incident Timeline

- 14:30: Red Team begins reconnaissance on our system
- 14:45: First anomalous statistical patterns detected
- 15:10: Multiple failed attempts to poison statistical baseline
- 15:30: Red Team attempts to exploit weak LFSR configurations
- 15:45: Attack attempts cease after consistent detection

3. Attacks Detected & Mitigated

Statistical Test Poisoning

- **Attack Vector:** Specially crafted sequences designed to pass statistical tests
- **Detection:** Comparative baseline analysis revealed anomalous patterns
- **Mitigation:** Implemented challenge-response protocol

Algorithm Exploitation

- **Attack Vector:** Attempted exploitation of LFSR weaknesses
- **Detection:** Monitoring of input patterns and correlation analysis
- **Mitigation:** Rate limiting and input validation

4. System Hardening Recommendations

1. **Enhanced Input Validation:** Stricter validation of submitted sequences
2. **Behavioral Analysis:** Machine learning-based anomaly detection
3. **Authentication:** User authentication for system access
4. **Logging & Monitoring:** Enhanced security logging and real-time alerts

5. Lessons Learned

- Even analysis tools need robust security measures
- Input validation is critical for all external data sources
- Continuous monitoring is essential for detecting sophisticated attacks
- Defense in depth applies to all systems, not just production applications

Team Reflection

What We Learned

We progressed from breaking abstract ciphers to understanding how theoretical weaknesses manifest in real-world systems. The distinction between a "working" system and a "secure" system became profoundly clear. We learned that cryptography encompasses not just mathematical foundations but also critical implementation details that can become single points of failure.

Challenges Faced

The most significant challenge emerged during Phase 3, requiring us to adopt an attacker's mindset. Understanding that a system is vulnerable represents only the first step; crafting reliable exploits demands deeper insight. Defending our own system compelled us to consider threats we had previously overlooked.

Ethical Considerations

Our work involved constructing and deconstructing security tools. We reaffirm that these capabilities must be employed responsibly, exclusively with proper authorization, and solely within ethical contexts such as academic research and authorized penetration testing. "With great power comes great responsibility" constitutes the foundational ethic of security professionals.

Key Technical Insights

- **Implementation Matters:** Robust algorithms can be compromised by weak implementations
- **Side Channels:** Timing attacks and error messages can disclose critical information
- **Key Management:** Appropriate key storage proves equally important as cryptographic algorithms
- **Defense in Depth:** Multiple security layers are necessary for resilient systems
- **Statistical Testing:** Comprehensive analysis requires multiple complementary tests

Conclusion

This project provided comprehensive exposure to the complete cryptography lifecycle—from cryptanalysis and system design to offensive testing and defensive hardening. The experience emphasized that security represents a continuous process rather than a final destination, requiring ongoing vigilance, learning, and adaptation to emerging threats and vulnerabilities.